Python Structures

Structured Python for the Principled Programmer

Primordial Edition (not for distribution)

(Software release 40)

Duane A. Bailey

Williams College
September 2013

Contents

for Mary,
my wife and best friend

from friends, love.
from love, children
from children, questions
from questions, learning
from learning, joy
from joy, sharing
from sharing, friends

Preface

Python is a fun language for writing small programs. This book is about writing larger programs in Python. In more significant programs there begins to be a concern about how to structure and manipulate the data for efficient access. Python enjoys a rich set of data types for holding data—arrays, lists, dictionaries, and others—but our eventual focus will be what to do when these initial types are insufficient for the kinds of abstractions you might need in more complex tasks.

Python is frequently used to support the quick development of small programs that generate or filter data, or that coordinate the interactions of other programs. In this regard, Python can be thought of as a *scripting language*. Scripting languages are used for small quick tasks. Python's ease of use, however, has made is possible to write quite large programs, as well.

When programs become large, there is always a concern about efficiency. Does the program run quickly? Can the program manipulate large amounts of data quickly and with a small memory *footprint*? These are important questions, and if programmers are to be effective, they must be able to ask and answer questions about the efficiency of their programs from the point-of-view of data management. This book is about how to do that.

In the early chapters we learn about basic Python use. We then discuss some problems that involve the careful manipulation of Python's more complex data types. Finally, we discuss some object-oriented strategies to implement and make use of data structures that are either problem specific, or not already available within the standard Python framework. The implementation of new data structures is an important programming task, often at least as important as the implementation of new code. This book is about that abstract process.

Finally, writing Python is a lot of fun, and we look forward to thinking openly about a number of fun problems that we might use Python to quickly solve.

Chapter 0

Python

This book focuses on data structures implemented in the Python language. Python is a scripting language. This means that it can be used to solve a wide variety of problems and its programs are presented as scripts that are directly interpreted. In traditional programming languages, like C or Java, programs must first be compiled into a machine language whose instructions can be efficiently executed by the hardware. In reality, the difference between scripting and traditional programming languages is relatively unimportant in the design of data structures, so we do not concern ourselves with that here.[1]

Much of Python's beauty comes from its simplicity. Python is not, for example, a typed language—names need not be declared before they are used and they may stand for a variety of unrelated types over the execution of a program. This gives the language a light feel. Python's syntax makes it a useful desk calculator. At the same time it supports a modularity of design that makes programming-in-the-large possible.

The "pythonic" approach to programming is sometimes subtly different than other languages. Everything in Python is an object, so constants, names, objects, classes, and type hierarchies can be manipulated at runtime. Depending on your care, this can be beautiful or dangerous. Much of this book focuses on how to use Python beautifully.

Before we go much further, we review important features of the language. The intent is to ready you for understanding the subtleties of the code presented later. For those seeking an in-depth treatment, you should consider the wealth of resources available at `python.org`, including the Python Language Reference Manual (PLRM, `http://docs.python.org/3/reference/`). Where it may be useful, we point the reader to specific sections of the PLRM for a more detailed discussion of a topic.

0.1 Execution

There are several ways to use Python, depending on the level of interaction required. In interactive mode, Python accepts and executes commands one statement at a time, much like a calculator. For very small experiments, and

[1] For those who are concerned about the penalties associated with crafting Python data structures we encourage you to read on, in Part II of this text, where we consider how to tune Python structures.

while you are learning the basics of the language, this is a helpful approach to learning about Python. Here, for example, we experiment with calculating the golden ratio:

```
% python3
Python 3.1.2 (r312:79360M, Mar 24 2010, 01:33:18)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> golden = (1+math.sqrt(5))/2
>>> print(golden)
1.61803398875
>>> quit()
```

Experimentation is important to our understanding of the workings of any complex system. Python encourages this type of exploration.

Much of the time, however, we place our commands in a single file, or *script*. To execute those commands, one simply types `python3`[2] followed by the name of the script file. For example, a traditional first script, `hello.py`, prints a friendly greeting:

```
print("Hello, world!")
```

This is executed by typing:

```
python3 hello.py
```

This causes the following output:

```
Hello, world!
```

Successfully writing and executing this program makes you a *Python programmer*. (If you're new to Python programming, you might print and archive this program for posterity.)

Many of the scripts in this book implement new data structures; we'll see how to do that in time. Other scripts—we'll think of them as *applications*—are meant to be directly used to provide a service. Sometimes we would like to use a Python script to the extend the functionality of the operating system by implementing a new command or application. To do this seamlessly typically involves some coordination with the operating system. If you're using the Unix operating system, for example, the first line of the script is the *shebang line*[3]. Here are the entire contents of the file `golden`:

```
#!/usr/bin/env python3
import math
golden = (1+math.sqrt(5))/2
print(golden)
```

When the script is made executable (or *blessed*), then the command appears as

[2] Throughout this text we explicitly type `python3`, which executes the third edition of Python. Local documentation for your implementation of Python 3 is accessible through the shell command `pydoc3`.

[3] The term *shebang* is a slang for hash-bang or shell-comment-bang line. These two characters form a *magic number* that informs the Unix loader that the program is a script to be interpreted. The script interpreter (*python3*) is found by the `/usr/bin/env` program.

a new application part of the operating system. In Unix, we make a script executable by giving the user permission (`u+`) the ability to execute the script (`x`):

```
% chmod u+x golden
% golden
1.61803398875
```

Once the executable script is placed in a directory mentioned by the `PATH` environment variable, the Python script becomes a full-fledged Unix command.[4]

0.2 Statements and Comments

Python is a *line-oriented language*. Unlike Java and C, most statements in Python are written one per line. This one assumption—that lines and statements correspond—goes a long way toward making Python a syntactically simple language. Where other languages might use a terminal punctuation mark, Python uses the line's end (indicated by a *newline* character) to indicate that the statement is logically complete.

Some Python instructions—compound statements—control a *suite* of subordinate instructions with the suite consistently indented underneath the controlling statement. These *control statements* are typically entered across several lines. For example, the following single statement prints out factors of 100 from 1 up to (but not including) 100:

```
for f in range(1,100):
    if 100%f == 0:
        print(f)
```

The output generated by this statement is:

```
1
2
4
5
10
20
25
50
```

The `for` statement *controls* the `if` statement and the `if` statement *controls* the `print` statement. Each statement in a control statement's suite is consistently indented by four spaces.

When Python statements get too long for a single line, they may be continued on the next line *if* there are unmatched parentheses, curly braces, or square brackets. The line that follows is typically indented to just beyond the nearest unmatched paren, brace, or bracket. Long statements that do not have open parens can be explicitly continued by placing a backslash (`\`) just before the end of the line. The statement is then continued on the next line. Generally,

[4] In this text, we'll assume that the current directory (in Unix, this directory is `.`) has been placed in the path of executables.

explicitly continued long statements make your script unreadable and are to be avoided.

Comments play an important role in understanding the scripts you write; we discuss the importance of comments in Chapter 2. Comments begin with a hash mark (#) and are typically indented as though they were statements. The comment ends at the end of the line.

0.3 Object-Orientation

Much of the power of Python comes from the simple fact that everything is an object. Collections of data, methods, and even the most primitive built-in constants are objects. Because of this, effective programmers can be uniformly expressive, whether they're manipulating constants or objects with complex behavior. For example, elementary arithmetic operations on integers are implemented as *factory methods* that take one or more integer values and produce new integer values. Some *accessor* methods gain access to object state (e.g. the length of a list), while other *mutator* methods modify the state of the object (e.g. append a value to the end of the list). Sometimes we say the mutator methods are *destructive* to emphasize the fact an object's state changes. The programmer who can fully appreciate the use of methods to manipulate objects is in a better position to reason about how the language works at all levels of abstraction. In cases where Python's behavior is hard to understand, simple experiments can be performed to hone one's model of the language's environment and execution model.

We think about this early on—even before we meet the language in any significant way—because the deep understanding of how Python works and how to make it most effective often comes from careful thought and simple experimentation.

Exercise 0.1 *How large an integer can Python manipulate?*

A significant portion of this text is a dedicated to the design of new ways of structuring the storage of data. The result is a new *class*, from which new *objects* or instances of the class may be derived. The life of an object involves its allocation, initialization, manipulation, and, ultimately, its deletion and freeing of resources. The allocation and initialization of an object appears as the result of a single step of construction, typically indicated by calling the class. In this way, we can think of a class as being a factory for producing its objects.

```
>>> l = list() # construct a new list
>>> l.append(1)
>>> l
[1]
```

Here, `list` is the class, and `l` is a *reference* to the new list object. Once the list is constructed, it may be manipulated by calling methods (e.g. `l.append(1)`) whose execution is in the context of the object (here, `l`). Only the methods defined for a particular class may be used to directly manipulate the class's objects.[5]

It is sometimes useful to have a place holder value, a targetless reference, or a value that does not refer to any other type. In Python this value is `None`. Its sole purpose in Python is to represent "I'm not referring to anything." In Python, an important idiomatic expression guards against accessing the methods of a `None` or *null* reference:

```
if v is not None:
    ...
```

0.4 Built-in Types

Python starts up with a large number of *built-in* data types. They are fully developed data structures that are of general use. Many of these structures also support basic arithmetic operations. The list class, for example, interprets the addition operator (+) as a result of concatenating one list with another. These operations are, essentially, *syntactic sugar* for calling a list-specific special method that performs the actual operation. In the case of the addition operator, the method has the name `__add__`. This simple experiment demonstrates this subtle relationship:

```
>>> l1 = [1,2,3,4]
>>> l2 = [5,6,7,8]
>>> l1+l2
[1,2,3,4,5,6,7,8]
>>> l1.__add__(l2)
[1,2,3,4,5,6,7,8]
```

This *operator overloading* is one of the features that makes Python a simple, compact, and expressive language to use, especially for built-in types. Well designed data structures are easy to use. Much of this book is spent thinking about how to use features like operator overloading to reduce the apparent complexity of manipulating our newly-designed structures.

One of Python's strengths is its support for an unusually diverse collection of built-in data types. Indeed, a significant portion of this book is dedicated to leveraging these built-in types to implement new data types efficiently. The table of Figure 1 shows what the constant or *display* values of some of these built-in types look like.

0.4.1 Numbers

Numeric types may be integer, real, or complex (PLRM 3.2). Integers have no fractional part, may be arbitrarily large, and integer literals are written without

[5] Python makes heavy use of generic methods, like `len(l)` that, nonetheless, ultimately make a call to an object's method, in this case, the *special* method `l.__len__()`.

type	*literal*

numbers	`1`, `-10`, `0`
	`3.4`, `0.0`, `-.5`, `3.`, `1.0e-5`, `0.1E5`
	`0j`, `3.1+1.3e4J`

booleans	`False`, `True`

strings	`"hello, world"`
	`'hello, world'`

tuples	`(a,b)`
	`(a,)`

lists	`[3,3,1,7]`
dictionaries	`{"name": "duane", "robot": False, "height": 72}`
sets	`{11, 2, 3, 5, 7}`
objects	`<__main__.animal object at 0x10a7edd90>`

Figure 1 Literals for various Python types (PLRM 2.4).

a decimal point. Real valued literals are always written with a decimal point, and may be expressed in scientific notation. Pure complex literals are indicated by a trailing `j`, and arbitrary complex values are the result of adding an integer or real value. These three built-in types are all examples of the `Number` type.[6] The arithmetic operations found in the table of Figure 3 are supported by all numeric types. Different types of values combined with arithmetic operations are *promoted* to the most general numeric type involved to avoid loss of information. Integers, for example, may be converted to floating point or complex values, if necessary. The functions `int(v)`, `float(v)`, and `complex(v)` convert numeric and string values, `v`, to the respective numerics. This may appear to be a form of *casting* often found in other languages but, with thought, it is obvious that the these conversion functions are simply constructors that flexibly accept a wide variety of types to initialize the value.

C or Java programmers should note that exponentiation is directly supported and that Python provides both true,full-precision division (`/`) and division, truncated to the smallest nearby integer (`//`). The expression

```
(a//b)*b+(a%b)
```

always returns a value equal to `a`. Comparison operations may be *chained* (e.g. `0 <= i < 10`) to efficiently perform range testing for numeric types. Integer values may be written in base 16 (with an `0x` prefix), in base 8 (with an `0o` prefix), or in binary (with an `0b` prefix).

[6] The `Number` type is found in the `number` module. The type is rarely used directly.

operation	*interpretation*	*special method*
`{key:value,...}`	dictionary display (highest priority)	
`{expression,...}`	set display	
`[expression,...]`	list display	
`(expression,...)`	tuple display	
`x.attr`	attribute access	
`f(...)`	method call	
`x[i]`	indexing	`__getitem__`
`x[i:j:k]`	indexing by slice	`__getitem__`
`x**y`	exponentiation	`__pow__`
`~x`	bit complement	`__invert__`
`-x,+x`	negation, identity	`__neg__`, `__pos__`
`x/y`	division	`__truediv__`
`x//y`	integer division	`__floordiv__`
`x*y`	multiplication/repetition	`__mul__`
`x%y`	remainder/format	`__mod__`
`x+y, x-y`	addition, subtraction	`__add__`, `__sub__`
`x>>y`	shift left	`__lshift__`
`x<<y`	shift right	`__rshift__`
`x&y`	bitwise and, set intersection	`__and__`
`x^y`	bitw. excl. or, symm. set diff.	`__xor__`
`x|y`	bitwise or, set union	`__or__`
`x==y, x!=y`	value equality	`__eq__`, `__ne__`
`x<y, x<=y, x>=y, x>y`	value comparison	`__lt__`, `__le__`, etc.
`x is y, x is not y`	same object test	
`x in y, x not in y`	membership	`__contains__`
`not x`	logical negation	`__bool__`
`x and y`	logical and (short circuit)	
`x or y`	logical or (short circuit)	
`x if t else y`	conditional value	
`lambda args: expression`	anonymous function	
`yield x`	value generation (lowest priority)	

Figure 2 Python operators from highest to lowest priority (PLRM 6.15).

`and`	`as`	`assert`	`break`	`class`	`continue`	`def`	`del`	`elif`
`else`	`except`	`finally`	`for`	`from`	`global`	`if`	`import`	`in`
`is`	`lambda`	`nonlocal`	`not`	`or`	`pass`	`raise`	`return`	`try`
	`while`	`with`	`yield`		`False`	`None`	`True`	

Figure 3 The 33 words reserved in Python and not available for other use

0.4.2 Booleans

The boolean type, `bool`, includes the values `False` and `True`. In numeric expressions these are treated as values 0 and 1, respectively. When boolean values are constructed from numeric values, non-zero values become `True` and zero values become `False`. When container objects are encountered, they are converted to `False` if they are empty, or to `True` if the structure contains values. Where Python requires a condition, non-boolean expressions are converted to boolean values first.

Boolean values can be manipulated with the logical operations `not`, `and`, and `or`. The unary `not` operation converts its value to a boolean (if necessary), and then computes the logical opposite of that value. For example:

```
>>> not True
False
>>> not 0
True
>>> not "False" # this is a string
False
```

The operations `and` and `or` are short-circuiting operations. The binary `and` operation, for example, immediately returns the left-hand side if it is equivalent to `False`. Otherwise, it returns the right-hand side. One of the left or right are always returned and that value is never converted to a `bool`. For example:

```
>>> False and True
False
>>> 0 and f(1) # f is never called
0
>>> "left" and "right"
"right"
```

This operation is sometimes thought of as a *guard*: the left argument to the `and` operation is a condition that must be met before the right side is evaluated.

The `or` operator returns the left argument if it is equivalent to `True`, otherwise it returns its right argument. This behaves as follows:

```
>>> True or 0
True
>>> False or True
True
>>> '' or 'default'
'default'
```

We sometimes use the `or` operator to provide a default value (the right side), if the left side is `False` or empty.

Although the `and` and `or` operators have very flexible return values, we typically imagine those values are booleans. In most cases the use of `and` or `or` to compute a non-boolean value makes the code difficult to understand, and the same computation can be accomplished using other techniques, just as efficiently.

0.4.3 Tuples

The `tuple` type is a container used to gather zero or more values into a single object. In C-like languages these are called `structs`. Tuple constants or *tuple displays* are formed by enclosing comma-separated expressions in parentheses. The last field of a tuple can always be followed by an optional comma, but if there is only one expression, the final comma is required. An empty tuple is specified as a comma-free pair of parentheses (`()`). In non-parenthesized contexts the comma acts as an operator that concatenates expressions into a tuple. Good style dictates that tuples are enclosed in parentheses, though they are only required if the tuple appears as an argument in a function or method call (`f((1,2))`) or if the empty tuple (written `()`) is specified. Tuple displays give one a good sense of Python's syntactic flexibility.

Tuples are immutable; the collection of item references cannot change, though the objects they reference may be mutable.

```
>>> emptyTuple = ()
>>> president = ('barack','h.','obama',1.85)
>>> skills = 'python','snake charming'
>>> score = (('red sox',9),('yankees',2))
>>> score[0][1]+score[1][1]
11
```

The fields of a tuple are directly accessed by a non-negative index, thus the value `president[0]` is `'barack'`. The number of fields in a tuple can be determined by calling the `len(t)` function. The `len(skills)` method returns 2. When a negative value is used as an index, it is first added to the length of the tuple. Thus, `president[-1]` returns `president[3]`, or the president's height in meters.

When a tuple display is used as a value, the fields of a tuple are the result of evaluation of expressions. If the display is to be an assignable target (i.e. it appears on the left side of an assignment) each item must be composed of assignable names. In these *packed assignments* the right side of the assignment must have a similar shape, and the binding of names is equivalent to a collection of independent, simultaneous or *parallel* assignments. For example, the idiom for exchanging the values referenced by names `jeff` and `eph` is shown on the second line, below

```
>>> (jeff,eph) = ('trying harder','best')
>>> (jeff,eph) = (eph,jeff) # swap them!
>>> print("jeff={0} eph={1}".format(jeff,eph))
jeff=best eph=trying harder
```

This idiom is quite often found in sorting applications.

0.4.4 Lists

Lists are variable-length ordered containers and correspond to arrays in C or vectors in Java. List displays are collections of values enclosed by square brackets (`[...]`). Unlike tuples, lists can be modified in ways that often change their length. Because there is little ambiguity in Python about the use of square brackets, it is not necessary (though it is allowed) to follow the final element of a list with a comma, even if the list contains one element. Empty lists are represented by square brackets without a comma (`[]`).

You can access elements of a list using an index. The first element of list `l` is `l[0]` and the length of the list can be determined by `len(l)`. The last element of the list is `l[len(l)-1]`, or `l[-1]`. You add a new element `e` to the "high index" end of list `l` with `l.append(e)`. That element may be removed from `l` and returned with `l.pop()`. Appending many elements from a second list (or other iterable[7]) is accomplished with `extend` (e.g. `l.extend(l2)`). Lists can be reversed (`l.reverse()`) or reordered (`l.sort()`), *in place*.

Portions of lists (or tuples) may be extracted using *slices*. A slice is a regular pattern of indexes. There are several forms, as shown in the table of Figure 4. To delete or remove a value from a list, we use the `del` operator: `del l[i]` removes the element at the `i`th index, moving elements at higher indices downward.

Slices of lists may be used for their value (technically, an *r-value*, since they appear on the *right* side of assignment), or as a *target of assignment* (technically, an *l-value*). When used as an r-value, a copy of the list slice is constructed as a new list whose values are shared with the original. When the slice appears as a target in an assignment, that portion of the original list is removed, and the assigned value is inserted in place. When the slice appears as a target of a `del` operation, the portion of the list is logically removed.

slice syntax	*interpretation*
`l[i]`	the single element found at index `i`
`l[i:j]`	elements `i` up to but not including `j`
`l[i:]`	all elements at index `i` and beyond; `l[i:len(l)]`
`l[:j]`	elements 0 up to but not `j`; `l[0:j]`
`l[i:i]`	before element `i` (used as an l-value)
`l[:]`	a shallow copy of `l`
`l[i:j:s]`	every `s` element from `i`, to `i+s` up to (but not) `j`
`l[::-1]`	a copy of `l` reversed

Figure 4 Different slices of lists or tuples

Lists, like tuples, can be used to bundle targets for parallel list assignment. For example, the swapping idiom we have seen in the tuple discussion can be recast using lists:

```
>>> here="bruce wayne"
>>> away="batman"
>>> [here,away] = [away,here]
```

[7] Lists are just one example of an iterable object. Iterable objects, which are ubiquitous in Python, are discussed in more detail in Chapter 5.

method	*returned value*
`l.append(item)`	Add `item` to end of `l`
`l.extend(iterable)`	Add items from `iterable` to end of `l`
`l.clear()`	Remove all elements of `l`
`l.index(item)`	Index of first `item` in `l`, or raise `ValueError`
`l.insert(loc,item)`	Insert `item` into `l` at index `loc`
`l.pop()`	Value removed from high end of `l`
`l.reverse()`	Reverse order of elements in `l`
`l.sort()`	Put elements of `l` in natural order

Figure 5 Common `list` methods.

```
>>> here
'batman'
>>> away
'bruce wayne'
```

In practice, this approach is rarely used.

When the last assignable item of a list is preceded by an asterisk (`*`), the target is assigned the list of (zero or more) remaining unassigned values. This allows, for example, the splitting of a list into parts in a single packed assignment:

```
>>> l = [2, 3, 5, 7, 11]
>>> [head,*l] = l
>>> head
2
>>> l
[3, 5, 7, 11]
```

As we shall see in Chapter 4, lists are versatile structures, but they can be used inefficiently if you are not careful.

0.4.5 Strings

Strings (type `str`) are ordered lists of characters enclosed by quote (`'`) or quotation (`"`) delimiters. There is no difference between these two quoting mechanisms, except that each may be used to specify string literals that contain the other type of quote. For example, one might write:

```
demand = "See me in the fo'c'sle, sailor!"
reply = "Where's that, Sir?"
retort = 'What do you mean, "Where\'s that, Sir?"?'
```

In any case, string-delimiting characters may be protected from interpretation or *escaped* within a string. String literals are typically specified within a single line of Python code. Other *escape sequences* may be used to include a variety

escaped character	*interpretation*
`\a`	alarm/bell
`\b`	backspace
`\f`	formfeed
`\n`	newline
`\r`	return
`\t`	tab
`\v`	tab
`\'`	single quote
`\"`	double quote
`\\`	slash

Figure 6

of white space characters (see Figure 6). To form a multi-line string literal, enclose it in triple quotation marks. As we shall see throughout this text, triple-quoted strings often appear as the first line of a method and serve as a simple documentation form called a *docstring*.

Unlike other languages, there is no character type. Instead, strings with length 1 serve that purpose. Like lists and tuples, strings are indexable *sequences*, so indexing and slicing operations can be used. String objects in Python may not be modified; they are immutable. Instead, string methods are factory operations that construct new `str` objects as needed.

The table of Figure 7 is a list of many of the functions that may be called on strings.

method	*returned value*
`s.find(s2)`	Index of first occurrence of `s2` in `s`, or -1
`s.index(s2)`	Index of first `s2` in `s`, or raise `ValueError`
`s.format(args)`	Formatted representation of `args` according to `s`
`s.join(s1,s2,...,sn)`	Catenation of `s1+s+s2+s+...+s+sn`
`s.lower()`	Lowercase version of `s`
`s.replace(old,new)`	String with occurrences of `old` replaced with `new`
`s.split(d)`	Return list of substrings of `s` delimited by `d`
`s.startswith(s2)`	`True` when `s` starts with `s2`; `endswith` is similar
`s.strip()`	`s` with leading and trailing whitespace removed

Figure 7 Common string methods.

0.4.6 Dictionaries

We think of lists as *sequences* because their values are accessed by non-negative index values. The *dictionary* object is a container of key-value pairs (*associations*), where they keys are immutable*The technical reason for immutability is discussed later, in Section* **??**. and unique. Because the keys may not have any natural order (for example, they may have multiple types), the notion of a sequence is not appropriate. Freeing ourselves of this restriction allows dictionaries to be efficient mechanisms for representing discrete functions or *mappings*.

Dictionary displays are written using curly braces ({...}), with key-value pairs joined by colons (:) and separated by commas (,). For example, we have the following:

```
>>> bigcities = { 'new york':'big apple',
...               'new orleans':'big easy',
...               'dallas':'big d' }
>>> statebirds = { 'AL' : 'yellowhammer', 'ID' : 'mountain bluebird',
...                'AZ' : 'cactus wren', 'AR' : 'mockingbird',
...                'CA' : 'california quail', 'CO' : 'lark bunting',
...                'DE' : 'blue hen chicken', 'FL' : 'mockingbird',
...                'NY' : 'eastern bluebird' }
```

Given an immutable value, `key`, you can access the corresponding value using a simple indexing syntax, `d[key]`. Similarly, new key-value pairs can be inserted into the dictionary with `d[key] = value`, or removed with `del d[key]`:

```
>>> bigcities['edmonton'] = 'big e'
>>> for state in statebirds:
...   if statebirds[state].endswith('bluebird'):
...     print(state)
...
NY
ID
```

A list or *view*[8] of keys available in a dictionary `d` is `d.keys()`. Because the mapping of keys to values is one-to-one, the keys are always unique. A view of values is similarly available as `d.values()`; these values, of course, need not be unique. Tuples of the form `(key,value)` for each dictionary entry can be retrieved with `d.items()`. The order of keys, values, and items encountered in views is not obvious, but during any execution it is consistent between the views.

Sparse multi-dimensional arrays can be simulated by using tuples as keys:

```
>>> n = 5
>>> m = {}
>>> for a in range(1,n+1):
...     for b in range(1,n+1):
```

[8] A *view* generates its value "lazily." An actual list is not created, but *generated*. We look into generators in Section 5.1.

```
...         m[a,b] = a*b
...
>>> m
{(1, 2): 2, (3, 2): 6, (1, 3): 3, (3, 3): 9, (4, 1): 4, (3, 1): 3,
(4, 4): 16, (1, 4): 4, (2, 4): 8, (2, 3): 6, (2, 1): 2, (4, 3): 12,
(2, 2): 4, (4, 2): 8, (3, 4): 12, (1, 1): 1}
>>> m[3,4]
12
```

It is illegal to access a value associated with a key that is not found in the dictionary. Several methods allow you to get defaulted and conditionally set values associated with potentially nonexistent keys without generating exceptions.

The design and implementation of dictionaries is the subject of Chapter 14.

0.4.7 Sets

Sets are unordered collections of unique immutable values, much like the collection of keys in a dictionary. Because of their similarity to dictionary types, they are specified using curly braces (`{...}`), but the entries are simply immutable values—numbers, strings, and tuples are common. Typically, all entries are the same type of object, but they may be mixed. Here are some simple examples:

```
>>> newEngland = { 'ME', 'NH', 'VT', 'MA', 'CT', 'RI' }
>>> commonwealths = { 'KY', 'MA', 'PA', 'VA' }
>>> colonies13 = { 'CT', 'DE', 'GA', 'MA', 'MD', 'NC', 'NH',
...                'NJ', 'NY', 'PA', 'RI', 'SC', 'VA' }
>>> newEngland - colonies13 # new New England states
{'ME', 'VT'}
>>> commonwealths - colonies13 # new commonwealths
{'KY'}
```

Sets support union (`s1|s2`), intersection (`s1&s2`), difference (`s1-s2`), and symmetric difference (`s1^s2`). You can test to see if `e` is an element of set `s` with `e in s` (or the opposite, `e not in s`), and perform various subset and equality tests (`s1<=s2`, `s1<s2`).

We discuss the implementation of sets in Chapter 13.

0.5 Sequential Statements

Most of the real work in a Python script is accomplished by statements that are executed in program order. These manipulate the state of the program, as stored in objects.

0.5.1 Expressions

Any expression may appear on a line by itself and has no impact on the script unless the expression has some *side effect*. Useful expressions call functions that directly manipulate the environment (like assignment or `print()`) or they perform operations or method calls that manipulate the *state* of an object in the environment.

When Python is used interactively to run experiments or perform calculations expressions can be used to verify the state of the script. If expressions typed in this mode return a value, that value is printed, and the variable `_` is set to that value. The value `None` does not get printed, unless it is part of another structure.

0.5.2 Assignment (=) Statement

The assignment operator (`name = object`) is a common means of binding an object reference to one or more names. Unlike arithmetic operators, the assignment operation does not return a result that can be used in other calculations, but you can assign several variables *the same value*, by chaining together several assignments at once. In practice this is rarely used. Because of these constraints we'll think of the assignment as a stand-alone statement.

0.5.3 The `pass` Statement

In some cases we would like to have a statement or suite that does nothing. The Python `pass` statement serves the syntactic purpose of a statement but it does not generate any executable code. For example, it is sometimes useful to provide an empty method, either because the implementation has not yet been worked out or because the logic of the operation effectively does nothing. The ellipsis (`...`) statement is synonym that is often used when one wishes to indicate "the implementation has yet to be provided." (Don't confuse the ellipsis statement with the interactive Python continuation prompt.)

0.5.4 The `del` Statement

An important aspect of efficiently using memory is recycling or *collecting* unreferenced objects or *garbage*. The `del` statement allows the programmer to explicitly indicate that a value will not be used. When `del` is applied to a name, the effect is to remove any binding of that name from the environment. So, for example, we have:

```
>>> a = 1
>>> a
1
>>> del a
>>> a
NameError: name ’a’ is not defined
```

When `del` is applied to slices of lists or other container structures, the item references are removed from the container. For example,

```
>>> del l[10]
>>> del l[10:20]
```

Remember that `del` is statement, not an operator.

0.6 Control Statements

Python has a rich set of *control statements*—statements that make decisions or conditionally execute blocks or *suites* of statements zero or more times. The language supports an `if` statement, as well as rich iteration and looping mechanisms. To get a sense of typical usage, we might consider a simple script to print out perfect numbers: numbers that are equal to the sum of their nontrivial factors:

```
for number in range(1,100):
    sum = 0
    for factor in range(1,nnumber):
        if number%factor == 0:
            sum += factor
    if sum == number:
        print(number)
```

This code generates the output:

```
6
28
```

We review the particular logical and syntactic details of these constructs, below.

0.6.1 If statements

Choices in Python are made with the `if` statement. When this statement is encountered, the condition is evaluated and, if `True`, the statements of the following suite are executed. If the condition is `False`, the suite is skipped and, if provided, the suite associated with the optional `else` is executed. For example, the following statement set `isPrime` to `False` if `factor` divides `number` exactly:

```
if (number % factor) == 0:
    isPrime = False
```

Sometimes it is useful to choose between two courses of action. The following statement is part of the “hailstone” computation:

```
if (number % 2) == 0:
    number = number / 2
else:
    number = 3*number + 1
```

If the number is divisible by 2, it is halved, otherwise it is approximately tripled.

The `if` statement is potentially nested (each suite could contain `if` statements), especially in situations where a single value is being tested for a series of conditions. In these cases, when the indentation can get unnecessarily deep and unreadable, the `elif` reserved word is equivalent to `else` with a subordinate `if`, but does not require further indentation. The following two if statements both search a binary tree for a value:

```
if tree.value == value:
    return True
else:
    if value < tree.value:
        return value in tree.left
    else:
        return value in tree.right

if value == tree.value:
    return True
elif value < tree.value:
    return value in tree.left
else:
    return value in tree.right
```

Both statements perform the same decision-making and have the same performance, but the second is more pleasing, especially since the conditions perform related tests.

It is important to note that Python does not contain a “case” or “switch” statement as you might find in other languages. The same effect, however, can be accomplished using code snippets stored in a data structure; we’ll see an extended example of this later.

0.6.2 While loops

Python supports two basic types of iteration—`while` and `for`. Unlike other languages, these loop constructs work in dramatically different ways. The `while` loop of Python is most similar to its equivalent in C and Java-like languages: a condition is tested and, if `True`, the statements of the following suite are executed. These potentially change the state of the condition, so the condition is then retested and the suite is potentially re-executed. This process continues until the condition fails. The suite will never be executed if the condition is initially `False`. If the condition never becomes `False`, the loop never terminates. The following code would find an item in a to-do list.

```
item = todo.first
while item is not None:
    if task == item.task:
        break
    item = item.next
# Either item is None or task location
```

When the loop is finished, `item` is either `None` (if the task is not found in the to-do list), or is a reference to the list element that matches the task. The `break` statement allows exiting from the loop from within the suite. A `continue` statement would immediately jump to the top of the loop, reconsidering the condition. These statements typically appear within suites of an enclosed `if` (as above), and therefore are the result of some condition that happens mid-suite.

An unusual feature of Python’s `while` statement is the optional `else` suite which is executed if the loop exited because of a failure of the loop’s condition. The following code tests to see if `number` is prime:

```
f = 2
```

```
isPrime = True
while f*f <= n:
    if n%f == 0:
        isPrime = False
        break
    f = 3 if f == 2 else f+2
if isPrime:
    print(n)
```

Here, the boolean value, `isPrime`, keeps track of the premature exit from the loop, if a factor is found. In the following code, Python will print out the number only if it is prime:

```
f = 2
while f*f <= n:
    if n%f == 0:
        break
    f = 3 if f == 2 else f+2
else:
    print("{} is prime.".format(n))
```

Notice that the else statement *is* executed if the suite of the while statement is never executed. In Python, the `while` loop is motivated by the need to *search* for something. In this light, the `else` is seen as a mechanism for handling a failed search. In languages like C and Java, the programmer has a choice of addressing these kinds of problems with a `while` or a `for` loop. As we shall see over the course of this text, Python's `for` loop is a subtly more complex statement.

0.6.3 For statements

The `for` statement is used to consider values derived from an *iterable* structure.[9] It has the following form:

```
for item in iterable:
    suite
```

The value `iterable` is a source for a stream of values, each of which is bound to the symbol `item`. Each time `item` is assigned, `suite` is executed. Typically, this suite performs a computation based on `item`. The following prints the longest line encountered in a file:

```
longest = ""
for line in open("textfile","r"):
    if len(line) > len(longest):
        longest = line
print(longest)
```

As with the `while` statement, the `for` loop may contain `break` and `continue` statements which control the loop's behavior. The suite associated with the `else`

[9] One definition of an iterable structure is, essentially, that it can be used as the object of `for` loop iteration.

clause is only executed if no `break` was encountered—that is, if a search among iterable values failed. The following loop processes lines, stopping when the word `stop` is encountered on the input. It warns the user if the `stop` keyword is not found.

```
for line in open("input","r"):
    if line == "stop":
        break
    process(line)
else:
    print("Warning: No 'stop' encounted.")
```

Notice that we have used the `for` loop to *process*, in some manner, every element that appears in a stream of data.

Because of the utility of writing traditional loops based on numeric values, the `range` object is used to generate streams of integers which may be processed by a `for` loop. These streams can then be traversed using the `for` loop. The `range` object is constructed using one, two, or three parameters, and has semantics that is similar to that of array slicing.[10] The form `range(i,j)` generates a stream of values beginning with `i` up to but not `j`. If `j` is less than or equal to `i`, the result will be empty. The form `range(j)` is equivalent to `range(0,j)`. Finally, the form `range(i,j,s)` generates a stream whose first value is `i`, and whose successive values are the result of incrementing by `s`. This process continues until `j` is encountered or passed. As an example, the following loop generates indices into string of RNA nucleotides (letters `'A'`, `'C'`, `'G'`, or `'U'`), and translates triples into an equivalent protein sequence:

```
# rna contains nucleotides (letters A,C,G, U):
length = len(rna)
protein = ""
for startPos in range(0,length-2,3):
    triple = rna[startPos:startPos+3]
    protein += code[triple]
```

If the length of the RNA string is 6, the values for `start` are 0 and 3, but do not include any value that is 4 (i.e. `length-2`) or higher. The structure `code` could be a dictionary, indexed by nucleotide triples, with each entry indicating one amino acid letter:

```
code = { 'UUU' : 'F', 'AUG' : 'M', 'UUA' : 'L', ... }
```

While many `for` loops often make use of `range`, it is important to understand that `range` is simply one example of an iterable. A focus of this book is the construction of a rich set of iterable structures. Making the best use of `for` loops is an important step toward making Python scripts efficient and readable. For example, if you're processing the characters of a string, one approach might be to loop over the legal index values:

```
# s is a string
```

[10] In fact, one may think of a slice as performing indexing based on the variable of a `for` loop over the equivalent range.

```
for i in range(len(s)):
    process(s[i])
```

Here, `process(s[i])` performs some computation on each character. A better approach might be to iterate across the characters, `c`, directly:

```
# s is a string
for c in s:
    process(c)
```

The latter approach is not only more efficient, but the loop is expressed at an appropriate logical level that can be more difficult in other languages.

Iterators are a means of generating that values that are encountered as one traverses a structure. A *generator* (a method that contains a `yield`) is a method for generating a (possibly infinite) stream of values for consideration in a `for` loop. One can easily imagine a random number generator or a generator of the sequence of prime numbers. Perfect numbers are easily detected if one has access to the nontrivial factors of a value:

```
sum = 0
for f in factors(n):
    sum += f
if sum == n:
    print("{} is perfect.".format(n))
```

Later, we'll see how `for` loops can be used to compactly form comprehensions—literal displays for built-in container classes.

0.6.4 Comprehensions

One of the novel features of Python is the ability to generate *tuple*, *list*, *dictionary*, and *set* displays from from conditional loops over iterable structures. Collectively, these runtime constructions are termed *comprehensions*. The availability of comprehensions in Python allows for the compact on-the-fly construction of container types. The simplest form of list comprehension has the general form:

```
[ expression for v in iterable ]
```

where `expression` is a function of the variable `v`. Tuple, dictionary, and set comprehensions are similarly structured:

```
( expression for v in iterable ) # tuple comprehension
{ key-expression:value-expression for v in iterable } # dictionary
{ expression for v in iterable } # set comprehension
```

Note that the difference between `set` and `dictionary` comprehension is use of colon-separated key-values pairs in the dictionary construction.

It is also possible to conditionally include elements of the container, or to nest multiple loops. For example, the following defines an upper-triangular multiplication table:

```
>>> table = { (i,j):i*j for i in range(10) for j in range(10) if i < j }
>>> table[2,3]
6
>>> table[3,2]
```

```
KeyError: (3, 2)
```

Here are some examples of various comprehensions based on integer properties:

```
>>> n = 1000
>>> factors = [ f for f in range(1,n+1) if n % f == 0 ]
>>> mtable = { (a,b) : a*b for a in range(1,n+1) for b in range(1,n+1) }
>>> composites = { a*b for a in range(2,n+1) for b in range(2,n+1) }
>>> primeset = { x for x in range(2,n+1) } - composites
>>> primes = list(primeset)
>>> primes.sort()
```

It is important to understand that these are fully realized structures, as opposed to generable, iterable, or *lazily realized* streams of data.

0.6.5 The `return` statement

The `return` statement identifies the result of the containing method call. Whenever a `return` is encountered, the containing method terminates and the associated value is used as the result of the method's computation.

Because Python supports the parallel assignment of tuples of values, the `return` statement can effectively return multiple values by returning a tuple.

0.6.6 The `yield` operator

Python computation is often the result of many collaborating threads. One mechanism for supporting potential concurrency is the use of generators—threads of execution that perform computations on-demand, or *lazily*. The `yield` operator is the mechanism that a generator uses to communicate the next just-computed value. When the `yield` statement is encountered, the value is returned as the result of the request and the generator computation is temporarily suspended. When the next value is requested, the generator is restarted at the point after the yield. We discuss `yield` when we consider the implementation of generators in Section 5.1.

0.7 Exception Handling

When computations generate exceptional conditions—using an incorrect index or violating assumptions about the state of the program—it is important to handle the flow of control carefully so that the program can recognize the bad state and recover gracefully. Python, like most modern programming languages, provides a number of statements that are used to manage exceptional conditions.

When unexpected problems or failed assumptions occur in a program, the the interpreter can *throw* or *raise* an exception. Exceptions are objects that contain information about the state of the interpreter when the exception occurred. It may include, for example, a nested list of methods that are still computing (a *traceback*), a textual description of the error, and possibly additional information that might be helpful in understanding the context of the problem. The process of *handling* an exception involves searching through the currently outstanding methods for a method that is willing to *catch* the exception and continue processing. The following statements are useful in the exception handling process.

0.7.1 The `assert` statement

The `assert` statement checks to make sure that an assumption holds true. It has the form

```
assert condition [, description]
```

The condition is a statement that is expected to be `True`. If it is not, Python will raise an `AssertionError` that, if not caught (they rarely are), prints a *stack trace* followed by a description of the assertion that failed. If no `description` is provided in the assert statement, few details are available to the user. Thus, it is useful to provide an expression that describes how the condition failed. Typically it is a string. The statement generating the description is not evaluated if the assertion condition is `True`.

Here is a use of `assert` that signals an error when a data file does not contain any values:

```
count = 0
for line in open(filename,"r"):
    process(line)
    count += 1
assert count > 0,"The file '{}' was empty.".format(filename)
```

Processing an empty file causes the script to halt with the following message:

```
AssertionError: The file 'datafile' was empty.
```

Although the description is optional, it almost always helps if the description gives some hint as to what condition failed. Since the actual text of the condition is not available at runtime, as a bare minimum, that much is helpful.

0.7.2 The `raise` Statement

To manually throw an exception, we use the `raise` statement. Typically, the `raise` statement is given either a class of exceptions (e.g. `IndexError`), or a particular instance of an exception that should be thrown. It is also possible to re-raise an exception that has previously been caught.

```
raise exception [from exception]
raise exception-class [from cause]
raise
```

Each of these statements causes termination of methods found on the current runtime stack. The process of *unwinding* the call stack stops when a `try` statement is found that takes responsibility for catching the particular type of exception (see below).

The optional `from` modifier allows a high-level exception to indicate another exception (one that was typically caught) that has lead to the failure. Lacking any information, a lone `raise` statement simply throws the last exception that was caught.

The `assert` statement, as described above, raises a particular type of exception. The `raise` statement allows programmers to carefully control for very specific types of conditions. As we shall see momentarily the conditions may not indicate errors in logic, but may simply signal a need to change the path of execution. The `for` loop, for example, makes use of the `StopIteration` exception as part of normal processing.

0.7.3 The `try` Statement

Python does not require the catching of exceptions. Generally, though, resilient programs will catch and recover from exceptional behaviors in subordinate code. The `try` statement is responsible for indicating how exceptions are to be handled:

```
try:
    suite
[except [exception-class [as exception]]:
    suite1]*
[else:
    suite2]
[finally:
    suite3]
```

Under the `try` statement, the first suite is executed. If no exception is raised during the suite's execution the optional suite marked `finally` (`suite3`) wraps things up.

If an exception is raised, processing halts and the class of the exception is compared to the classes that appear in any of the `except` clauses, in the order listed. The suite associated with the first matching `except` clause (e.g. `suite1`) is executed. If the particular exception instance is needed during processing, the `as` keyword identifies a variable that is bound to the particular exception. If no exceptions are raised during the exception handling, the `finally` suite is executed.

If the `try` statement wishes to catch all exceptions, but fails to list a thrown exception in one of the `except` clauses, the `else` suite (`suite2`) should be provided. The finally suite (`suite3`), if provided, is executed.

To summarize, the `try` suite is executed. If an exception is raised, a matching except clause handles the error. If the exception is not explicitly listed, the `else` suite handles the error. The `finally` suite will always eventually be called, unless an exception is raised that is not handled by this `try`.

As we noted, Python depends heavily on exception handling to implement normal control flow. For example the `for` loop

```
for item in container:
        print(item)
```

is actually a `while` construct with the following form:

```
try:
    iterator = iter(conainer)
    while True:
        item = next(iterator)
        print(item)
except StopIteration:
    pass
```

Knowing this helps us better understand the subtle techniques that are used to control `for` loops. We'll see more on this topic when we consider iteration, in Chapter 5.

0.8 Definitions

What makes Python more than a simple calculator is the ability to define new methods and classes. Methods describe how computations may be performed, while classes describe how new objects are constructed. These are important tasks with subtleties that give Python much of its expressiveness and beauty. Like most beautiful things, this will take time to appreciate. We will touch on the most important details of method and class definition, here, but much of this book may be regarded as a discussion of how these important mechanisms may be used to make programs more efficient. The reader is encouraged to scan through this section quickly and then return at a later time with more experience.

0.8.1 Method Definition

In Python, we will think of functions and procedures informally as methods, as in "methods for doing things." New methods are declared using the `def` statement. This statement describes a pattern (or *signature*) for calling the method and provides a subordinate suite of commands that make up its *body*. Here are three examples of simple arithmetic functions which we have stored in a file named `arith.py`:

```
def odd(n):
    """Return True iff an integer n is odd."""
    return n%2 == 1

def hail(x):
    """From x, compute the next term of the 'hailstone sequence'"""
    if odd(x):
        return 3*x+1
    else:
        return x//2
```

```
def hailstorm(n):
    """Returns when the hailstone sequence, beginning at n, reaches 1"""
    while n != 1:
        n = hail(n)
```

In the definition of `odd`, `n` is a *formal parameter*. When the `odd` function is called, `n` stands in for the *actual parameter* passed to the method. The formal parameter names provide a way for the programmer to describe a computation before there are any actual parameter values in hand. This is a common practice in most sciences: when computing the roots of a polynomial $a \cdot x^2 + b \cdot x + c$ we can perform the calculations $(b \pm \sqrt{b^2 - 4ac})/2a$. Here, the values a, b, and c are formal parameters. When we have a particular problem to solve, say solving $3x^2 + 11x + 10 = 0$, the coefficients 3, 11, and 10 are the actual values we use in our root calculation.

If the first line of a definition is a string (as it often is) it is a documentation string or *docstring*. This is the value returned whenever a description of the `odd` definition is called for. For example, typing

```
pydoc3 arith.odd
```

in the shell, or

```
help(odd)
```

directly in Python will use the docstring to document the functionality of the definition. It is idiomatic Python to enclose docstrings in triple quotes to allow documentation to extend over many lines, even if the documentation currently takes only one line. The idiom is so strong that many programmers immediately associate triple quoted strings as documentation whether or not that is the actual purpose.

In this text, when we discuss functions or procedures we will use those terms to emphasize whether a method intends on returning a useful value (a function) or not (a procedure). Typically, procedures have side effects (printing output, modifying structures, etc.) while functions quietly compute and return a value without modifying their environment. The `return` statement—executed when the procedure or function is finished—specifies the resulting method value. Where the method was called, this value is used in its place. If no `return` statement is given in the suite of a method, or if there is no value specified in a `return`, the procedure returns the value `None`. The `odd` method, above, returns an integer. The Python `print` method, however, is called like a procedure. It returns the value `None`.

The procedure `hailstorm`, above, is a function returns the value 1 if it is ever encountered. There is no obvious reason to believe that this should happen for any arbitrary value, but there is considerable evidence that `hailstorm` always eventually returns. The beauty of the `hailstorm` method lies in the mathematics, not the way it is defined.

Generally, the number of formal and actual parameters must match and the order of the actual and formal parameters should be consistent for correct interpretation. This requirement can be relaxed by providing default values for one or more of the trailing formal parameters. If the corresponding actual parameter is not provided, the default value is used as the actual.[11] Here is a function that counts the number of vowels that appear in a word:

```
def vowelCount(word,vowels='aeiouy'):
    """return the number of vowels appearing in word
    (by default, the string of vowels is 'aeiouy')"""
    occurances = [c for c in word.lower() if c in vowels]
    return len(occurances)
```

and here is how the `vowelCount` function might be called:

```
>>> vowelCount("ytterbium") # english
4
>>> vowelCount("cwm","aeiouwy") # welsh
1
```

If the ordering of arguments is unsatisfactory, or if a function call would benefit from making the association of formal and actual parameters explicit, the actual argument may be specified as a keyword assignment:

```
>>> vowelCount(vowels='aeiouwy',word='cwm')
1
```

In any case, the non-defaulted formal parameters must all have an associated actual parameter. You can see a nice example of this style of programming in the description of Python's built-in `print` method.

Python also provides two special formal parameters that gather together extra actual parameters. The following procedure uses the `*args` form (the asterisk is important; the keyword `args` is typical) to gather all actual parameters that have not otherwise been assigned to a formal parameter. The result is the sum of all the parameters.

```
def sum(*args):
    """Sum all the values provided in the call."""
    total = 0
    for value in args:
        total += value
    return total
```

allows:

```
>>> sum()
0
>>> sum(1,2,3)
6
```

Obviously, one may only specify one `*args`-style formal parameter.

When actual parameters mention keywords that are not names of formal parameters, the `**kargs` form (here, the `**` is important; `kargs` is typical and nearly universal) will gather together all the keyword–value pairs into a dictionary. The following method translates a string of words using a dictionary of word–translation pairs:

[11] There are some subtleties here stemming from the fact that the default value—which may be mutable—is constructed once, at the time the method is defined. Be aware.

```
def translate(phrase, **table):
    """Translate a phase using table, a dictionary."""
    words = phrase.split()
    rewritten = [(table[w] if w in table else w) for w in words]
    return " ".join(rewritten)
```

This code might be used as:

```
>>> translate('now is the time')
'now is the time'
>>> translate("for all good men",all="some",men="people")
'for some good people'
>>> translate("to wear snakeskin belts",cotton="silk")
'to wear snakeskin belts'
```

Here, as well, there may be only one `**kargs`-style formal parameter. When both `*args` and `**kargs` are specified, the `**kargs` formal is always second.

It is also possible to specify a list (`*args`) or dictionary (`**table`) as an *actual* parameter, from which positional arguments or keyword-value pairs are drawn for the function call. This allows the above functions to be called in very flexible ways:

```
>>> factorSet = {2,4,7,14}
>>> sum(1,*factorSet)
28
>>> swedish = {'all':'alla', 'boots':'stovlar',
               'good':'goda', 'is':'ar', 'now':'nu',
               'people':'manniskor', 'snakeskin':'ormskinn',
               'the':'det', 'time':'dags', 'to':'att',
               'wear':'bara'}
>>> translate('all snakeskin boots',**swedish)
'alla ormskinn stovlar'
```

Notice that because we are not in the context of keyword processing when we define the `swedish` dictionary, the keywords must be explicitly written as strings. Before, the expression `cotton='silk'` generated a dictionary entry of the form `'cotton':'silk'`.

It should be noted that built-in functions (e.g. `print`) are defined in a way that is consistent with the defining mechanism we've just seen. This allows programmers to use these pre-built functions in a variety of novel ways. Be aware, though, that ingenuity can obscure beauty.

0.8.2 Method Renaming

Methods, in Python, are simply objects that can be *called* with actual parameters. When we us the name of a method, say `print`, it is simply a reference the method. Because of this, a method name may be used as an l-value in an assignment to bind another name to the method. In some cases it is useful to pass a method reference as an actual parameter to a function. We will make considerable use of this in the next section as well as in Chapter **??**, where we discuss sorting.

0.8.3 Lambda Expressions: Anonymous Methods

Occasionally, it is important to be able to construct small functions at runtime for a particular use. For example, many of Python's sorting utilities make use of a *key function* to extract a sort keys from the values to be sorted. In these circumstances, it is common to make use of a *lambda expression*,[12] a single value-producing computation wrapped in a nameless method. Lambda expressions are encapsulations of very simple functions as nameless objects. These objects may be bound to names or passed around as we might with any other value.

Here, is a lambda expression for the hailstone function described above:

```
>>> hail = lambda x: 3*x + 1 if ((x%2) == 1) else x//2
>>> print(hail(27))
82
```

Lambda expressions must compute a value which is (implicitly) returned. Lambda expressions may be written to take any number of formal parameters. However, when no parameters are required, the parentheses are dropped (this is different than normal method definition).

0.8.4 Class Definitions

It is sometimes the case—and in this book, *often* the case—that none of the built-in data types is sufficient to capture the structure of the data needed for in a script. In these situations, we must describe a new *class* of data objects. In Python this is accomplished with the `class` keyword. Generally, class definitions have the following form:

```
class name [(super [, super]* [, metaclass=builder])]:
    suite
```

Here, `name` is the name of the class of objects. The definitions provided in the suite describe the fields and methods associated with the structure. Collectively, we think of these as *attributes*.

New instances of the class are constructed or *initialized* by calling the class. This is a subtle point: the class, itself, is callable and produces, like a factory, objects or *instances* whose type is the class. Each of these instances gets a dedicated copy of each of the attributes associated with the class. In general, the attributes associated with an object are dynamic. So, for example, new attributes may be associated with an object by assignment. Much of this text is dedicated to understanding how class definitions are effectively leveraged to describe different types.

Class definition is a difficult but powerful programming device. The eager programmer is encouraged to bookmark the documentation on class definition

[12] Use of the word "lambda," here, is a reference to the lambda calculus, a formal system for understanding computation. In that formalism, anonymous functions are often reasoned about and treated as objects. The analogy here is entirely appropriate.

in the PLRM as a backdrop to this text's more extended discussion on class design.

0.8.5 Modules and Packages

The main organizational mechanism for Python definitions is a *module*. The name of the module is derived from the name of the file that contains the module's definitions. (In our `hailstorm` example, on Page 24, we essentially wrote a module called `arith`.) When Python looks up a name, global definitions are found within the module that contains the reference. In this way, similar names in two different files can be distinguished by the name of the module that contains them. So, for example, `math.sin` is a reference to the `sin` function, defined in the `math` module, and `arith.odd` is a reference to our odd-integer-detector discussed earlier.

If we think of files as defining modules, then the directories that contain the files define packages. The hierarchy of directories, then, corresponds to an equivalent hierarchy of packages and modules. This book describes the modules found within a specific package, the `structure` package. The art of package design is beyond the charge of this book, but many good examples are available from the Python community.

To make use of definitions outside the current module, we make use of the `import` command. To include, for example, the definitions of math functions we type:

```
import math
```

This makes all the `math` definitions available for use. If necessary, we can selectively import definitions. To access only the a few definitions from `math`, we could

```
from math import cos, sin, pi
```

which makes `cos`, `sin`, `pi` valid definitions in the current module. Though many other definitions of the `math` module might have aided in the definition of `pi` (for example, `math.acos(-1)`), they are not accessible unless they are explicitly imported.[13] All of this helps keep our dictionary of names or *namespace* uncluttered.

Sometimes it is useful to incorporate an external definition into the definitions of the current module but in a manner that avoids a conflicting name. For example, we might want to be able to call the `sin` function with the name `sine`. This is accomplished with

```
from math import sin as sine
```

From this point, onward, we can use `sine` anywhere where `math.sin` would have otherwise been useful.

Finally, it some sometimes useful to import *all* the definitions made in a module and incorporate all of those as local definitions. This is accomplished with

[13] These definitions, though not directly accessible, are accessible to functions you *do* import.

```
from math import *
```

Given this, one can call `sin` and `cos` and make use of `pi`, without the `math` prefix. This can be dangerous: the `math` module may have definitions that conflict with definitions we have already made in this module. Do you have use for the name `e`? So does `math`.

0.8.6 Scope

As we have seen above, Python accesses the objects (e.g. data, methods, classes) in our programs through *names*. When we use a name as a value Python must work hard to find the correct definition of the name. In method definition, for example, the names of formal parameters can obscure definitions of those names made outside the method. The visibility of a name throughout a script determines its *scope* and the rules that determine where Python finds definitions are called *scope rules*.

Names come into existence through the *binding* of the name to an object reference. Examples of bindings are

1. Assignment of the name to a value (i.e. using the assignment operator, =),
2. Names of formal parameters in a method definition,
3. Use of the name in a `for` loop,
4. Definition of the name as a method (`def`) or class (`class`), or
5. Values imported with the `from ... import` statements.

(There are other more obscure binding operations as well. For those details see Section 4.1 of the Python Language Reference Manual.)

When we use or *reference* a name, Python searches for the binding that is in the tightest containing *block* of code. A block is a collection of statements that are executed as a unit.

1. A function definition (beginning `def name...`),
2. A class definition (beginning `class name...`), or
3. The containing module (i.e. the file the contains the definition).

The *scope* of a name is that portion of code that where the name is *visible*. So, for example, the scope of any name bound within a function is the entire body of the function. The scope of a name bound in a class is the set of executable statements of the class (*not* including its functions). Names bound at the module level are visible everywhere within the module, so they are called *global* names.

It is sometimes useful to bind names locally in a way that might hide bindings in a wider scope. For example, formal parameters in method definitions are local bindings to actual values. References to these names will resolve to the actual parameters, not to similarly named bindings found in a larger scope.

The *environment* of a statement is the set of all scopes that contain the statement. The environment of a statement contains, essentially, all the code that might contain bindings that could be referenced by the statement. This environment is, effectively, an encapsulation of all information that helps us understand the meaning or *semantics* of a statement's execution. The *execution stack* keeps track of the nesting of the execution of methods and indirectly annotates the environment with bindings that help us understand the information necessary to execute the statement.

When Python makes a reference to a name bound in the tightest containing scope, we say the binding and any reference are *local*. Again, because scoping is implied by the location of bindings, it is occasionally difficult to re-bind objects to names that were originally bound outside the current local scope. By default, bindings are made within the local scope. So, for example, when we make the following definition:

```
def outer():
    v = 1
    def inner():
        v = 2
    inner()
    return v
```

a call to `outer()` returns 1.

To force a binding to the name just visible outside the local scope, we give Python a hint by indicating the a name is *nonlocal*. In this definition of `outer`, the value returned is 2, not 1:

```
def outer():
    v = 1
    def inner():
        nonlocal v
        v = 2
    inner()
    return v
```

When there are global values (including built-ins and imported values) that you wish to rebind, we declare those bindings with `global`:

```
v = 0
def outer():
    v = 1
    def inner():
        global v
        v = 2
    inner()
    return v
```

Here, a call to `outer()` returns 1, but the global value, `v`, becomes 2.

In practice, Python's scope rules lead to the most obvious behavior. Because of this, the use of scope hinting is rarely required. The need to use scope hints, in most cases, suggests an obscure and, perhaps, ugly approach.

0.9 Discussion

Python is a fun, modern, object-oriented scripting language. We can write big programs in Python, but if we need to get good performance from our programs we need to organize our data effectively. Before we can be investigate how to build efficient data structures, it is vitally important that we understand the fundamentals of language, itself.

Everything in Python is an object, even the most primitive types and values. Objects are accessed through names that stand for references to objects. All data—including classes, objects, and methods—are accessed through these names. All features of an object are associated with attributes, some of which are determined when the object is constructed. All objects are produced by a class, which determines their type. Since the objects that are referenced by names can be dynamically changed, so can their types.

Python provides a few constructs that allow us to control the flow of execution. Some, like the `if` statement, parallel those from other languages. The looping statements, however, differ in significant ways—ways that, as we shall see, support very powerful programming abstractions.

Python is a subtle language that takes time to understand. The reader is encouraged to return to this chapter or, more importantly, the resources available at `python.org`, as a means of reconsidering what has been learned in light of the new ideas we encounter in this brief text.

Self Check Problems

Solutions to these problems begin on page **??**.

0.1 Where are the answers for these "self-check" problems found?

0.2 Python is an *interpreter*. What does that mean?

0.3 Should you read the entire text?

0.4 Where can one find the resources associated with this text? Is it possible to use this software for our own applications?

0.5 How do we know when code is *beautiful*?

0.6 Where is the *Python Language Reference Manual*?

0.7 Typically, we start an interactive Python session by typing `python3`. How do we typically terminate an interactive session?

0.8 What is the purpose of the shell command `pydoc3`?

0.9 Many programmers use Python 2. Why is this book about Python 3?

0.10 Why would we write data structures in Python, where interpretation can result in slow implementations?

Problems

Selected solutions to problems begin on page **??**.

0.1 Write a script that prints `Hello, world.`

0.2 Write scripts for each of the following tasks:

1. Print the integers 1 through 5, one per line.
2. Print the integers 5 downto 1, all on one line. You may need to learn more about the statment `print` by typing `help(print)` from within Python.
3. Print the multiples of 3 between 0 and 15, inclusive. Make it easy for someone maintaining your script to change 3 or 15 to another integer.
4. Print the values `-n` to `n`, each with its remainder when divided by `m`. Test your program with `n=m` and `m=3` and observe that `n` modulo `m` is not necessarily `-n` modulo `m`.

0.3 What does the statement `a=b=42` do?

0.4 Assume that `neg=-1`, `pos=1`, and `zero=0`. What is the result of each of the following expressions?

1. `neg < zero < pos`
2. `neg < zero > neg`
3. `(zero == zero) + zero`
4. `(zero > neg and zero < pos)`
5. `zero > (neg and zero) < 1/0`

0.5 What is an *immutable object*? Give two examples of immutable object types.

0.6 What is the difference between a name, a reference, and an object?

0.7 Indicate the types of the following expressions, or indicate it is unknown:

1. i
2. 1729
3. 'c'
4. "abc"
5. "abc"[0]
6. ('baseball',)
7. 'baseball','football','soccer'
8. 'US' : 'soccer', 'UK' : 'football'
9. i*i for i in range(10)

Chapter 1

The Object-Oriented Method

The focus of language designers is to develop programming languages that are simple to use but provide the power to accurately and efficiently describe the details of large programs and applications. The development of an *object oriented* model in Python is one such effort.

Throughout this text we focus on developing data structures using *object-oriented programming*. Using this paradigm the programmer spends time developing templates for structures called *classes*. The templates are then used to construct *instances* or *objects*. A majority of the statements in object-oriented programs involve applying functions to objects to have them report or change their state. Running a program involves, then, the construction and coordination of objects. In this way, languages like Python are *object-oriented*.

OOP: Object-oriented programming.

In all but the smallest programming projects, *abstraction* is a useful tool for writing working programs. In programming languages including Java, Scheme, and C, and in the examples we have seen so far in Python the details of a program's implementation are hidden away in its procedures and functions. This approach involves *procedural abstraction*. In object-oriented programming the details of the implementation of data structures are hidden away within its objects. This approach involves *data abstraction*. Many modern programming languages use object orientation to support basic abstractions of data. We review the details of data abstraction and the concepts involved in the design of *interfaces* for objects in this chapter.

1.1 Data Abstraction and Encapsulation

If you purchase a pizza from Hot Tomatoes in Williamstown, Massachusetts, you can identify it as a pizza without knowing its ingredients. Pizzas have crusts and toppings, and their great hot or cold. The particular ingredients in a pizza are, actually, typically of little concern to you. Of course, Hot Tomatoes is free to switch from using sugar to honey to help its crusts rise, as long as the taste of the pizza is preserved.[1] The ingredients list of a pizza, and its construction are details that typically don't interest the consumer.

Likewise, it is often unimportant to know how data structures are *implemented* in order to appreciate their *use*. For example, most of us are familiar

[1] The author once saw the final preparations of a House pizza at Supreme Pizza which involved the sprinkling of a fine powder over the fresh-from-the-oven pie. Knowing, exactly, what that powder was would probably *detract* from the taste of the pie.

with the workings or *semantics* of lists or strings of characters, but, if pressed, we might find it difficult to describe their *mechanics*: *Do all consecutive locations in a list appear close together in memory in your computer, or are they far apart?* The answer is: *it is unimportant*. As long as the list behaves like an list, or the string behaves like a string, we are happy. The less one knows about how lists or strings are implemented, the less one becomes dependent on a particular implementation. Another way to think about this abstractly is that the data structure lives up to an implicit "contract": *a string is an ordered list of characters*, or *elements of a list may be accessed in any order*. The implementor of the data structure is free to construct it in any reasonable way, as long as all the terms of the contract are met. Since different implementors are in the habit of making very different implementation decisions, anything that helps to hide the implementation details—any means of using *abstraction*—serves to make the world a better place to program.

When used correctly, object-oriented programming allows the programmer to separate the details that are important to the user from the details that are only important to the implementation. Later in this book we shall consider the very general behavior of data structures; for example, in Section **??** we will study structures that allow the user only to remove the most recently added item. Such behavior is inherent to our most abstract understanding of how the data structure works. We can appreciate the unique behavior of this structure even though we haven't yet discussed how these structures might be implemented. Those abstract details that are important to the user of the structure—including abstract semantics of the methods—make up its *contract* or *interface*. The interface describes the abstract behavior of the structure. Most of us would agree that while strings and lists are very similar structures, they behave differently: you can shrink or expand a list, while you cannot directly do the same with a string; you can print a string directly, while printing a list often involves explicitly converting each of its elements to an equivalent string-based representation, and then printing them in sequence. These distinctions suggest they have distinct abstract behaviors; there are distinctions in the design of their interfaces.

The unimportant details hidden from the user are part of what makes up the *implementation*. We might decide (see Figure **??**) that a list is to be constructed from a large array of elements with an attendant element count. Alternatively, we might think of the list as a sequence of element references, linked together like train cars. Both of these approaches are perfectly satisfactory, but there are trade-offs. The first implementation (called a *vector* in some languages) has its length stored explicitly, while the length of the second implementation (called a *linked list*) is implied. It takes longer to determine the length of a linked list because we have to count the rail-car/element-references. On the other hand, the linked list can easily grow, while the vector grows by fits-and-starts, based on the need to allocate new memory. If implementors can hide these details, users do not have to be distracted from their own important design work. As applications mature, a fixed interface to underlying objects allows alternative implementations of the object to be considered.

Data abstraction in languages like Python allows a structure to take responsibility for its own state. The structure knows how to maintain its own state without bothering the programmer. For example, if two strings have to be concatenated into a single string structure, a request will have to be made for a new allotment of memory. Thankfully, because strings know how to perform operations on themselves, the user doesn't have to worry about managing memory.

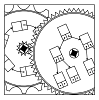
1.2 The Object Model

To facilitate the construction of well-designed objects, it is useful to have a design method in mind. As alluded to earlier, we will often visualize the data for our program as being managed by its objects. Each object manages its own data that determine its state. A point on a screen, for example, has two coordinates. A medical record maintains a name, a list of dependents, a medical history, and a reference to an insurance company. A strand of genetic material has a sequence of base pairs. To maintain a consistent state we imagine the program manipulates the data within its objects only through *method calls* on the objects. A string might receive a message "tell me your length," while a medical record might receive a "change insurance" message. The string message simply accesses information, while the medical record method may involve changing several pieces of information in this and other objects in a consistent manner. If we directly modify the reference to the insurance company, we may forget to modify similar references in each of the related structures (for example, a list of subscribers might have to be updated in both insurance companies). For large applications with complex data structures, it can be extremely difficult to remember to coordinate all the operations that are necessary to move a single complex object from one consistent state to another. We opt, instead, to have the designer of the data structure provide us a method for carefully moving between states; this method is activated in response to a high-level method call applied to the object.

This text, then, focuses on two important topics: (1) how we implement and evaluate objects with methods that are logically complex and (2) how we might use the objects we create. These objects typically represent *data structures*, our primary interest. Occasionally we will develop *control structures*—structures whose purpose is to control the manipulation of other objects. Control structures are an important concept and are described in detail in Chapter **??**.

1.3 Object-Oriented Terminology

In Python, data abstraction is accomplished through *encapsulation* of data in an *object*—an instance of a *class*. Objects save their state in an object's *attributes* Attributes and methods of an object may be declared with the intent of being accessed externally; they are considered *public*. With some care we can express our intent to limit access to them, in which case we think of them as being *private*. A typical class declaration is demonstrated by the following simple class that keeps track of the location of a rectangle on a screen:



Rect

```
class Rect:
    """A class describing the geometry of a rectangle."""
    __slots__ = [ "_left", "_top", "_width", "_height" ]

    def __init__(self, midx = 50, midy = 50, width = 100, height = 100):
        """Initialize a rectangle.
           If not specified otherwise, the rectangle is located
           with it's upper-left at (0,0), with width and height 100."""
        self._width = int(width)
        self._height = int(height)
        self._left = int(midx) - (self._width//2)
        self._top = int(midy) - (self._height//2)

    def left(self):
        """Return the left coordinate of the rectangle."""
        return self._left

    def right(self):
        """Return the right coordinate of the rectangle."""
        return self.left()+self.width()

    def bottom(self):
        """Return the bottom coordinate of the rectangle."""
        return self.top()+self.height()

    def top(self):
        """Return the top coordinate of the rectangle."""
        return self._top

    def width(self):
        """Return the width of the rectangle."""
        return self._width

    def height(self):
        """Return the height of the rectangle."""
        return self._height

    def area(self):
        """Return the area of the rectangle."""
        return self.width()*self.height()

    def center(self):
        """Return the (x,y) coordinates of the center of the Rect."""
        return (self.left()+self.width()//2,self.top()+self.height()//2)

    def __repr__(self):
        """Return a string representation of the rectangle."""
        (x,y) = self.center()
        return "Rect({0},{1},{2},{3})".format(
            x, y, self.width(), self.height())
```

Near the top of the class definition, we declare the *slots*, or a list of names of all the attributes we expect this class to have. Our `Rect` object maintains four pieces of information: its left and top coordinates, and its width and height, so four slots are listed. The declaration of `__slots__` is not required, but it serves as a commitment to using only these attributes to store the characteristics of the rectangle object. If we attempt to assign a value to an attribute that does not appear in the attribute list, Python will complain. This is one of the first of many principles that are meant to instill good Python programming practice.

Principle 1 *Commit to a fixed list of attributes by explicitly declaring them.*

The method `__init__` is responsible for filling in the attributes of a newly allocated `Rect` object. The formal comments (found in triple quotes) at the top of each method are pre- and postconditions. We will discuss these in detail in Chapter 2.) The `__init__` method[2] is the equivalent of a *constructor* in other languages. The method initializes all the attributes of an associated object, placing the object into a predictable and consistent initial state. To construct a new `Rect` users make a call to `Rect()`. Parameters to this method are passed directly to the `__init__` method. One extra parameter, `self` is a reference to the the object that is to be initialized. Python makes it fairly easy to pick-and-choose which parameters are passed to the constructor and which are left to their default values. Without exception, attributes are accessed in the context of `self`.

It is important to notice, at this point, that there were choices on how we would represent the rectangle. Here, while we allow the user to construct the rectangle based on its *center* coordinate, the rectangle's location is actually stored using the left-top coordinates. The choice is, of course, arbitrary, and from the *external* perspective of a user, it is unimportant.

The methods `left`, `right`, `top`, `bottom`, `width` and `height`, and `area` are accessor or *property* methods that allow users to access these logical features of the rectangle. Some (`left` and `top`) are directly represented as attributes in the structure, while others (`right`, `bottom` and `area`), are computed on-the-fly. Again, because the user is expected to use these methods to access the logical attributes of the rectangle through methods, there is no need to access the actual attributes directly. Indeed, we indicate our *expectation of privacy* for the actual attributes by prefixing their names with an underscore (`_`). If you find yourself

[2] Actually, `__init__` works together with a rarely used method `__new__`.

directly accessing attributes with an expression of the form `obj._attr`, *stop*: you will be violating the expectation of privacy.[3]

The `center` method demonstrates a feature of Python not always found in programming languages: the ability to return a tuple. Since the user will likely access both the x- and y-coordinates of the rectangle at the same time, we return both as a tuple that may be used directly as a point, or assigned to a 2-tuple of assignable targets or *l-values*.

Finally, the `__repr__` method generates the preferred printable representation of the object. It is customary to return a string that, if interpreted by a Python interpreter, would regenerate the object in its current state. Indeed, as we shall see in Chapter **??**, these representations are useful in preserving the state of objects in permanent files.

Exercise 1.1 *Nearly everything can be improved. Are there improvements that might be made to the* `Rect` *class?*

We should feel pleased with the progress we have made. We have developed the signatures for the rectangle interface, even though we have no immediate application. We also have some emerging ideas about implementing the `Rect` internally. Though we may decide that we wish to change that internal representation, by using an interface to protect the implementation of the internal state, we have insulated ourselves from changes suggested by inefficiencies we may yet discover.

To understand the merit of this technique of class design, we might draw an analogy between the design of an object and the design of a light bulb for your back porch. The protected attributes and methods of an object are analogous to the *internal* design of the bulb. The *externally observable* features, including the voltage and the size of the socket, are provided without giving any details about the implementation of the object. If electrical socket manufacturers depended on a particular *internal* features of light bulbs—for example the socket only supported bright xenon bulbs—it might ultimately restrict the variety of suppliers of light bulbs in the future. Likewise, manufacturers of light bulbs should enjoy a certain freedom in their implementation: as long as they only draw power in an agreed-upon way and as long as their bulb fits the socket, they should be free to use whatever design they want.[4]

[3] Python, unlike Java and C++, does not actually *enforce* a notion of privacy. As the implementers are fond of pointing out, that privacy is a "gentleman's agreement".

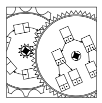
[4] It is interesting to note that many compact fluorescent bulbs actually have (very small) processor chips that carefully control the power used by the light. Newer bulbs use *digital* circuitry that effectively simulates more expensive analog circuitry (resistors, capacitors, and the like) of the past. We realize the implementation doesn't matter, because of the power of abstraction.

1.4 A Special-Purpose Class: A Bank Account

We now look at the detailed construction of an admittedly simplistic class: a `BankAccount`. Many times, it is necessary to provide a tag associated with an instance of a data structure. You might imagine that your bank balance is kept in a database at your bank. When you get money for a trip through the Berkshires, you swipe your card through an automated teller bringing up your account. Your account identifier, presumably, is unique to your account. Nothing about you or your banking history is actually stored in your account identifier. Instead, that number is used to find the record linked to your account: the bank searches for a structure associated with the number you provide. Thus a `BankAccount` is a simple, but important, data structure. It has an `account` (an identifier that never changes) and a `balance` (that potentially *does* change). The public methods of such a structure are as follows:

Automated teller: a robotic palm reader.

```
class BankAccount:
    def __init__(self, acc, bal=0.0):
        """Create a bank account."""

    def account(self):
        """Get account number."""

    def balance(self):
        """Get balance from account."""

    def deposit(self, amount):
        """Make deposit into account."""

    def withdraw(self, amount):
        """Make withdrawal from account."""

    def __eq__(self, that):
        """Compare bank accounts by account number."""
```


`BankAccount`

The substance of these methods has purposefully been removed because, again, it is unimportant for us to know exactly how a `BankAccount` is implemented. We have ways to construct new accounts, as well as ways to access the account name and value, and two methods to update the balance. The special method `__eq__` is provided to allow us to see if two accounts are the same (i.e. they have the same account number).

Let's look at the implementation of these methods, individually. To build a new bank account, you must call the `BankAccount` constructor with one or two parameters (the initial balance is optional, and defaults to 0). The account number provided never changes over the life of the `BankAccount`—if it were necessary to change the value of the account number, a new `BankAccount` would have to be made, and the balance would have to be transferred from one to the other. The constructor plays the important role of performing the one-time initialization of the account name field. Here is the code for a `BankAccount` constructor:

```
def __init__(self, acc, bal=0.0):
    """Create a bank account."""
    self._account = acc
    self._balance = bal
```

Two attributes—`_account` and `_balance`—of the `BankAccount` object are responsible for maintaining the object's state. The `_account` keeps track of the account name, while the `_balance` field maintains the balance.

Since account identifiers are unique to `BankAccounts`, to check to see if two accounts are "the same," we need only compare the `account` attribute. This is the purpose of the `__eq__` method. Here's the code:

```
def __eq__(self, that):
    """Compare bank accounts by account number."""
    return self.account == that.account
```

This method is called whenever two `BankAccounts` are compared with the `==` operator. Notice that the `BankAccount` `__eq__` method calls the `__eq__` method of the account identifier. Every object in Python has an `__eq__` method. If you don't explicitly provide one, the system will write one for you. Generally speaking, one should assume that the automatically written or *default* `equals` method is of little use. This notion of "equality" of objects is often based on the complexities of our abstraction; its design must be considered carefully.

One can ask the `BankAccount` about various aspects of its state by calling its `account` or `balance` methods:

```
@property
def account(self):
    """Get account number."""
    return self._account

@property
def balance(self):
    """Get balance from account."""
    return self._balance
```

In these methods, we annotate each definition with the *decorator* `@property`. Decorators allow us to carefully control how a method is to be used. In this case, the `@property` decorator causes Python to define a new pseudo-attribute or *property* that is accessed like an attribute. We will see, later, that if `js` is a `BankAccount`, we will be able to access the account balance, not with a function call, but as a read-only attribute using `js.balance`. We are not, however, able to *modify* the balance; we cannot use `js.balance` as a target of assignment. In general, when we want to provide read-only access to features of an object–often identified by the use of a noun (e.g. `balance`) as the name of the feature–we decorate the accessor method with the `@property` decoration, allowing us to apparently directly access the property like an object attribute. Property access always returns a value, and rarely has a side-effect. When attributes, properties, or features of an object are *acted on*—typically indicated by the use of a verb (e.g. `deposit`)—we declare the method normally, without the decoration. Actions optionally return values, and frequently have side-effects. Thus properties are accessed as attributes, and actions are performed by methods.

In the end, properties do little more than pass along the information found in the `_account` and `_balance` attributes, respectively. We call such methods *accessors*. In a different implementation of the `BankAccount`, the balance would not have to be explicitly stored—the value might be, for example, the difference between two attributes, `_deposits` and `_drafts`. Given the interface, it is not much of a concern to the user which implementation is used.

We provide two more methods, `deposit` and `withdraw`, that explicitly modify the current balance. These are actions that change the state of the object; they are called *mutator* methods:

```
def deposit(self, amount):
    """Make deposit into account."""
    self._balance += amount

def withdraw(self, amount):
    """Make withdrawal from account."""
    assert self._balance >= amount, "Insufficient funds for withdrawal."
    self._balance -= amount
```

Again, because these actions are represented by verbs, they're not decorated with the `@property` annotation. Because we would like to change the balance of the account, it is important to have a method that allows us to modify it. On the other hand, we purposefully don't have a `setAccount` method because we do not want the account number to be changed without a considerable amount of work (work that, by the way, models reality).

Here is a simple application that determines whether it is better to deposit \$100 in an account that bears 5 percent interest for 10 years, or to deposit \$100 in an account that bears $2\frac{1}{2}$ percent interest for 20 years. It makes use of the `BankAccount` object just outlined:

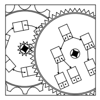
```
jd = BankAccount("Jain Dough")
jd.deposit(100)
js = BankAccount("Jon Smythe",100.00)
for year in range(10):
    jd.deposit(jd.balance * 0.05)
for year in range(20):
    js.deposit(js.balance * 0.025)
print("Jain invests $100 over 10 years at 5%.")
print("After 10 years {0} has ${1}".format(jd.account,jd.balance))
print("Jon invests $100 over 20 years at 2.5%.")
print("After 20 years {0} has ${1}".format(js.account,js.balance))
```

Exercise 1.2 *Which method of investment would you pick?*

1.5 A General-Purpose Class: A Key-Value Pair

At least Dr. Seuss started with 50 words!

The following small application implements a Pig Latin translator based on a dictionary of nine words. The code makes use of a list of `KV`s or key-value pairs, each of which establishes a relation between an English word and its Pig Latin translation. For each string passed as the argument to the `main` method, the list is searched to determine the appropriate translation.[5]



atinlay

```
translation = [ KV("a","aay"),
                KV("bad","adbay"),
                KV("had","adhay"),
                KV("dad","adday"),
                KV("day","ayday"),
                KV("hop","ophay"),
                KV("on","onay"),
                KV("pop","oppay"),
                KV("sad","adsay"),
                ]
story = "hop on pop"
for word in story.split():
    for entry in translation:
        if entry.key == word:
            print(entry.value)
```

When this application is run:

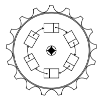
```
python3 atlinLay hop on pop
```

the results are

```
ophay
onay
oppay
```

While this application may seem rather trivial, it is easy to imagine a large-scale application with similar needs.[6]

We now consider the design of the `KV` class. Notice that while the *type* of data maintained is different, the *purpose* of the `KV` is very similar to that of the `BankAccount` class we just discussed. A `KV` is a key-value pair such that the `key` cannot be modified, but the value can be manipulated. Here is the interface for the `KV` class:



KV

[5] Python provides many more efficient techniques for implementing this application. The approach, here, is for demonstration purposes. Unprofessional driver, open course, please attempt.

[6] Pig Latin has played an important role in undermining court-ordered restrictions placed on music piracy. When Napster—the rebel music trading firm—put in checks to recognize copyrighted music by title, traders used Pig Latin translators to foil the recognition software!

```
class KV:
    __slots__ = ["_key", "_value"]

    def __init__(self,key,value=None):

    @property
    def key(self):

    @property
    def value(self):

    @value.setter
    def value(self,value):

    def __str__(self):

    def __repr__(self):

    def copy(self):

    def __eq__(self,other):

    def __hash__(self):
```

Like the `BankAccount` class, the `KV` class has a straightforward initializer that sets the attributes pre-declared in the slot list:

```
__slots__ = ["_key", "_value"]

def __init__(self,key,value=None):
    assert key is not None, "key-value pair key is not None"
    self._key = key
    self._value = value
```

Usually the initializer allows the user to construct a new `KV` by initializing both fields. On occasion, however, we may wish to have a `KV` whose `key` field is set, but whose `value` field is left *referencing nothing*. (An example might be a medical record: initially the medical history is incomplete, perhaps waiting to be forwarded from a previous physician.) For this purpose, we provide a default value for the `value` argument of `None`. `None` is the way we indicate a reference that references nothing.

Now, given a particular `KV`, it is useful to be able to retrieve the key or value. The attribute implementing the key (`_key`) is hidden. Our intent is that users of the `KV` class must depend on methods to read or write the attribute values. In Python, read-only attributes are typically implemented as a property. This approach makes the key portion of the `KV` *appear* to be a standard attribute, when in fact it is a hidden attribute accessed by a similarly named method. We make both `key` and `value` properties:

```
@property
def key(self):
    return self._key

@property
def value(self):
    return self._value
```

The `value` property, however, can be changed as well. Mutable properties are set using a *setter* method, which is indicated by a setter decoration. The setter method simply takes its parameter and assigns it to the underlying `_value` attribute:

```
@value.setter
def value(self,value):
    self._value = value
```

Setter methods are used in circumstances that the associated property is the target of an assignment.

There are other methods that are made available to users of the `KV` class, but we will not discuss the details of that code until later. Some of the methods are required and some are just nice to have around. While the code may look complicated, we take the time to implement it correctly, so that *we will not have to reimplement it in the future*.

Principle 2 *Free the future: reuse code.*

It is difficult to fight the temptation to design data structures from scratch. We shall see, however, that many of the more complex structures would be very difficult to construct if we could not base our implementations on the results of previous work.

1.6 Sketching an Example: A Word List

Suppose we're interested in building a word-guessing game called *Superbrain*. The computer selects random 5-letter words and we try to guess them. Over several rounds, the computer should pick a variety of words and, as each word is used, it should be removed from the word list. Using an object-oriented approach, we'll determine the essential features of a `WordList`, the Python object that maintains our list of words.

Our approach to designing the data structures has the following five informal steps:

1. Identify the types of operations you expect to perform on your object. What operations *access* your object only by reading properties of its state? What operations might modify or *mutate* your objects?

2. Identify, given your operations, those attributes that support the *state* of your object. Information about an object's state is carried by attributes within the object between operations that modify the state. Since there may be many ways to encode the state of your object, your description of the state may be very general.
3. Identify any rules of consistency. In the `Rect` class, for example, it would not be good to have a negative width or height. We will frequently focus on these basic statements about consistency to form pre- and postcondition checks. A word list, for example, should probably not contain numeric digits.
4. Determine the general form of the initializers. Initializers are synthetic: their sole responsibility is to get a new object into a good initial and consistent state. Don't forget to consider the best state for an object constructed using the parameterless constructor.
5. Identify the types and kinds of information that, though not directly visible outside the class, *efficiently* provide the information needed by the methods that transform the object. Important choices about the internals of a data structure are usually made at this time. Sometimes, competing approaches are developed until a comparative evaluation can be made. That is the subject of much of this book.

The operations necessary to support a list of words can be sketched out without much controversy, even if we don't know the intimate details of constructing the word game itself. Once we see how the data structure is used, we have a handle on the design of the interface. We can identify the following general use of the `WordList` object:

```
possibleWords = WordList() # read a word list
possibleWords.length(min=2,max=4)   # keep only 5 letter words

playing = True
while playing:
    word = possibleWords.pick()     # pick a random word
    print("I'm thinking of a {} letter word.".format(len(word)))
    userWord = readGuess(word,possibleWords)
    while userWord != word:
        printDifferences(word,userWord)
        userWord = readGuess(word,possibleWords)
    print('You win!')
    possibleWords.remove(word)      # remove word from play
    playing = playAgainPrompt()
```

WordGame

Let's consider these lines. One of the first lines (labeled `declaration`) declares a *reference* to a `WordList`. For a reference to refer to an object, the object must be constructed. We require, therefore, an initializer for a `WordList` that reads a list of words from a source, perhaps a dictionary. To make the game moderately challenging, we thin the word list to those words that are exactly 5 letters long.

At the beginning of each round of Superbrain, a random word is selected (`selectAny`), setting the `word` reference. To make things interesting, we presume that the `pick` method selects a random word each time. Once the round is finished, we use the `remove` method to remove the word from the word list, eliminating it as a choice in future rounds.

There are insights here. First, we have said very little about the Superbrain game other than its interaction with our rather abstract list of words. The details of the screen's appearance, for example, do not play much of a role in understanding how the `WordList` structure works. We knew that a list was necessary for our program, and we considered the program *from the point of view of the object*. Second, we don't really know how the `WordList` is implemented. The words may be stored in a list, or in a file on disk, or they may use some technology that we don't currently understand. It is only important that we have *faith* that the structure can be implemented. We have sketched out the method headers, or *signatures*, of the `WordList` *interface*, and we have faith that an *implementation* supporting the interface can be built. Finally we note that what we have written is not a complete program. Still, from the viewpoint of the `WordList` structure, there are few details of the interface that are in question. A reasoned individual should be able to look at this design and say "this will work—provided it is implemented correctly." If a reviewer of the code were forced to ask a question about how the structure works, it would lead to a refinement of our understanding of the interface.

We have, then, the following required interface for the `WordList` class:

```
class WordList:
    def __init__(self,file='dolch.dict'):
        """Initialize a word list from a dictionary.
           By default, use word list making up 25% of written English"""

    @property
    def count(self):
        """The number of words in the list."""

    def pick(self):
        """Select a word randomly from word list."""

    def contains(self,word):
        """Returns true exactly when word list contains word."""

    def remove(self, word):
        """Remove a word from the word list, if present."""

    def length(self, min=0, max=50):
        """Thin word list to words with lengths between min and max."""

    def alphabet(self,alpha):
        """Thin word list to words drawn from alphabet alpha."""
```

We will leave the implementation details of this example until later. You might consider various ways that the WordList might be implemented. As long as the methods of the interface can be supported by your data structure, your implementation is valid.

Exercise 1.3 *Finish the sketch of the* WordList *class to include details about its attributes.*

1.7 A Full Example Class: Ratios

Suppose we want to keep track of a ratio or fraction. Typically, we represent these quantities as a fraction or quotient of two values, a numerator and a denominator. How might we implement a ratio as an object in Python?

Obviously, it is important the we be able to construct a ratio from other numeric values, especially a pair of integers. When we generate a string representation, the object should generate a representation that can be parsed as a call to this constructor.

Typically, we represent ratios in *lowest terms*, where the greatest common divisor of the numerator and denominator is 1. To be truly useful, we would also like to be able to support basic arithmetic operations.

Here, for example, is a sketch of a program that would compute the sum of the reciprocals of the first few factorials:

```
from Ratio import *

def fact(n):
    result = 1
    for i in range(1,n+1):
        result *= i
    return result

def computeE():
    e = Ratio(0)
    for i in range(15):
        e += Ratio(1,fact(i))
    print("Euler's constant = {}".format(e.numerator/e.denominator))
    print("Approximate ratio = {}".format(e))

computeE()
```

The result of this computation is an approximation to *Euler's number*, the base of the natural logarithm. By computing it as a ratio, we have the opportunity to develop a ratio of integers that is a good approximation:

```
Euler's constant = 2.71828182846
Approximate ratio = 47395032961/17435658240
```

Given our thinking, the following attributes and constructor lay the foundation for our `Ratio` class:

```
@total_ordering
class Ratio(numbers.Rational):
    __slots__ = [ "_top", "_bottom" ]

    def __init__(self,numerator=1,denominator=1):
        """Construct a ratio of numerator/denominator."""
        self._top = int(numerator)
        self._bottom = int(denominator)
        assert denominator != 0
        self._lowestTerms()
```

When constructing a `Ratio` object, we default to the value `1`, or, if a numerator is provided, a denominator of `1`. If both numerator and denominator are provided, we make sure the `Ratio` is in the best state possible: its denominator should not be zero and any common divisors of the numerator and denominator should be factored out, to keep the `Ratio` in lowest terms.

Two methods support generating this canonical form of a `Ratio`: `_lowestTerms` and `gcd`. The `_lowestTerms` method is a class method that we only want to make available internally; it is, in some sense, private. For this reason, we place an underscore at the beginning of its name, much like the private attributes, `_top` and `_bottom`.

```
def _lowestTerms(self):
    """Reduce the ratio to lowest terms:
       Divide top and bottom by any common divisor."""
    if self._bottom < 0:
        self._top = -self._top
        self._bottom = -self._bottom
    f = gcd(self._top,self._bottom)
    self._top //= f
    self._bottom //= f
```

This method makes use of `gcd`, a general purpose function that is declared outside the class. First, it is not a computation that depends on a `Ratio` (so it should not be a method of the `Ratio` class), and second, it may be useful to others. By making `gcd` a top-level method, everyone can make use of it. The computation is a slightly optimized form of the granddaddy of all computer algorithms, first described by Euclid:

```
def gcd(a,b):
    """A private utility function that determines the greatest common
       divisor of a and b."""
    while b != 0:
        (a,b) = (b,a%b)
    return a if a >= 0 else -a
```

The Ratio object is based on Python's notion of a rational value (thus the mention of Rational in the class definition). This is our first encounter with *inheritance*. For the moment, we'll just say that thinking of this class as a rational value obliges us to provide certain basic operations, which we'll get to shortly.

The primary purpose of the class, of course, is to hold a numeric value. The various components of a Ratio that are related it its value we make accessible through basic properties, numerator and denominator:

```
@property
def numerator(self):
    return self._top

@property
def denominator(self):
    return self._bottom
```

Because these are declared as properties, we may read these values as though numerator and denominator were fields of the class. We are not able to modify the values. We also provide two methods—int and float—that provide floating point and integer equivalents of the Ratio. We may find these useful when we're mixing calculations with Python's other built-in types.

```
def int(self):
    """Compute the integer portion of the Ratio."""
    return int(float(self))

def float(self):
    """Compute the floating point equivalent of this Ratio."""
    return n/d
```

Now, because Python has a rich collection of numeric classes, and because many of the operations that act on those values are associated with mathematical operations, it is useful to have the Ratio class provide the special methods that support these basic operations. First, some basic operations support several forms of comparison, while others extend the meaning of existing mathematical functions, like abs that computes the absolute value of a numeric. These methods are as follows:

```
def __lt__(self,other):
    """Compute self < other."""
    return self._top*other._bottom < self._bottom*other._top

def __le__(self,other):
    """Compute self < other."""
    return self._top*other._bottom <= self._bottom*other._top

def __eq__(self,other):
    """Compute self == other."""
```

```
        return (self._top == other._top) and \
               (self._bottom == other._bottom)

    def __abs__(self):
        """Compute the absolute value of Ratio."""
        return Ratio(abs(self._top),self._bottom)

    def __neg__(self):
        """Compute the negation of Ratio."""
        return Ratio(-self._top,self._bottom)
```

Notice that many of these methods all have a notion of `self` and `other`. The `self` object appears on the left side of the binary operator, while `other` appears on the right.

Basic mathematical operations on `Ratio` are similarly supported. Most of these operations return a new `Ratio` value holding the result. Because of this, the `Ratio` class consists of instances that are read-only.

```
    def __add__(self,other):
        """Compute new Ratio, self+other."""
        return Ratio(self._top*other._bottom+
                     self._bottom*other._top,
                     self._bottom*other._bottom)

    def __sub__(self,other):
        """Compute self - other."""
        return Ratio(self._top*other._bottom-
                     self._bottom*other._top,
                     self._bottom*other._bottom)

    def __mul__(self,other):
        """Compute self * other."""
        return Ratio(self._top*other._top,
                     self._bottom*other._bottom)

    def __floordiv__(self,other):
        """Compute the truncated form of division."""
        return math.floor(self/other)

    def __mod__(self,other):
        """Computer self % other."""
        return self-other*(self//other)

    def __truediv__(self,other):
        return Ratio(self._top*other._bottom,self._bottom*other._top)
```

Finally, we provide two operations that construct string representations of `Ratio` values—`__str__` and `__repr__`. The `__str__` method is called when printing human-readable values out. The `__repr__` method has a subtle difference: it generates a string that could be understood or *parsed* as an expression that would generate the value. (We've seen this before, in our discussion of the `KV` class.)

```
def __repr__(self):
    if self._bottom == 1:
        return "Ratio({0})".format(self._top)
    else:
        return "Ratio({0},{1})".format(self._top,self._bottom)

def __str__(self):
    if self._bottom == 1:
        return "{0}".format(self._top)
    else:
        return "{0}/{1}".format(self._top,self._bottom)
```

The integers in the `format` string give the relative positions of the parameters consumed, starting with the first, parameter 0. Notice that we are using the `format` method for strings but we're not printing the value out; the result can be used of a variety of purposes. That decision—whether to print it out or not—is the decision of the caller of these two functions.

You can see the difference by comparing the results of the following interactive Python expressions:

```
>>> a = Ratio(4,6)
>>> print(str(a))
2/3
>>> print(repr(a))
Ratio(2,3)
```

It is a good time to note that the `format` statement can, with these routines, print both forms of ratio:

```
>>> "{0!r} = {0}".format(a)
Ratio(2,3) = 2/3
```

The string returned by the `__str__` method is meant to be understood by humans. The `__repr__` method, on the other hand, can be interpreted in a way that constructs the current state of the `Ratio`. This will be pivotal when we discuss how to checkpoint and reconstruct states of objects.

1.8 Interfaces

Sometimes it is useful to describe the interface for a number of different classes, without committing to an implementation. For example, in later sections of this text we will implement a number of data structures that are mutable—that is, they are able to be modified by adding or removing values. We can, for all of these classes, specify a few of their fundamental methods by using the Python *abstract base class* mechanism, which is made available as part of Python 3's `abc` package. Here, for example, is an abstract base class, `Linear`, that requires a class to implement both an `add` and `remove` method:

```
import abc
from collections import Iterable, Iterator, Sized, Hashable
from structure.view import View
from structure.decorator import checkdoc, mutatormethod, hashmethod

class Linear(Iterable, Sized, Freezable):
    """
    An abstract base class for classes that primarily support organization
    through add and remove operations.
    """

    @abc.abstractmethod
    def add(self, value):
        """Insert value into data structure."""
        ...

    @abc.abstractmethod
    def remove(self):
        """Remove next value from data structure."""
        ...
```

Notice that the method bodies have been replaced by ellipses. Essentially, this means *the implementation has yet to be specified*. Specifying just the *signatures* of methods in a class is similar to writing boilerplate for a contract. When we are interested in writing a new class that meets the `Linear` interface, we can choose to have it *implement* the `Linear` interface. For example, our `WordList` structure of Section 1.6 might have made use of our `Linear` interface by beginning its declaration as follows:

```
class WordList(Linear):
```

`WordList`

When the `WordList` class is compiled by the Python compiler, it checks to see that each of the methods mentioned in the `Linear` interface—`add` and `remove`—is actually implemented. If any of these methods that have been marked with the `@abstractmethod` decorator do not ultimately get implemented in `WordList`, the program will compile with errors. In this case, only `remove` is part of the `WordList` specification, so we must either (1) not have `WordList` implement the `Linear` interface or (2) augment the `WordList` class with an `add` method.

Currently, our `WordList` is close to, but not quite, a `Linear`. Applications that demand the functionality of a `Linear` will not be satisfied with a `WordList`. Having the class implement an interface increases the flexibility of its use. Still, it may require considerable work for us to upgrade the `WordList` class to the level of a `Linear`. It may even work against the design of the `WordList` to provide the missing methods. The choices we make are part of an ongoing design process that attempts to provide the best implementations of structures to meet the demands of the user.

1.9 Who Is the User?

When implementing data structures using classes and interfaces, it is sometimes hard to understand *why* we might be interested in hiding the implementation. After all, perhaps we know that ultimately *we* will be the only programmers making use of these structures. That might be a good point, except that if you are really a successful programmer, you will implement the data structure flawlessly this week, use it next week, and not return to look at the code for a long time. When you *do* return, your view is effectively that of a user of the code, with little or no memory of the implementation.

One side effect of this relationship is that we have all been reminded of the need to write comments. If you do not write comments, you will not be able to read the code. If, however, you design, document, and implement your interface carefully, you might not ever have to look at the implementation! That's good news because, for most of us, in a couple of months our code is as foreign to us as if someone else had implemented it. The end result: consider yourself a user and design and abide by your interface wherever possible. If you know of some public field that gives a hint of the implementation, do not make use of it. Instead, access the data through appropriate methods. You will be happy you did later, when you optimize your implementation.

Principle 3 *Design and abide by interfaces as though you were the user.*

A quick corollary to this statement is the following:

Principle 4 *Use hidden attributes to hold state data.*

If the data are protected in this manner, you cannot easily access them from outside the class, and you are forced to abide by the restricted access of the interface.

1.10 Discussion

The construction of substantial applications involves the development of complex and interacting structures. In object-oriented languages, we think of these structures as objects that communicate through the passing of messages or, more formally, the invocation of methods.

We use object orientation in Python to write the structures found in this book. It is possible, of course, to design data structures without object orientation, but any effective data structuring model ultimately depends on the use of some form of abstraction that allows the programmer to avoid considering the complexities of particular implementations.

In many languages, including Python, data abstraction is supported by separating the interface from the implementation of the data structure. To ensure that users cannot get past the interface to manipulate the structure in an uncontrolled fashion, the system controls access to fields, methods, and classes. The implementor plays an important role in making sure that the structure is usable, given the interface. This role is so important that we think of implementation as supporting the abstract—sometimes usefully considered a *contract* between the implementor and the user. This analogy is useful because, as in the real world, if contracts are violated, someone gets upset!

Initial design of the interfaces for data structures arises from considering how they are used in simple applications. Those method calls that are required by the application determine the interface for the new structure and constrain, in various ways, the choices we make in implementing the object.

Self Check Problems

Solutions to these problems begin on page **??**.

1.1 What is meant by abstraction?

Problems

Selected solutions to problems begin on page **??**.

1.1 Which of the following are primitive Java types: `int`, `Integer`, `double`, `Double`, `String`, `char`, `Association`, `BankAccount`, `boolean`, `Boolean`?

1.2 Which of the following variables are associated with valid constructor calls?

```
BankAccount a,b,c,d,e,f; Association g,h; a = new
BankAccount("Bob",300.0); b = new BankAccount(300.0,"Bob"); c = new
BankAccount(033414,300.0); d = new BankAccount("Bob",300); e = new
BankAccount("Bob",new Double(300)); f = new
BankAccount("Bob",(double)300); g = new Association("Alice",300.0); h =
new Association("Alice",new Double(300));
```

Chapter 2

Comments, Conditions, and Assertions

CONSIDER THIS: WE CALL OUR PROGRAMS "CODE"! Computer languages, including Python, are designed to help express algorithms in a manner that a machine can understand. Making a program run more efficiently often makes it less understandable. If language design was driven by the need to make the program readable by programmers, it would be hard to argue against programming in English.

Okay, perhaps French!

A *comment* is a carefully crafted piece of text that describes the state of the machine, the use of a variable, or the purpose of a control construct. Many of us, though, write comments for the same reason that we exercise: we feel guilty. You feel that, if you do not write comments in your code, you "just *know*" something bad is going to happen. Well, you are right. A comment you write today will help you out of a hole you dig tomorrow.

Ruth Krauss: "A hole is to dig."

Commenting can be especially problematic in Python, a scripting language that is often used to hastily construct solutions to problems. Unfortunately, once a program is in place, even if it was hastily constructed, it gains a little momentum: it is much more likely to stick around. And when uncommented programs survive, they're likely to be confusing to someone in the future. Or, comments are hastily written after the fact, to help understand the code. In either case, the time spent thinking seriously about the code has long since passed and the comment might not be right. If you write comments beforehand, while you are designing your code, it is more likely your comments will describe what you want to do as you carefully think it out. Then, when something goes wrong, the comment is there to help you find the errors in the code. In fairness, the code and the comment have a symbiotic relationship. Writing one or the other does not really feel complete, but writing both supports you with the redundancy of concept.

The one disadvantage of comments is that, unlike code, they cannot be checked. Occasionally, programmers come across comments such as "If you think you understand this, you don't!" or "Are you reading this?" One could, of course, annotate programs with mathematical formulas. As the program is compiled, the mathematical comments are distilled into very concise descriptions of what should be going on. When the output from the program's code does not match the result of the formula, something is clearly wrong with your logic. But *which* logic? The writing of mathematical comments is a level of detail most programmers would prefer to avoid.

Semiformal convention: a meeting of tie haters.

A compromise is a semiformal convention for comments that provide a reasonable documentation of *when* and *what* a program does. In the code associated with this book, we see one or two comments for each method or function that describe its purpose. These important comments are the *precondition* and *postcondition*.

2.1 Pre- and Postconditions

The *precondition* describes, as succinctly as possible in your native tongue, the conditions under which a method may be called and expected to produce correct results. Ideally the precondition expresses the *state* of the program. This state is usually cast in terms of the parameters passed to the routine. For example, the precondition on a square root function might be The authors of this square root function expect that the parameter is not a negative number. It is, of course, legal in Python to call a function or method if the precondition is not met, but it might not produce the desired result. When there is no precondition on a procedure, it may be called without failure.

The *postcondition* describes the state of the program once the routine has been completed, *provided the precondition was met*. Every routine should have some postcondition. If there were not a postcondition, then the routine would not change the state of the program, and the routine would have no effect! Always provide postconditions.

Pre- and postconditions do not force you to write code correctly. Nor do they help you find the problems that *do* occur. They can, however, provide you with a uniform method for documenting the programs you write, and they require more thought than the average comment. More thought put into programs lowers your average blood pressure and ultimately saves you time you might spend more usefully playing outside, visiting museums, or otherwise bettering your mind.

2.2 Assertions

In days gone by, homeowners would sew firecrackers in their curtains. If the house were to catch fire, the curtains would burn, setting off the firecrackers. It was an elementary but effective fire alarm.

And the batteries never needed replacing.

An *assertion* is an assumption you make about the state of your program. In Python, we will encode our assumptions about the state of the program using an assertion statement. The statement does nothing if the assertion is true, but it halts your program with an error message if it is false. It is a firecracker to sew in your program. If you sew enough assertions into your code, you will get an early warning if you are about to be burned by your logic.

Principle 5 *Test assertions in your code.*

Here's an example of a check to make sure that the precondition for the `root` function was met: Should we call `root` with a negative value, the assertion fails, the message describing the failure is printed out, and the program comes to a halt. Here's what appears at runtime: The first two lines of the traceback indicate that we were trying to print the square root of -4, on line 22 in `root.py`. The `root` function's assertion, on line 9 of `root.py` failed *because* the value was negative. This resulted in a failed assertion (thus the `AssertionError`). The problem is (probably) on line 22 in `root.py`. Debugging our code should probably start at that location.

When you form the messages to be presented in the `AssertionError`, remember that something went wrong. The message should describe, as best as possible, the context of the failed assumption. Thus, while only a string is required for an assertion message, it is useful to generate a formatted string describing the problem. The formatting is not called if the assertion test succeeds.

A feature of Python's native assertion testing is that the tests can be automatically removed at compile time when one feels secure about the way the code works. Once fixed, we might run the above code with the command:

```
python3 -O root.py
```

The `-O` switch indicates that you wish to *optimize* the code. One aspect of the optimization is to remove the assertion-testing code. This may significantly improve performance of functions that are frequently called, or whose assertions involve complex tests.

2.3 Documentation-based Testing

When we want to verify post-conditions we are essentially verifying the correctness of the function. One technique is to perform assertion testing just before the exit of the code. Python provides support for testing postconditions using a system called *documentation testing*.

When you write documentation for functions that you hope others will use, it is useful to provide a *small* number of examples and the expected results. For example, in our square root code, we might add the following lines to the docstring: Notice that these examples demonstrate how the function can be called and thus serve as part of the basic documentation. We can, in the file `root.py` execute the statements: When a file is executed as a script, the module name (`__name__`) is `__main__`. Thus, executing the script `root.py` will cause the tests (prefixed by triple-less signs) appearing in the documentation to be interpreted. The return value is matched against the text that follows in the docstring.

When a test fails, the `doctest` module prints the broken test, the expected output, and the actual output that was generated:

2.4 Craftsmanship

If you *really* desire to program well, a first step is to take pride in your work—pride enough to sign your name on everything you do. Through the centuries, fine furniture makers signed their work, painters finished their efforts by dabbing on their names, and authors inscribed their books. Programmers should stand behind their creations.

Computer software has the luxury of immediate copyright protection—it is a protection against piracy, and a modern statement that you stand behind the belief that what you do is worth fighting for. If you have crafted something as best you can, add a comment at the top of your code: If, of course, you *have* stolen work from another, avoid the comment and carefully consider the appropriate attribution.

2.5 Discussion

Effective programmers consider their work a craft. Their constructions are well considered and documented. Comments are not necessary, but documentation makes working with a program much easier. One of the most important comments you can provide is your name—it suggests you are taking credit *and* responsibility for things you create. It makes our programming world less anonymous and more humane.

Special comments, including conditions and assertions, help the user and implementor of a method determine whether the method is used correctly. While it is difficult for compilers to determine the "spirit of the routine," the implementor is usually able to provide succinct checks of the sanity of the function. Five minutes of appropriate condition description and checking provided by the implementor can prevent hours of debugging by the user.

I've done my time!

Self Check Problems

Solutions to these problems begin on page **??**.

2.1 Why is it necessary to provide pre- and postconditions?

2.2 What can be assumed if a method has no precondition?

2.3 Why is it not possible to have a method with no postcondition?

2.4 Object orientation allows us to hide unimportant details from the user. Why, then, must we put pre- and postconditions on hidden code?

Problems

Selected solutions to problems begin on page **??**.

2.1 What are the pre- and postconditions for the `math.sqrt` method?

2.2 What are the pre- and postconditions for the `ord` method?

2.3 What are the pre- and postconditions for the `chr` method?

2.4 What are the pre- and postconditions for `str.join`?

2.5 Improve the comments on an old program.

2.6 What are the pre- and postconditions for the `math.floor` method?

2.7 What are the pre- and postconditions for `math.asin` class?

Chapter 3

Contract-Based Design

One way we might measure the richness of a programming language is to examine the diversity of the data abstractions that are available to a programmer. Over time, of course, as people contribute to the effort to support an environment for solving a wide variety of problems, new abstractions are developed and used. In many object-oriented languages, these data types are arranged into *type hierarchies* that can be likened to a taxonomy of ways we arrange data to effectively solve problems. The conceptually simplest type is stored at the *top* or the *base* of the hierarchy. In Python, this type is called `object`. The remaining classes are all seen as either direct extensions of `object`, or other classes that, themselves, are ultimately extensions of `object`. When one class, say `int`, extends another, say `object`, we refer to `object` as the *superclass* or *base class*, and we call `int` a *subclass* or *extension*.

Languages, like Python, that support this organization of types benefit in two important ways. First, type extension supports the notion of the specialization of purpose. Typically, through the fleshing out of methods using *method overriding*, we can commit to aspects of the implementation, but only as necessary. There are choices, of course, in the approach, so each extension of a type leads to a new type motivated by a specific purpose. In this way, we observe specialization. Perhaps the most important aspect of this approach to writing data structures is the adoption, or *inheritance*, of code we have already committed to in the base class. Inheritance is motivated by the need to *reuse code*. Secondly, because more general superclasses—like `automobile`—can (often) act as stand-ins for more specific classes—like `beetle`—the users of subclass `beetle` can (often) manipulate `beetle` objects in the more general terms of its superclass, `automobile`. That is because many methods (like `inspect`, `wash`, and `race`) are features that are common to all specializations of `automobile`. This powerful notion—that a single method can be designed for use with a variety of types—is called *polymorphism*. The use of inheritance allows us to use a general base type to manipulate instances of any concrete subclass.

One of the motivations of this book is to build out the object hierarchy in an effort to develop a collection of general purpose data structures in a careful and principled fashion. This exercise, we hope, serves as a demonstration of the benefits of thoughtful engineering. Helpful to our approach is the use of the *abstract base class* system. This is an extension to the standard Python type system that, nonetheless, is used extensively in Python's built-in `collections` package. It supports a modern approach to the incremental development of classes and allows the programmer to specify the user's view of a class, even if the specifics of implementations have not yet been fleshed out. In other languages, like Java and C++, the specification of an incomplete type is called an *interface*. Python does not formally have any notion of interface design, but as we shall see, it allows for the notion of partially implemented or *abstract* classes that can form the basis of fully implemented, concrete type extension. This chapter is dedicated to developing these formal notions into a organized approach to data structure design.

3.1 Using Interfaces as Contracts

As we begin our investigation of new classes of data structures, with few conceptual tools in our belt, our discussion will be at a fairly abstract level. We will focus less how the data is represented and think more about the *logical consistency* of a structure. How will a class's methods maintain this consistency? For example, will adding objects to a structure necessarily mean that they can be removed later? When objects are removed will we be able to specify the target object, or will the choice be made for us? Will the objects be extracted in a way that leaves only larger values behind? This notion of consistency is a subtle, but it frees the software engineer from making *programming decisions* and emphasizes making appropriate *design decisions*. Typically, a thoughtful discussion of new type in such abstract terms leads to the development of an interface or a collection of unimplemented methods we can depend on. In Python, these methods form an *abstract class* that that will be completed when the class is extended to a concrete implementation.

An abstract base class is, essentially, a *contract* that identifies methods that concrete subclasses must provide as part of its *abstract programming interface*, or API. Here, for example, is an abstract base class for what it might mean to be a *list*, in Python:

```
@checkdoc
class List(Iterable, Sized):
    """
    An abstract base class for all forms of list.
    """

    @abc.abstractmethod
    def append(self, value):
        """Append object to end of list."""
        ...

    @abc.abstractmethod
    def count(self, value):
        """Return the number of occurences of value."""
        ...

    @abc.abstractmethod
    def extend(self, iterable):
        """Extend list with values from iterable."""
        ...

    @abc.abstractmethod
    def index(self, value, start=0, stop=-1):
        """Return index of value in list, or -1 if missing."""
        ...

    @abc.abstractmethod
    def insert(self, index, value):
        """Insert object before index."""
        ...

    @abc.abstractmethod
    def pop(self, index=-1):
        """Remove and return item at index (default last).

        Raises IndexError if list is empty or index is out of range."""
        ...

    @abc.abstractmethod
    def remove(self, value):
        """Remove first occurrence of value.

        Raises ValueError if the value is not present."""
        ...

    @abc.abstractmethod
    def reverse(self):
        """Reverses list, in place."""
        ...

    @abc.abstractmethod
    def sort(self,key=None,reverse=False):
        """Performs stable sort of list, in place."""
        ...

    def __str__(self):
        """Return string representation of this list."""
        return str(list(self))

    def __repr__(self):
        """Return string representation of this list."""
        return "{}({})".format(type(self).__name__,str(list(self)))
```

This interface, `List`, outlines what the user can expect from the various implementations of list-like objects. First, there are several methods (`append`, `count`, `extend`, etc.) that are available for general use. Their docstrings help the user understand the designer's intent, *but there is no implementation*. Instead, the specifics of the implementation—which are generally unimportant to the user—are left to implementors of `List`-like classes. Notice that we were able to construct this definition even though we don't know any details of the actual implementation of the built-in `list` class.

These code-less *abstract methods* are decorated with the `@abc.abstractmethod` decoration[1]. To indicate that code is missing, we use the ellipses (`...`) or the Python synonym, `pass`. The ellipses indicate the designer's *promise*, as part of the larger *contract*, to provide code as more *concrete* extension classes are developed. Classes that extend an abstract base class *may* choose to provide the missing code, or not. If a concrete implementation of the abstract method is not provided the subclass remains "abstract." When a class provides concrete implementations of all of the abstract methods that have been promised by its superclasses, the class designer has made all of the important decisions, and the implementation becomes *concrete*. The purpose of this book, then, is to guide the reader through the process of developing an abstract base class, which we use as the description of an interface, and then committing (in one or more steps in the type hierarchy) to one or more concrete implementations.

Because the specification of an abstract base class is incomplete, it is impossible to directly construct instances of the class. It *is* possible, however that some concrete extension of the abstract class will provide all the implementation details that are missing here. Any instance of the *concrete* subclass are, however, also considered instances of this base class. It is often convenient to cast algorithms in terms of operations on instances of an abstract base class (e.g. to `inspect` an `automobile`), even though the actual objects are concrete (e.g. `beetle` instances), and have committed to particular implementation decisions (e.g. air-cooled engines). If the performance of a concrete class is less than ideal, it is then a relatively minor task to switch to another implementation.

The portion of the type hierarchy developed in this text (the "`structure` package") is depicted in Appendix **??**. There, we see that abstract base classes appear high in the hierarchy, and that concrete implementations are direct or indirect extensions of these core interfaces.

In some cases we would like to *register* an existing concrete class as an implementation of a particular interface, an extension of an abstract base class of our own design. For example, the `List` interface, above, is consistent with the built-in `list` class, but that relationship is not demonstrated by the type hierarchy. After all, the `List` interface was developed *after* the internal `list` class was developed. In these situations we make use of the special `register` class method available to abstract base classes:

```
List.register(list)
```

[1] In Python, decorations are identifiers preceded by the `@` symbol. These are functions that work at the *meta level*, on the code itself, as it is defined. You can read more about decorations in Section **??**.

Before we leave this topic we point out that the `List` interface is an extension of two other interfaces that are native to Python's `collections` package: `Iterable` and `Sized`. These are abstract base classes that indicate promises to provide even more methods. In the case of the `Iterable` interface, a important method supporting iteration, called `__iter__`, and in the case of the `Sized` interface, the method `__len__` which indicates the structure has a notion of length or size. Collectively, these "mix-in" interfaces put significant burdens on anyone that might be interested in writing a new `List` implementation. If, however, the contract can be met (i.e. the abstract methods are ultimately implemented), we have an alternative to the `list` class. We shall see, soon, that alternatives to the `list` class can provide flexibility when we seek to improve the performance of a system.

How, then, can we write application code that avoids a commitment to a particular implementation? We consider that question next, we we discuss *polymorphism*.

3.2 Polymorphism

While variables in Python are not declared to have particular types, every object is the result of instantiating a particular class and thus has a very specific origin in the type hierarchy. One can ask a value what its type is with the `type` function:

```
>>> type(3)
type('int')
>>> i = 3
>>> type(i)
type('int')
>>> i = 'hello, world'
>>> type(i)
type('str')
```

A consistent development strategy—and our purpose, here, is to foster such techniques—means that the position of new classes will be used to reflect the (sometimes subtle) relationship between types. Those relations are the result of common ancestors in the type hierarchy. For example, the boolean class, `bool`, is a subclass of the integer type, `int`. This allows a boolean value to be used in any situation where an `int` is required. A little experimentation will verify that the boolean constant `True` is, essentially, the integer 1, while `False` is a stand-in for zero. We can see this by using `bool` values in arithmetic expressions:

```
>>> 1+True
2
>>> 3*False
0
>>> False*True
0
```

When a method (like addition) works in consistent ways on multiple types (here, `int` and `bool` values), we call this *polymorphism*.

Because Python allows us to write programs without respect to specific types, it may appear that *all* methods are polymorphic. In reality, since Python does not automatically check for appropriate types, we must sometimes perform those checks ourselves. Three methods—`type`, `isinstance`, and `issubclass`—help us to perform the type checking by hand.

`type(obj)`. This method, as we have just seen, returns the type of the object `obj`.

`isinstance(obj,cls)`. This boolean method returns `True` precisely when `obj` is an instance of the class `cls`.

`issubclass(cls,supercls)`. This boolean method checks to see if either class `cls` *is* `supercls` or if it is an extension of `supercls`. Notice that writing `isinstance(obj,cls)` is the same as `issubclass(type(obj),cls)`.

Although Python appears lax about declaring and checking the types of data, its type hierarchy provides an organizational mechanism for identifying the most general types of data we wish to support in our methods and data structures.

Python's classes are, themselves, *first class objects*. Each class is an instance of the `type` class (including `type`, itself!) and, as a result, classes have a number of methods that allow us to reason about them.[2] This is one of the most subtle and powerful features of the language. It means, for example, that we can write Python programs that are `reflective`—that is they can inspect and reason about themselves and other Python programs. The discussion of this topic is well beyond the scope of this book, but for the advanced programmer, reflection is well worth investigating.

3.3 Incrementalism: Good or Bad?

In these days of software with double-digit version numbers, it makes one wonder *Can we not get software right?* Initially, one might get the feeling that registering new interface types (like `List`) high in the built-in type hierarchy is simply enabling poor, incremental software design. Isn't the process of registering new interfaces simply a "patch" mechanism that would make up for poor design?

The process of designing good software is complex and involved process. Frequently, it is difficult to appreciate abstraction of a problem when one first approaches a solution. This can be because of a lack of experience in solving the problem. If you are new to a problem domain, it is difficult, sometimes, to develop the correct abstractions until experience is gained in solving the problems. Engineers, of all types, frequently build mock-ups and prototypes before a

[2] Actually, we know that classes are *callable* types because we actually call the class (viz. `list()`) as a factory method to create instances of the class.

final engineering approach is developed. Because software engineers can often make significant use of materials that were developed in a prototyping stage, it is useful to facilitate back-fitting or *refactoring* of an approach. Frequently, in this book we will return to early structures and refine their design after experience has been gained with the development of a wider collection of classes. This is a natural aspect of the process of programming. We should not fight it, we should understand it as evidence of notable progress in learning.[3]

It is also possible that success in the small project leads to the development of larger tools to solve more general problems. In some ways, it is possible to see the development of (ultimately) successful programming languages, like Python, as being an incremental commitment to a unified approach to solving problems. It would be nice to think that the package of structures we develop here is a consistent and useful extension of the approach initially investigated in early Python development. In such a situation, it seems natural to establish and refine a wider collection of interfaces that support the development of data structures that work particularly well with the programming language philosophies *du jure*.

3.4 Discussion

A great deal of the optimism about Python as a programming utility is its uniform commitment to a type hierarchy. This hierarchy reflects the relationships between not only concrete types, but also the partially implemented abstract types, or interfaces, that establish expectations about the consistency-preserving operations in concrete classes. While Python provides many classes, it is impossible (and unnecessary) to predict every need, and so Python has many features that allow us to extend the type hierarchy in ways the reflect common engineering principles. These include:

- Support for the design of interfaces.
- The establishment of types as first-class objects.
- The use of polymorphism to allow general operations to be performed on disparate types.
- The provision of methods for performing detailed type-checking.

The remainder of this text is a discussion about one approach to extending Python's type hierarchy to include classic data structures in a principled way, consistent with modern philosophies about type engineering.

[3] Williams students should note that, at graduation, the Dean need only recommend students for a degree based on "notable progress in learning".

Self Check Problems

Solutions to these problems begin on page **??**.

3.1 Define the following terms: (1) object, (2) class, (3) supertype, (4) subtype, (5) baseclass, (6) abstract method, (7) inheritance, (8) polymorphism, (9) type hierarchy, and (10) `object`.

3.2 What is the type of (a) 42, (b) 42., (c) 42j, (d) range(42), (e) [i for i in range(42)].

3.3 Which of the following are `True`?

1. `isinstance(True,bool)`
2. `isinstance(True,int)`
3. `isinstance(type(True),bool)`
4. `isinstance(type(z),type)` (assume `z` is defined)
5. `issubclass(bool,bool)`
6. `issubclass(bool,int)`
7. `issubclass(type(0),type(False))`
8. `issubclass(type(z),object)` (assume `z` is any value)

Problems

Selected solutions to problems begin on page **??**.

3.1 For a class to be declared abstract, they must be prefixed with the `@abc.abstractmethod` decoration. Explain why it is not sufficient to simply provide elipses (`...`) for the method body. **3.2** The abstract base class mechanism is an add-on to the Python 3 language. The notion of an abstract method is not directly supported by the language itself. Observing the output of `pydoc3 collections.Iterable`, indicate how an abstract class might be identified.

3.3 Investigate each of the following interfaces and identify the corresponding methods required by the abstract class: (1) `collections.abc.Iterable`, (2) `collections.abc.Sized`, (3) `collections.abc.Container`, (4) `collections.abc.Hashable`, and (4) `collections.abc.Callable`.

Chapter 4

Lists

In Python, a *sequence* is any iterable object that can be efficiently indexed by a range of integers. One of the most important built-in sequences is the `list`, first discussed in Section 0.4. In this chapter we will design a family of list-like sequences, each with different performance characteristics.

Before we develop our own structures, we spend a little time thinking about how Python's native `list` structure is implemented. This will allow us to compare alternative approaches and highlight their differences.

4.1 Python's Built-in `list` Class

Python's `list` class is a simple yet versatile structure for storing a collection of items that can be indexed by their position. Lists are container classes that hold references to other objects. They may be empty, and they may change their size over time. Each item (a reference, or the value `None`, indicating "not a reference") in a list has an index that is the number of entries that appear before it in the collection. We call this type of indexing *0-origin* because the first element has index 0. Lists are *mutable* in that values can be added, replaced, or removed. In other languages we might think of a `list` as an *extensible array* or *vector*. In Python, the `list` class is just one example of a *sequence*.

4.1.1 Storage and Allocation

Internally, `lists` are implemented by a reference to an fixed length chunk of memory—we will think of it as an array—that has enough object reference slots to hold the data of the `list`. Because the underlying array holds at least the number of elements that are in the list (perhaps many more were preallocated), the `list` also keeps track of an *allocation size*. Clearly this allocation size is always at least as large as the length of the list. If the `list` should grow and require more references than can be stored in the current chunk of memory, a larger array must be allocated and then the data is transferred from the old store to the new. The old array, no longer useful, can be reclaimed as part of the *garbage collection* process.

The process of copying the entries of a `list` during extension is time-consuming and grows linearly with the number of entries to be preserved. For this reason, Python allocates more space than is needed so that several small increases in the list length can be readily accommodated before the underlying array needs to be reallocated again. One approach that is commonly used when reallocating an extensible array is to initially allocate a small array (in analyses we will assume it has one element, but it is typically larger) and to double the length of the array as necessary.

Each time a `list` is extended, we must transfer each of the values that are already stored within the vector. Because the extension happens less and less frequently, the total number of reference copies required to incrementally grow a list to size n is as large as

$$1 + 2 + 4 + ... + 2^{\log n} = 2n - 1$$

or approximately twice the number of elements currently in the array. Thus the *average number of times each value is copied* as n elements are added to a `list` is approximately 2.

Storage Overhead

We can make a few more observations. If we construct a list by adding a single element at a time it eventually fills the current allocation—it is making 100% use of the available space—and so it must reallocate, doubling the size—making its use 50% of the available space. It consumes this space in a linear manner until the next extension. Averaging over all sizes we see that the list takes up 75% of the space allocated. Another way to think about this is that, on average, a list has a 33% storage overhead—a possibly expensive price to pay, but we are motivated by a stronger desire to reduce the cost of maintaining the integrity of the data through the extension process.

The designers of Python felt 33% overhead was a bit too much to pay, so Python is a little less aggressive. Each time a vector of size n is extended, it allocates $\frac{n}{8} + c$ *extra* locations. The value of c is small, and, for analysis purposes, we can think of the reallocation as growing to about $\frac{9}{8}n$. As the size of the list grows large, the average number of copies required per element is approximately 9. Since, in the limit, the array is extended by about 12.5%, the average unused allocated space is about 6.25%, or an average list overhead of about 6.7%.

When a `list` shrinks one can always live within the current allocation, so copying is not necessary. When the list becomes very small compared to its current allocation, it may be expensive to keep the entire allocation available. When doubling is used to extend lists, it may be useful to shrink the allocation when the list size drops below one third of the allocated space. The new allocation should contain extra cells, but not more than twice the current list size. Shrinking at one third the allocation may seem odd, but shrinking the allocation when the list length is only half the current allocation does not actually result in any freed memory.

Other approaches are, of course, possible, and the decision may be aided by observing the actual growth and shrinkage of `lists` in real programs. Python reallocates a smaller list (to $\frac{9}{8}n + c$) when the size, n, dips below 50% of allocation. This is a fairly conservative approach to freeing memory, probably reflecting the reality that most `lists` frequently grow and rarely shrink.

4.1.2 Basic Operations

As we saw in Chapter 0, the `list` abstraction supports several basic operations. Because Python's `list` implementation is, essentially, a wrapper for an array, the costs associated with many of its operations are the same as those for arrays in other languages. Thinking of `lists` as array helps us evaluate the inherent overhead associated with each of the `list` operations.

The cost of locating and updating existing entries by their index in the list takes constant ($O(1)$) time. If we access the value based on its index, the target of the operation is at a fixed and easily determined offset from the beginning of the underlying array. Since indexing can be accomplished without changing the length of the list, there is no other overhead associated with this list operation.

The elements of a `list` are always stored contiguously.

Exercise 4.1 *Explain why it is important that a list's elements are stored contiguously.*

Since it is important to maintain that property, the costs of inserting or removing elements from the `list` depend heavily on how many *other elements* must be moved. For example, if we wish to insert an element at the "end" of the list (to be found at an index that is equal to the current length of the `list`), the process is simply inserting the new reference just beyond the end of the list. The cost of doing this is constant time; it does not depend on the size of the list (we ignore the overhead of possible list resizing, but as argued before, the cost is constant). If, on the other hand, we are interested in inserting a new element at the front of the list (i.e. at index 0), it is necessary to shift each of elements that will ultimately follow it to positions with a higher index. This operation takes $O(n)$ time, and is the worst case. Insertions at intermediate locations take intermediate amounts of time, but the general insertion takes time proportional to the length of the list.

The costs associated with removing elements is analyzed in a similar fashion since, in the general case, items stored at higher locations must be shifted one element lower. Removal, like insertion is a linear ($O(n)$) operation.

When a sequence of m inserts (or deletions) are to be performed as part of a single operation (for example, inserting a `list` of elements into the middle of another `list` of n elements) a careful implementation will perform the necessary shifting of values only once, instead of m times. The result is an operation that takes $O(n+m)$ time instead of $O(n \cdot m)$ time. While it is tempting to think of such operations as being composed of many smaller tasks, we must be vigilant in seeking the most efficient approach.

4.2 Alternative `list` Implementations

As with any class, when we use the Python `list` we must accept the performance characteristics inherent in its design. In this section we discuss a number of alternative implementations of the `List` interface that provide alternate performance characteristics. For example, many of the structures we will investigate can insert a value at the beginning (or *head*) of the list in constant time. There are trade-offs, of course, and we highlight these as appropriate.

One alternative to the built-in `list` class dynamically allocates *nodes* to hold to each member of the list. A small number of links (typically one or two) are used to chain the nodes together into a list. Though the nodes may be allocated from different locations in memory, the references that interconnect these nodes make them appear adjacent. The "natural ordering" of values, then, is determined by the order of the nodes that appear as these nodes are encountered from head to tail. When values have to be inserted or removed, the "adjacent" values need not be relocated. Instead, links in each node are re-targeted to give the list its new structure. Generally, we call these structures *linked lists*.

4.2.1 Singly-linked Lists

When each data node references just the node that follows it in the list, we say the list is *singly linked*. The private `_Node` class for singly linked lists is fairly simple. (Again, recall that the leading underscore suggest the object is for private use.)

```
class _Node(object):
    __slots__ = ["_data", "_next"]

    def __init__(self, data=None, next=None):
        """Initialize node for singly linked list.

        Args:
            data: the data to be stored in this node.
            next: the node that follows this in the linked list.
        """
        self._data = data
        self._next = next

    @property
    def value(self):
        """The value stored in this node."""
        return self._data

    @property
    def next(self):
        """Reference to the node that follows this node in the list."""
        return self._next
```

```
@next.setter
def next(self, new_next):                                    25
    """Set next reference to new_next.

    Args:
    new_next: the new node to follow this one."""
    self._next = new_next
```

This structure is, essentially, a pair of references—one that refers to the data stored in the list, and another that is a link to the node that follows. The initializer, `__init__`, has keyword parameters with default values so that either parameter can be defaulted and so that when `_Nodes` are constructed, the parameter interpretation can be made explicit. In this class we have chosen to think of the slots as properties (through the use of the `@property` decorator) even though this essentially makes the slots public. Again, this approach allows us to hide the details of the implementation, however trivial, so that in the future we may chose to make changes in the design. For example, the use of properties to hide accessor methods makes it easier to instrument the code (for example, by adding counters) if we were interested in gathering statistics about `_Node` use.

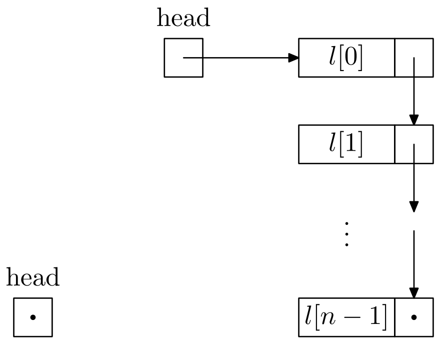


Figure 4.1 An empty singly-linked list (left) and one with n values (right).

Let's now consider how multiple `_Nodes` can be strung together, connected by `next` references, to form a new class, `LinkedList`. Internally, each `LinkedList` object keeps track of two private pieces of information, the *head* of the `LinkedList` (`_head`) and its length (`_len`). The head of the list points to the first `_Node` in the list, if there is one. All other `_Nodes` are indirectly referenced through the first node's `next` reference. To initialize the `LinkedList`, we set the `_head` reference to `None`, and we zero the `_len` slot. At this point, the list is in a consistent state (we will, for the moment, ignore the statements involving `frozen` and `hash`).

Our initializer, like that of the built-in `list` class, can take a source of data that can be used to prime the list.

```
def __init__(self, data=None, frozen=False):
    """Initialize a singly linked list."""
    self._hash = None
    self._head = None
    self._len = 0
    self._frozen = False
    if data is not None:
        self.extend(data)
    if frozen:
        self.freeze()
```

Because the `LinkedList` is in a consistent internal state, it is valid to call its `extend` method, which is responsible for efficiently extending the `LinkedList` with the values found in the "iterable" data source. We will discuss the details of `extend` later, as well.

Before we delve too far into the details of this class, we construct a private utility method, `_add`. This procedure adds a new value after a specific `_Node` (`loc`) or, if `loc` is omitted or is `None`, it places the new value at the head of the list.

```
@mutatormethod
def _add(self,value,loc=None):
    """Add an element to the beginning of the list, or after loc if loc.

    Args:
        value: the value to be added to the beginning of the list
        loc: if non-null, a _Node, after which the value is inserted.
    """
    if loc is None:
        self._head = _Node(value,self._head)
    else:
        loc.next = _Node(value,loc.next)
    self._len = self._len + 1
```

Clearly, this method is responsible for constructing a new `_Node` and updating the length of list, which has grown by one. It will be the common workhorse of methods that add values to the list.

It is useful to take a moment and think a little about the number of possible ways we might call `_add` on a list with n values. The method can be called with `loc` set to `None`, or it can be called with `loc` referencing any of the n `_Nodes` that are already in the list. There are $n + 1$ different ways to call `_add`. This parallels the fact that there are $n + 1$ different ways that a new value can be inserted into a list of n elements. This is a subtle proof that the default value of `loc`, `None`, is necessary for `_add` to be fully capable. Thinking in this way is to consider an *information theoretic argument* about the appropriateness of an approach. Confident that we were correct, we move on.

Let's think about how the length of the list is accessed externally. In Python, the size of any object that holds a variable number of values—a *container* class—is found by calling the utility function, `len`, on the object. This function, in turn calls the object's special method, `__len__`. For the `LinkedList` class, which carefully maintains its length in `_add` and `_free`, this method is quite simple:

```
def __len__(self):
    """Compute the length of the list."""
    return self._len
```

Throughout this text we will see several other special methods. These methods allow our classes to have the look-and-feel of built-in classes, so we implement them when they are meaningful.

Next, is the `append` method. Recall that this method is responsible for adding a single value to the end of the list:

```
@mutatormethod
def append(self, value):
    """Append a value at the end of the list (iterative version).

    Args:
       value: the value to be added to the end of the list.
    """
    loc = self._head
    if loc:
        while loc.next is not None:
            loc = loc.next
    self._add(value,loc)
```

This method iterates through the links of the list, looking for the node that has a `None` next reference—the last node in the list. The `_add` method is then used to construct a new terminal node. The last element of the list is then made to reference the new tail.

Exercise 4.2 *Suppose we kept an extra, unused node at the head of this list. How would this change our implementation of* `append`*?*

Notice that it is natural to use a `while` loop here and that a `for` loop is considerably more difficult to use efficiently. In other languages, like C or Java, this choice is less obvious.

It may be instructive to consider a recursive approach to implementing `append`:

```
@mutatormethod
def append_recursive(self, value, loc=None):
    """Append a value at the end of the list.

    Args:
        value: the value to be appended
        loc: if non-null, a _Node, after which the value is inserted.
    """
```

```
        if loc is None:
            # add to tail
            loc = self._head
        if loc is None or loc.next is None:
            # base case: we're at the tail
            self._add(value,loc)
        else:
            # problem reduction: append from one element below
            self.append_recursive(value,loc.next)
```

While the method can take two parameters, the typical external usage is to call it with just one—the value to be appended. The `loc` parameter is a pointer into the list. It is, if not `None`, a mechanism to facilitate the `append` operation's search for the last element of the list. (The new value will be inserted in a new `_Node` referenced by this last element.) When `loc` is `None`, the search begins from the list's head; the first `if` statement is responsible for this default behavior. The second `if` statement recursively traverses the `LinkedList`, searching for the tail, which is identified by a `next` field whose value is `None`.

Logically, these two implementations are identical. Most would consider the use of recursion to be natural and beautiful, but there are limits to the number of recursive calls that can be supported by Python's call stack.[1] It is likely, also, that the recursive implementation will suffer a performance penalty since there are, for long lists, many outstanding calls to `append` that are not present in the iterative case; they are pure overhead. Compilers for some languages are able to automatically convert simple tail-recursive forms (methods that end with a single recursive call) into the equivalent iterative, `while`-based implementation. Over time, of course, compiler technology improves and the beauty of recursive solutions will generally outweigh any decrease in performance. Still, it is one decision the implementor must consider, especially in production code. Fortunately, procedural abstraction hides this decision from our users. Any beauty here, is *sub rosa*.

Our initializer, recall, made use of the `extend` operation. This operation appends (possibly many) values to the end of the `LinkedList`. Individually, each `append` will take $O(n)$ time, but only because it takes $O(n)$ time just to find the end of the list. Our implementation of `extend` avoids repeatedly searching for the end of the list:

```
    def _findTail(self):
        """Find the last node in the list.

        Returns:
           A reference to the last node in the list.
        """
        loc = self._head
        if loc:
```

[1] Python's stack is limited to 1000 entries. This may be increased, but it is one way that Python can halt, not because of errors in logic, but because of resource issues.

```
            while loc.next is not None:
                loc = loc.next
        return loc

    @mutatormethod
    def extend(self, data):
        """Append values in iterable to list.

        Args:
            data: an iterable source for data.
        """
        tail = self._findTail()
        for value in data:
            self._add(value,tail)
            tail = self._head if tail is None else tail.next
```

A private helper method, `_findTail`, returns a reference to the last element, or `None`, if the list is empty.

Exercise 4.3 *Write a recursive version of* `_findTail`.

Once the tail is found, a value is added after the tail, and then the temporary tail reference is updated, sliding down to the next node. Notice again that the use of the `_add` procedure hides the details of updating the list length. It is tempting to avoid the $O(n)$ search for the tail of the list if no elements would ultimately be appended. A little thought, though, demonstrates that this would introduce an `if` statement into the `while` loop. The trade-off, then, would be to improve the performance of the relatively rare trivial `extend` at the cost of slowing down *every other case*, perhaps significantly.

Before we see how to remove objects from a list, it is useful to think about how to see if they're already there. Two methods `__contains__` and `count` (a sequence method peculiar to Python) are related. The special `__contains__` method (called when the user invokes the `in` operator; see page 7) simply traverses the elements of the list, from head to tail, and compares the object sought against the value referenced by the current node:

```
    def __contains__(self,value):
        """Return True iff value is contained in list."""
        loc = self._head
        while loc is not None:
            if loc.value == value:
                return True
            loc = loc.next
        return False
```

As soon as a reference is found (as identified by the `==` operator; recall our extended discussion of this on page 42), the procedure returns `True`. This behavior is important: a full traversal of an n element list takes $O(n)$ time, but the value we're looking for may be near the head of the list. We should return as soon as the result is known. Of course if the entire list is traversed and the `while` loop finishes, the value was not encountered and the result is `False`. While the worst case time is $O(n)$, in the expected case[2] finds the value much more quickly.

The `count` method counts how many equal values there are:

```
def count(self, value):
    """Count occurrences of value in list."""
    result = 0
    loc = self._head
    while loc is not None:
        if loc.value == value:
            result += 1
        loc = loc.next
    return result
```

The structure of this code is very similar to `__contains__`, but `count` must always traverse the entire list. Failure to consider all the elements in the list may yield incorrect results. Notice that `__contains__` *could* be written in terms of `count` but its performance would suffer.

Now, let's consider how to remove elements. Removing a value, in some sense in the "inverse" of the insertion of a value, so it would not be surprising if the structure of the methods that supports removal of values was similar to the what we've seen so far, for insertion. The counterpart to `_add` is a private utility method, `_free`:

```
@mutatormethod
def _free(self, target):
    result = target.next
    target.next = None
    self._len -= 1
    return result
```

When we're interested in removing a particular node, say `target`, we pass `target` to `_free` and the routine unlinks the target node from the list. It returns the portion of the list that hangs off the target node. This value is useful, because it is the new value of the `next` field in the `_Node` that precedes `target` *or* if `target` was at the head of the list, it is the new `_head` value. We will frequently see, then, a statement similar to

```
prior.next = self._free(prior.next)
```

Finally, `_free` is the place where we hide the details of reducing the length of the list.

When we want to remove a specific value from a `LinkedList`, we call `remove`.

```
@mutatormethod
def remove(self,value):
```

[2] The term *expected case* depends, of course, on the *distribution* of accesses that we might "expect". For example, in some situations we might know that we're always looking for a value that is known to be in the list (perhaps as part of an assert). In other situations we may be looking for values that have a non-uniform distribution. Expected case analysis is discussed in more detail in Section **??**.

```
        """Remove first occurance of value from list, or raise ValueError."""
        (prior,loc) = (None,self._head)
        while loc is not None:
            if loc.value == value:
                result = loc.value
                if prior is None:
                    self._head = self._free(loc)
                else:
                    prior.next = self._free(loc)
                return result
            (prior,loc) = (loc,loc.next)
        raise ValueError("Value not found.")
```

This method searches for the appropriate value, much like `__contains__`, but unlike `__contains__`, `remove` keeps a reference to the previous node (`prior`) so it may call `_free` and update `prior`'s next link.

Methods that Support Indexing

Perhaps the most important feature of any sequence is the ability to access any element of the `list` based on its offset from the beginning, or its *index*. Valid index values for an n element list are integers between 0 and $n - 1$. As a convenience, Python also allows one to index from the *end* of the list using negative index values: -1 means "the last element", and $-n$ indicates the first.

The workhorse of the `List` interface is the special method, `__getitem__`, that supports the index operator, `l[i]`. This method is passed an integer[3] that is interpreted as the index of the desired value.

```
    def __getitem__(self,i):
        """The ith element of self.  Does not support slices."""
        l = len(self)
        if i < 0:
            i += l
        if i < 0 or l <= i:
            raise IndexError("Index out of range.")
        loc = self._head
        for offset in range(i):
            loc = loc.next
        return loc.value
```

The first lines are responsible for converting a negative index into an appropriate positive index. If, at that point, the index is not valid, an exception is thrown. Since we know the index is valid, we can safely traverse the appropriate number of nodes, after the head node, to find the value to be returned.

The `insert` method is used to place a value in the list before the entry at a particular index. In a list, `l`, with `n` values, `l.insert(0,value)` is similar to

[3] Actually, `__getitem__` can accept a range of fairly versatile indexing specifications called *slices*, as hinted at earlier on page 10.

l._add(value), while l.insert(n,value) acts like l.append(value). Negative index values identify positions from the end of the list. Index values that are too small or too large to be an index for l, are silently reset, or *clamped*, to 0 or len(l), respectively. It is important that insert always works.

```
@mutatormethod
def insert(self,index,value):
    """Insert value in list at offset index."""
    # to support negative indexing, add length of list to index if index is negative
    l = len(self)
    if index < 0:
        index += l
    if index > l:
        index = l
    if index <= 0:
        self._add(value,None)
    else:
        prior = self._head
        for offset in range(index-1):
            prior = prior.next
        self._add(value,prior)
```

As with __getitem__, the first few lines of insert normalize the index value. An offset of zero simply adds the value at the head of the list, otherwise the value is added *after* the node that falls *before* the current node with the appropriate index.

One method, pop, uses an index to identify the value that is to be removed. As a convenience the index defaults to -1, allowing the user to remove the last element of the list.[4]

```
@mutatormethod
def pop(self, index=-1):
    """Remove value from end of list or at index, if provided."""
    l = len(self)
    if index < 0:
        index = index + l
    if index < 0 or l <= index:
        raise IndexError("Index out of range.")
    if index == 0:
        result = self._head.value
        self._head = self._free(self._head)
    else:
        prior = self._head
        for offset in range(index-1):
            prior = prior.next
        result = prior.next.value
```

[4] The pop method, along with append, supports the notation of a *stack*. We will see more about stacks when we discussion linear structures in Chapter 6.

```
            prior.next = self._free(prior.next)
        return result
```
Since this method removes values from the list, it makes use of the `_free` method.

Pros and Cons

The `LinkedList` class has some clear advantages. First, while each entry takes a little more space to store (an entire `_Node`, about twice the space of an entry in a `list`), the list maintains *exactly* the number of `Nodes` needed to store the entries. Each time a new value is added or removed, the same process of allocating or freeing memory is executed; there are no sudden *hits* associated with the occasional reallocation of `lists`. Finally, insertion of values at the head of the list, where the index is small, takes constant time. Recall that `lists` must move *all* the current data items to new slots each time.

But there are also disadvantages. Adding values to the *end* of the `LinkedList` can take a long time, simply because it takes a long time to *find* the end of the list. With the built-in `list` class, appending a value is a constant-time operation.

Is it possible to make the addition of a new value at the *end* of the `LinkedList` a faster operation? If we were to keep a pointer to the tail of the `LinkedList` (as we do with the head), then *appending* values must be accomplished in constant time. We must, of course, maintain three components of the internal state—the length, the head, and the tail. This takes a little more time, but it may be worth it if we want to access both the head and tail of the list quickly. But what happens when we *remove* the tail of the list? How do we find the penultimate element? In the next section we consider the implementation of linked lists with two pointers in each `_Node`.

4.2.2 Doubly-Linked Lists

Our implementation of singly-linked lists is, externally, probably sufficient for many tasks. As we have seen, it can provide some advantages over Python's built-in `list` class. But there are several ways in which the implementation lacks the *beauty* that one might imagine is possible. First, there are several unfortunate asymmetries:

1. The head and tail of the `LinkedList` are treated somewhat differently. This directly leads to differences in performance between head-oriented operations and tail-oriented operations. Since, for many applications the head and tail are likely to be frequently accessed, it would be nice to have these locations supported with relatively fast operations.

2. The internal `_add` and `_free` methods both require you to be able to access the node that falls *before* the target node. This makes for some admittedly awkward code that manages a `prior` reference rather than a `current` reference. Some of this can be resolved with careful use of recursion,

but the awkwardness of the operation is inherent in the logic that *all the references point in one direction*, toward the tail of the list. It would be nice to remove a node given just a reference to *that* node.

3. LinkedLists that are empty have a _head that is None. All other lists have _head referencing a _Node. This leads to a number of un-beautiful if statements that must test for these two different conditions. For example, in the _add method, there are two ways that a new _Node is linked into the list, depending on whether the list was initially empty or not. Unusual internal states (like an empty LinkedList) are called *boundary conditions* or *edge-* or *corner-cases*. Constantly having to worry about the potential boundary case slows down the code for *all* cases. We would like to avoid this.

Here, we consider a natural extension to singly-linked lists that establishes not only forward references (toward the tail of the list), but also backward references (toward the list head). The class is DoublyLinkedList. Admittedly, this will take more space to store, but the result may be a faster or—for some, a greater motivation—more beautiful solution to constructing lists from linked nodes. Our discussion lightly parallels that of the LinkedList class, but we highlight those features that demonstrate the relative beauty of the DoublyLinkedList implementation.

Like the LinkedList class, the DoublyLinkedList stores its values in a private[5] _Node class. Because the list will be doubly-linked, two references will have to be maintained in each node: the tail-ward _next and head-ward _prev.

While, in most cases, a _Node will hold a reference to valid user-supplied data, it will be useful for us to have some seemingly extraneous or *dummy nodes* that do not hold data (see Figure 4.2). These nodes ease the complexities of the edge-cases associated with the ends of the list. For accounting and verification purposes, it would be useful to be able to identify these dummy nodes. We could, for example, add an additional boolean ("Am I a dummy node?"), but that boolean would be a feature of *every* node in the list. Instead we will simply have the _value reference associated with a dummy node be a *reference to the node itself*. Since _Node references are not available outside the class, it would be impossible to for an external user to construct such a structure for their own purpose.[6]

The _Node implementation looks like this:

```
@checkdoc
```

[5] The class is not "as private" as private classes in languages like Java. Here, we simply mean that the class is not mentioned in the module variable __all__, the collection of identifiers that are made available with the from...import * statement. In any case, Python will not confuse the _Node class that supports the LinkedList with the _Node class that supports the DoublyLinkedList class.

[6] This is another information theoretic argument: because we're constructing a structure we can prove it cannot be confused with user data, thus we're able to convey this is a dummy/non-data node. The construction of valid dummy node definitions can be complicated. For example, we cannot use a value field of None as an indicator, since None is reasonably found in a list as user data.

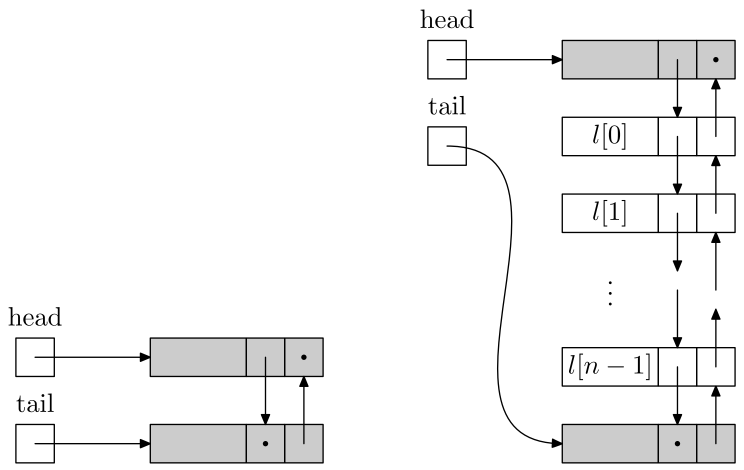


Figure 4.2 An empty doubly linked list, left, and a non-empty list, on right. Dummy nodes are shown in gray.

```
class _Node:
    __slots__ = ["_data", "_prev", "_next"]

    def __init__(self, data=None, prev=None, next=None, dummy=False):
        """Construct a linked list node."""
        self._data = self if dummy else data
        self._prev = prev
        self._next = next

    @property
    def value(self):
        """For non-dummy nodes, the value stored in this _Node."""
        assert self._data is not self
        return self._data

    @property
    def next(self):
        """The reference to the next element in list (toward tail)."""
        return self._next

    @next.setter
    def next(self, new_next):
        """Set next reference."""
        self._next = new_next
    @property
    def prev(self):
        """The reference to the previous element in list (toward head)."""
        return self._prev

    @prev.setter
    def prev(self, new_prev):
        """Set previous reference."""
        self._prev = new_prev
```

Typically, the `_Nodes` hold data; they're not "dummy" `_Nodes`. When a dummy node is constructed, the `_data` reference points to itself. If the `value` property accessor method is ever called on a dummy node, it fails the assertion check. In a working implementation it would be impossible for this assertion to fail; when we're satisfied our implementation is correct we can disable assertion testing (run Python with the `-O` switch), or comment out this particular assert.[7]

The initializer for the `DoublyLinkedList` parallels that of the `LinkedList`, except that the `_head` and `_tail` references are made to point to dummy nodes. These nodes, in turn, point to each other through their `next` or `prev` references. As the list grows, its `_Nodes` are found between `_head.next` and `_tail.prev`.

```
def __init__(self, data=None, frozen=False):
    """Construct a DoublyLinkedList structure."""
    self._head = _Node(dummy=True)
    self._tail = _Node(dummy=True)
    self._tail.prev = self._head
    self._head.next = self._tail
    self._len = 0
    if data:
        self.extend(data)
```

All insertion and removal of `_Nodes` from the `DoublyLinkedList` makes use of two private utility routines, `_link` and `_unlink`. Let's first consider `_unlink`. Suppose we wish to remove a an `item` from list `l`. We accomplish that with:

```
self._unlink(item)
```

Compare this with the the `_free` method in `LinkedList`. There, it was necessary for the *caller* of `_free` to re-orient the previous node's `next` reference because the previous node was not accessible from the removed node. Here, `_unlink` is able to accomplish that task directly. `_unlink` is written as follows:

```
@mutatormethod
def _unlink(self,node):
    """unlink this node from surrounding nodes"""
    node.prev.next = node.next
    node.next.prev = node.prev
```

The process of introducing a `_Node` into a list inverts the operation of `_unlink`:

[7] Commenting out the assertion is better than physically removing the assertion. Future readers of the code who see the assertion are then reminded of the choice.

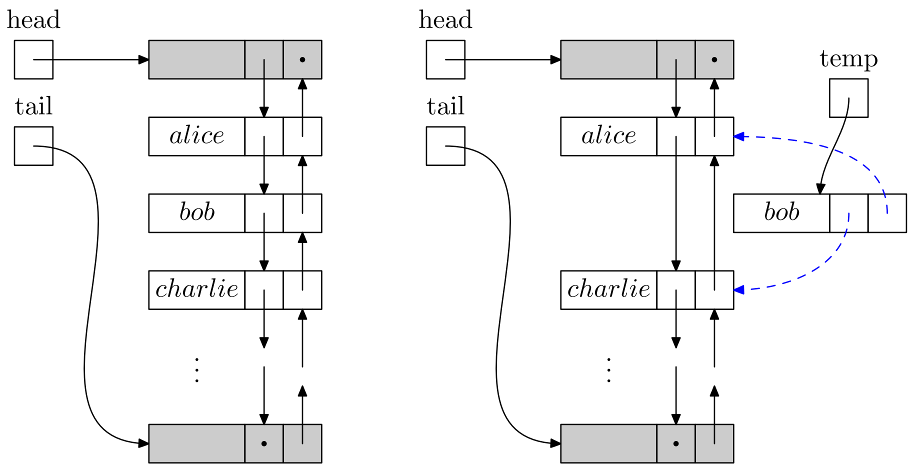


Figure 4.3 A doubly linked list before (left) and after (right) the action of `_unlink`. After the operation the node that has been removed (and referenced here, by a temporary reference) remembers its location in the list using the retained pointers. This makes re-insertion particularly quick.

```
    @mutatormethod
    def _link(self, node):
        """link to this node from its desired neighbors"""
        node.next.prev = node
        node.prev.next = node
```

Paired, these two methods form a beautiful way to temporarily remove (using `_unlink`) and re-introduce (using `_link`) a list element (see Figure 4.3).[8] Such operations are also useful when we want to move an element from one location to another, for example while sorting a list.

We can now cast the `_add` and `_free` methods we first saw associated with singly linked lists:

```
    @mutatormethod
    def _add(self, value, location):
        """Add a value in list after location (a _Node)"""
        # create a new node with desired references
        entry = _Node(value,prev=location,next=location.next)
        # incorporate it into the list
        self._link(entry)
        # logically add element to the list
```

[8] This observation lead Don Knuth to write one of the most interesting papers of the past few years, called *Dancing Links*, [**?**].

```
        self._len += 1

    @mutatormethod
    def _free(self,item):
        """Logically remove the indicated node from its list."""
        # unlink item from list
        self._unlink(item)
        # clear the references (for neatness)
        item.next = item.prev = None
        # logically remove element from the list
        self._len -= 1
```

Again, these operations are responsible for hiding the details of accounting (i.e. maintaining the list length) and for managing the allocation and disposal of nodes.

Most of the implementation of the `DoublyLinkedList` class follows the thinking of the `LinkedList` implementation. There are a couple of methods that can make real use of the linkage mechanics of `DoublyLinkedLists`. First, the `append` method is made considerably easier since the tail of the list can be found in constant time:

```
    @mutatormethod
    def append(self, value):
        """Append a value at the end of the list."""
        self._add(value,self._tail.prev)
```

Because of the reduced cost of the `append` operation, the `extend` operation can now be efficiently implemented in the naïve way, by simply appending each of the values found in the iterable `data`:

```
    @mutatormethod
    def extend(self, data):
        """Append values in iterable to list (as efficiently as possible)."""
        for x in data:
            self.append(x)
```

Finally, we look at the `insert` method. Remember that the cost of inserting a new value into an existing list is $O(n)$; the worst-case performance involves traversing the entire list to find the appropriate location. This could be improved by a factor of two by searching for the correct location from the beginning or the end of the list, whichever is closest:

```
    @mutatormethod
    def insert(self,index,value):
        """Insert value in list at offset index."""
        # normalize the index
        l = len(self)
        if index > l:
            index = l
        if index < -l:
            index = -l
        if index < 0:
            index += 1

        if index < l//2:
            # find previous element from head
            element = self._head
            for offset in range(index):
                element = element.next
        else:
            # find previous element from tail
            element = self._tail.prev
            for offset in range(l,index,-1):
                element = element.prev

        # insert element after location
        self._add(value, element)
```

Another approach might be to use the sign of the index to help find the appropriate location: positive indexing would traverse from the head of the list, while negative indexing would traverse from the tail. There is, however, no performance advantage to the signed technique; our approach always finds the appropriate spot as quickly as possible.

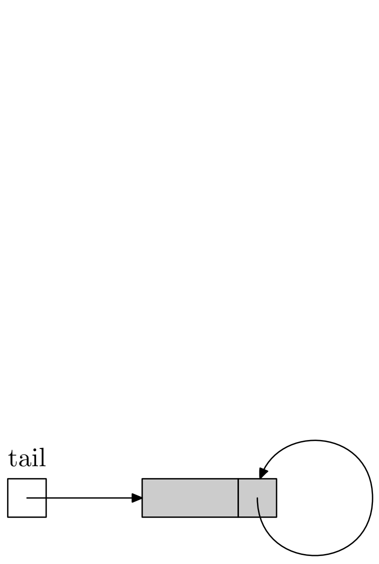
4.2.3 Circularly Linked Lists

The `DoublyLinkedList` implementation provides significant increases in performance for some important list operations. For example, operations on the tail of the list are just as fast as operations on the head of the list. This is not too surprising since the structure is very symmetric. This comes at a cost, however: we double the space needed for linking nodes. Is it possible to improve access to the tail of the list without increasing the reference space?

The answer is *Yes*. We note that the `LinkedList` contains one reference—at the tail of the list—that is always `None`. It serves as a *sentinel* value that indicates the end of the list. With care, we can recycle this reference to point to the *head* of the list, and still retain its sentinel properties.

As with `LinkedLists`, we use the singly linked `_Node` to hold list elements. Having learned from `DoublyLinkedLists`, however, we use a dummy node as a sentinel that appears between the tail and the head of the list. This is not necessary, but for the arguments made earlier, we opt to use the dummy node. Finally, instead of keeping track of the head of the list, we keep a single reference, to the tail (see Figure 4.4. We initialize the structure with the following method:

```
    def __init__(self, data=None, frozen=False):
        """Construct a LinkedList structure."""
        self._size = 0
        self._tail = _Node() # a dummy node
        self._tail.next = self._tail
        self._hash = None
```



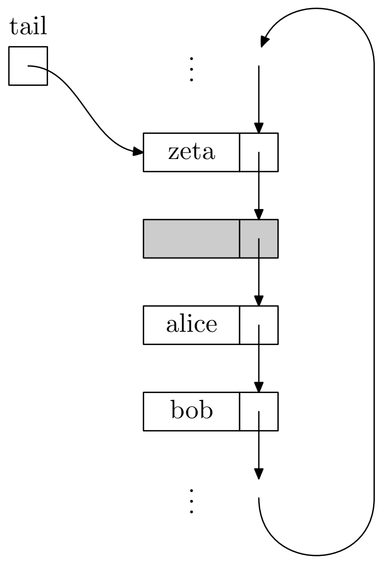


Figure 4.4 An empty circular list (left) and one with many nodes (right). Only the tail reference is maintained, and a dummy node is used to avoid special code for the empty list.

```
        self._frozen = False
        if data:
            self.extend(data)
        self._frozen = frozen
```

When the list is empty, of course, there is just one node, the dummy. Otherwise, the dummy node appears after the tail, and before the head of the list. For simplicity, we define a `_head` property:

```
    @property
    def _head(self):
        """A property derived from the tail of the list."""
        return self._tail.next.next
```

The `_add` and `_free` methods work with the node that falls before the desired location (because we only have forward links) and, while most boundary conditions have been eliminated by the use of the dummy node, they must constantly be on guard that the node referenced by `_tail` is no longer the last (in `_add`) or that it has been removed (in `_tail`). In both cases, the `_tail` must be updated:

```
    @mutatormethod
    def _add(self,value,location=None):
        """Add an element to the beginning of the list, or after loc if loc."""
        if location is None:
            location = self._tail.next
        element = _Node(value,next=location.next)
        location.next = element
        if location == self._tail:
            self._tail = self._tail.next
        self._size += 1

    def _free(self,previous):
        element = previous.next
        previous.next = element.next
        if self._tail == element:
            self._tail = previous
        element.next = None
        self._size -= 1
        return element.value
```

These few considerations are sufficient to implement the circular list. All other methods are simply borrowed from the lists we have seen before.

Exercise 4.4 *The* `pop` *method is typically used to remove an element from the tail of the list. Write the* `pop` *method for* `CircularList`*.*

In almost any case where a `LinkedList` is satisfactory, a `CircularList` is as efficient, or better. The only cost comes at the (very slight) performance decrease for methods that work with the head of the list, which is always indirectly accessed in the `CircularList`.

4.3 Discussion

It is useful to reflect on the various choices provided by our implementations of the `List` class.

The built-in `list` class is compact and, if one knows the index of an element in the list, the element can be accessed quickly—in constant time. Searching for values is linear (as it is with all our implementations). Inserting a value causes all elements that will ultimately appear later in the list to be shifted. This is a linear operation, and can be quite costly if the list is large and the index is small. The only location where insertion is not costly is at the high-indexed tail of the list. Finally, when a *list* grows, it may have to be copied to a new larger allocation of memory. Through careful accounting and preallocation, we can ameliorate the expense, but it comes at the cost of increasingly expensive (but decreasingly frequent) reallocations of memory. This can lead to occasional unexpected delays.

The singly linked list classes provide the same interface as the `list` class. They are slightly less space efficient, but their incremental growth is more graceful. Once a correct location for insertion (or removal) is found the inserting (removing) of a value can be accomplished quite efficiently. These structures are less efficient at directly indexed operations—operations that are motivating concepts behind the `list` design. The `CircularList` keeps track of both head and tail allowing for fast insertions, but removing values from the tail remains a linear operation.

Supporting a linked list implementation with links in both directions makes it a very symmetric data structure. Its performance is very similar to the other linked list classes. Backwards linking allows for efficient removal of values from the tail. Finally, it is quite easy to temporarily remove values from a doubly linked list and later undo the operation; all of the information necessary is stored directly within the node. In space conscious applications, however, the overhead of `DoublyLinkedList` is probably prohibitive.

Self Check Problems

Solutions to these problems begin on page 313.

4.1 Self-check problem.

Problems

Selected solutions to problems begin on page 313.

4.1 Regular problem.

Chapter 5

Iteration

We have spent a considerable amount of time thinking about how *data abstraction* can be used to hide the unnecessary details of an implementation. In this chapter, we apply the same principles of abstraction to the problem of hiding or *encapsulating* basic *control* mechanisms. In particular, we look at a powerful abstraction called *iteration*, which is the basis for Python's main looping construct, the `for` loop. In the first sections of this chapter, we think about general abstractions of iteration. Our last topics consider how these abstractions can be used as a unified mechanism for traversing data structures without exposing the details otherwise hidden by the data abstraction.

5.1 The Iteration Abstraction

Suppose we are interested in using the first few prime numbers for some purpose. We may be simply printing out the values, or we may wish to use them to break another number into its unique product of primes. Here's one approach we might use to consider each prime in turn:

```
def prime_process(max):
    sieve = []
    n = 2
    while n < max:
        is_prime = True
        for p in sieve:
            # have we exceeded possible factor size?
            if p*p > n:
                break
            # no.  is n composite (e.g. divisible by p)?
            if n % p == 0:
                is_prime = False
                break
        # if n has withstood the prime sieve, it's a new prime
        if is_prime:
            sieve.append(n)
            # heart of the problem: use n
            print("prime {}: {}".format(len(sieve),n))
        # consider next candidate
        n = 3 if n == 2 else n+2
```

Notice that while most of the code is focused on generating the primes, the part that is responsible for consuming the values (here, the `print` statement on line 18) is actually embedded in the code itself.

Our code would be improved if we could simply generate a list of primes, and then make use of that list elsewhere. We could construct and return a list:

```
def firstPrimes(max=None):
    sieve = []
    n = 2
    while (max is None) or (n < max):
        is_prime = True
        for p in sieve:
            if p*p > n:
                break
            if n % p == 0:
                is_prime = False
                break
        if is_prime:
            sieve.append(n)
        n = 3 if n == 2 else n+2
    return sieve
```

Now, the code that computes the primes has been logically separated from the code that makes use of them:

```
for p in firstPrimes(100):
    print(p)
```

Now, suppose we are interested in using our new prime-generating utility to compute a list of the prime factors of a number. We simply try reducing our number by each of the primes in the list, keeping track of the primes involved at each stage:

```
def primeFactors(n, maxp=100):
    result = []
    for p in firstPrimes(maxp):
        if n == 1:
            break
        else:
            while n % p == 0:
                result.append(p)
                n //= p
    return result
```

We must be careful, though, because our list of primes is limited to those that are less than 100. If we encounter a number that has a prime factor greater than 100 our procedure will produce the wrong factorization. How can we know how many primes are necessary? One approach, of course, is to compute all the primes up to the square root of the number at hand, and then we can be guaranteed that we have a sufficient number. Still, this approach is unsatisfying because much of the prime calculation may be unnecessary.

Fortunately, Python has a mechanism for computing values *lazily* or *on demand*. The idea is to *generate* the values, only as they are needed. Instead of returning a list of primes that have been computed[1] we *yield* each value as it is computed. The action of yielding a value turns over the computation to the consumer of the values. Only when the consumer needs another value does the process resume, picking up where it left off. Here's our modified code, `primes`:

```
def primes():
    sieve = []
    n = 2
    while True:
        is_prime = True
        for p in sieve:
            if p*p > n:
                break
            if n % p == 0:
                is_prime = False
                break
        if is_prime:
            sieve.append(n)
            yield n
        n = 3 if n == 2 else n+2
```

Our `for` loop can make use of the `primes` method, just as it had `firstPrimes`:

```
def primeFactors2(n):
    result = []
    for p in primes():
        if n == 1:
            break
        else:
            while n % p == 0:
                result.append(p)
                n //= p
    return result
```

But there is an important distinction between the two approaches. The `firstPrimes` method computed and returned the list of primes *before* the `for` loop was able to consider even the first prime. The `for` loop was iterating over the `list` that was returned from the `firstPrimes` method. In the `primes` implementation, the generator produces values for the `for` loop *on demand*. We say that the computation is *lazy* and does not compute any primes until asked.

Because `primes` does not generate any primes that are not used, we do not need to place a limit on how high it might go. We can uncover some of the details behind how the generator does its job by considering a slightly cleaner implementation that treats the `primes` method as a *generator*. Generators—

[1] We will still want to keep track of a list of primes, but only for internal use. This approach, of course, has all the same motivations we outlined when we discussed the abstract data type; here we're simply hiding the implementation of the determination of prime values.

signaled by the use of the `yield` statement—are actually classes that wrap the intermediate state of the computation. The initializer creates a new version of the generator, and the special method `__next__` forces the generator to compute the next value to be yielded:

```
class primeGen(Iterable):

    __slots__ = [ "_sieve", "_n" ]

    def __init__(self):
        self._sieve = []
        self._n = 2

    def __next__(self):
        yielding = False
        while not yielding:
            is_prime = True
            for p in self._sieve:
                if p*p > self._n:
                    break
                if self._n % p == 0:
                    is_prime = False
                    break
            if is_prime:
                self._sieve.append(self._n)
                # effectively yield the use of the machine
                # returning _n as the computed value
                result = self._n
                yielding = True
            self._n = 3 if self._n == 2 else self._n+2
        return result

    def __iter__(self):
        return self
```

When the `primes` generator is defined, the effect is to essentially generate the `primesGen` class. As with many of the special methods that are available in Python, the `__next__` method for objects of this class is indirectly invoked by the builtin method `next`[2]. Here is how we might explicitly make use of this approach:

```
def primeFactors3(n):
    result = []
    primeSource = primeGen()
    p = next(primeSource)
```

[2] This may seem strange to non-Python users. The reason that special methods are called in this indirect way is to help the language verify the correct use of polymorphism and to better handle targets that are `None`.

```
        while n > 1:
            if n % p == 0:
                result.append(p)
                n //= p
            else:
                p = next(primeSource)
        return result
```

As a final exercise, we consider how we might rewrite our prime factor computation as a generator. Here, the values that should be produced are the (possibly repeated) prime factors, and there is always a *finite number of them*. Because of this, it is necessary for our generator to indicate the end of the sequence of values. Fortunately, Python has a builtin exception class dedicated to this purpose, `StopIteration`. When this exception is raised it is an indication that any `for` loop that is consuming the values produced by the generator should terminate iteration.

Here, then, is our lazy generator for prime factors of a number, `primeFactorGen`:

```
def primeFactorGen(n):
    for p in primeGen():
        if n == 1:
            raise StopIteration()
        while n % p == 0:
            yield p
            n //= p
```

This could be used either as the basis of a `for` loop:

```
print([p for p in primeFactorGen(234234234)])
```

or as an equivalent loop that explicitly terminates on a `StopIteration` signal:

```
i = primeFactorGen(234234)
results = []
while True:
    try:
        results.append(next(i))
    except StopIteration:
        break
print(results)
```

Generators provide a powerful tool for efficiently controlling the iteration abstraction. In their purest forms they allow programmers to specify general algorithms using lazy computation—computation that unfolds only as it is needed. In a very specific setting, however, they provide an equally powerful mechanism for container classes to allow traversal over their values without sacrificing the hiding of the implementation. We turn our attention, then, to the development of iterators for simple container classes.

5.2 Traversals of Data Structures

One of the most important uses of *container classes*—that is, classes whose purpose is to hold (perhaps multiple) instances of other objects—is to be the target of efficient *traversals*. A simple motivation is, of course, just printing out the internal state of a container class. A simple (but general) approach to writing the function that prints the representation of a container class might be

```
def __repr__(self):
    for item in self:
        print(item)
```

As we have seen in the preceding sections, the `for` loop targets an `Iterable` object; the only requirement of an `Iterable` object is that calling the special method `__iter__` returns a generator that produces a useful, and typically finite, stream of values. This suggests that a container class must typically have an `__iter__` method that returns a generator that, over time, delivers the contained values. How is this accomplished?

5.2.1 Implementing Structure Traversal

Let's look at a common target of iteration, the `list`. Because we frequently traverse lists, it defines the special `__iter__` method as follows:

```
def __iter__(self):
    i = 0
    while True:
        try:
            yield self[i]    # or self.__getitem__(i)
            i += 1
        except IndexError:
            break
```

Again, the appearance of the `yield` keyword in the definition of the `__iter__` method causes Python to construct a `Iterable` class whose yielded values are lazily computed by the special `__next__` method. In this way, Python works hard to help the programmer to hide the details how iteration works on the `list` class. For example, it makes it possible for us to cast many interesting algorithms on `lists` without ever directly indexing the `list`. The power is not in *avoiding* those accesses, but in *providing alternative abstractions* that support common programming idioms. This is a subtle observation: that powerful techniques arise from supporting a diverse and sometimes duplicative set of operations on data structures.

The efficiency, then, of the traversal of the list `l`

```
for item in l:
    ...
```

is the same as the efficiency of the more explicit approach:

```
i = 0
while True:
    try:
    item = l[i]                                            4
except IndexError:                                         5
    break
...
i += 1
```

When a class does not define the special `__iter__` method, Python will provide an implementation that looks like the one for `lists`, by default. In other words, the definition of the `__getitem__` method, in a very indirect way, allows a data structure to be the target of a `for` loop. (It is instructive to write the simplest class you can whose traversal yields the values 0 through 9; see Problem **??**.)

Suppose, however, that `l` was a `LinkedList`. Then the indexing operation of line 4 in the previous code can be costly; we have seen that for linked structures, the `__getitem__` method takes time that is $O(n)$ in the worst case. Can an efficient traversal be constructed?

The answer, of course, is *Yes*. What is required is an alternative specification of the `__iter__` generator that generates the stream of values without giving away any of the hidden details of the implementation. Here, for example, is suitable implementation of the iteration method for the `LinkedList` class:

```
    def __iter__(self):
        """Iterate over the values of this list in the order they appear."""
        current = self._head
        while current is not None:
            yield current.value
            current = current.next
```

Because the local variable `current` keeps track of the state of the traversal between successive `yield` statements, the iterator is always able to quickly access the next element in the traversal. (This is opposed to having to, in the `__getitem__` method, restart the search for the appropriate element from the head of the linked list.) Now, we see that, while *random access* of linked list structures can be less efficient than for built-in `list` objects, the iterative *traversal* of linked lists is just as efficient as for their built-in counterparts.

We see, then, that the support of efficient traversals of container classes involves the careful implementation of one of two special methods. If there is a natural interpretation of the indexing operation, `c[i]`, the special method `__getitem__` should be provided. Since nearly every class holds a finite number of values, the value of the index should be verified as a valid index and, if it is not valid, an `IndexError` exception should be raised. Typically, the index is an integer, though we will see many classes where this need not be the case. If indexing can be meaningfully performed with small non-negative integers, Python can be depended upon to write an equally meaningful method for iterating across the container object. It may not be *efficient* (as was the case for all of our linked structures), but it will work correctly. Where traversal efficiency can be improved—it nearly always can, with a little state—a class-specific implementation of the special `__iter__` method will be an improvement over Python's default method. When indexing of a structure by small positive integers is not meaningful, a traversal mechanism is typically still very useful but

Python can not be depended upon to provide a useful mechanism. It must specified directly or inherited from superclasses.

5.2.2 Use of Structure Traversals

Although we often imagine the main use of iterators will be external to the definition of the class itself, one of the most important uses is the implementation efficient class methods. On page 79, for example, we saw the implementation of the special method `__contains__` in the `LinkedList` class. This method implements the set containment operator, `in`, which is `True` when the value on the left side is found within the container on the right. It is relatively easy to write equivalent code using iteration:

```
def __contains__(self,value):
    """Return True if value is in LinkedList."""
    for item in self:
        if item == value:
            return True
    return False
```

Many container classes, and especially those that implement the `List` interface support a wide variety of special methods. Most of these may be cast in terms of high-level operations that involve traversals and other special methods. As a demonstration of the power of traversals to support efficient operations, we evaluate a few of these methods.

First, we consider the `extend` method of the linked list class. An important feature of this operation is the traversal of the `Iterable` that is passed in:

```
def extend(self, data):
    """Append values in iterable to list.

    Args:
        data: an iterable source for data.
    """
    for value in data:
```

Here, the target of the iteration is a general `Iterable` object. In fact, it may be another instance of the `LinkedList` class. This might have been called, for example, from the `__add__` method, which is responsible for `LinkedList` concatenation with the + operator. There, we use the `extend` method twice.

```
def __add__(self,iterable):
    """Construct a new list with values of self as well as iterable."""
    result = LinkedList(self)
    result.extend(iterable)
    return result
```

The first call is within the initializer for the `result` object. Since we have efficiently implemented the special `__iter__` method for `self`, this operation will be as efficient as possible—linear in the list length. The second call to `extend`, which is explicit, is responsible for the catenation of values from the `iterable` object. That object may be a `LinkedList` instance; if it is, we can rest assured that the traversal is efficient.

Two special methods are often implemented quite early in the development process, mainly to support debugging. The first, `__str__`, is responsible for converting the object to a compact and “informal” string representation. This method is called whenever the object is used in a place where a string is called for.[3] One method might be to simply construct, by concatenation, a long, comma-separated list of string representations of each item in the container:

```
def __str__(self):
    result = ""
    for item in self:
        if result:
            result += ", "
        result += str(item)
    return result
```

Here, the container class (notice we don’t know what *kind* of container this is, just that it supports iteration) is traversed and each value encountered is produces a string representation of itself. These are concatenated together to form the final string representation.

This approach, while effective, is quite inefficient. Since strings are not mutable in Python, each catenation operation forces a new, larger copy of the representation to be constructed. The result is an operation whose number of character copies may grown as $O(n^2)$, not $O(n)$. An improvement on this approach is to use the `join` operation, which efficiently concatenates many strings together, with intervening separator strings (here, a comma):

```
def __str__(self):
    item_strings = [str(item) for item in self]
    return ", ".join(item_strings)
```

Here, `item_strings` is the list of collected string representations, constructed through a *list comprehension*. The comprehension operation, like the `for` loop, constructs an iterator to perform the traversal.

While we’re thinking about string-equivalents for container classes, it’s worth thinking about the `__repr__` special method. This method, like `__str__`, constructs a string representation of an object, but it typically meets one more requirement—that it be a valid Python expression that, when evaluated,[4] constructs an equal object. Since container classes can typically be constructed by providing a list of initial values, we can “cheat” by producing the list representation of the container class, wrapped by the appropriate constructor:

```
def __repr__(self):
    return "{}({})".format(type(self).__name__,repr(list(self)))
```

[3] This plays much the same role as the `toString` method in Java.

[4] Because Python is an interpreter, it is quite simple to evaluate an expression that appears within string `s`: one simply calls `eval(s)`. Thus, it should be the case that `obj == eval(repr(obj))` always returns `True`.

The phrase `type(self).__name__` produces the name of the particular container class at hand.

5.3 Iteration-based Utilities

Early in learning Python, most programmers write a few lines that are similar to:

```
for value in range(10):
    print(value)
```

The output, of course, we expect to be:

```
0
1
2
3
4
5
6
7
8
9
```

In version 3 of Python, the `range` class is implemented as a very simple generator, with the crudest equivalent being:

```
class range(Iterable):
    __slots__ = [ "_min", "_max"]

    def __init__(self,*args):
        self._min = args[0] if len(args) > 1 else 0
        self._max = args[-1]

    def __iter__(self):
        current = self._min
        while current < self._max:
            yield current
            current += 1
```

The initializer takes interprets one or two arguments[5] as the bounds on a range of values to be generated. When necessary, the `__iter__` method performs the idiomatic loop that yields each value in turn. The builtin `range` class is only slightly more complex, supporting increments or *strides* other than 1 and supporting a small number of simple features.[6] Python's `count` iterator is even

[5] The builtin `range` object is much more complex, taking as many as three arguments, as well as slice values.

[6] The `slice` class is a builtin class that provides the mechanism for complex slice indexing through the `__getitem__` method. While `slice` is not iterable, its provides an `indices(len)` method that returns a list of indicies for a container, provided the container has length `len`. The resulting tuple,

more simplistic: it simply generates a stream of values, starting at 0 (if 0 is not ideal, the starting value can specified as the sole parameter to the initializer).

Python provides a number of tools that allow one to manipulate iterators. The `zip` method, for example, allows one to form tuples of values that appear from one or more iterators. One of the more common uses for `zip` is to perform parallel traversals across two or more lists:

```
for (index, value) in zip(count(),container):
    print("container[{}] = {}".format(index,value))
```

The stream of tuples produced finishes whenever any of the subordinate iterators runs dry. How might one right such a method?

The implementation of `zip` makes use of several important Python features. First, all of the arguments for a procedure (here, the subordinate iterators) can be gathered together using the `*` specifier for exactly one argument that appears after each of the positional parameters. This argument is bound to a list of the arguments that remain after positional and keyword arguments have been accounted for.[7] Second, `zip` makes heavy use of explicit and implicitly driven iteration, allowing it to combine an arbitrarily large number of streams.

Here is one implementation of `zip`:

```
def zip(*args):
    # collect iterators for each of the subordinate iterables
    iters = tuple((iter(arg) for arg in args))               3
    while True:
        values = []                                          5
        for i in iters:
            try:
                v = next(i)                                  8
                values.append(v)
            except StopIteration: # stems from next(i)       10
                return
        yield tuple(values)
```

This is a subtle construction. On line 3, we construct a tuple that contains all of the iterators associated with each of the iterable objects passed in. Some of the actual parameters, may, of course, be iterators themselves. In such cases `iter` simply returns a reference to the outstanding iterator. In all other cases, a fresh iterator is constructed to traverse the iterable object. It's important that these values be collected *before* `zip` does its work; calling `iter` multiple times on an iterable would construct a new iterator, which is not what we want. The use of a `tuple` to collect these values suggests that the collection of iterator references will not change, though the individual iterators will, of course, change state over the course of `zip`'s action.

---

then, can be used as the arguments (using the `*` operator) to initialize a range object which you can then use to traverse the indicated portion of the container.

[7] A similar feature, not used here, is the `**` specifier that identifies a formal parameter that is to be bound to a *dictionary* of all the keyword-value pairs specified in as actual parameters. We learn more about dictionaries in Chapter 14.

The `values` list is intended to collect the next value from each of the subordinate iterators. This might not be successful, of course, if one of the iterators expires during a call to `next`. This is indicated by the raising of a `StopIteration` exception, caught on line 10. When that occurs, the complete list of values cannot be filled out, so the generator terminates by explicitly returning. This causes a `StopIteration` exception to be generated on behalf of the `zip` iterator. One might be tempted to simply re-raise the exception that was caught, but this generates is not a proper way to terminate a generator; the return is preferred. If the collection of next values is successful, the tuple of values is generated.

It might be tempting to use a tuple comprehension[8] similar to:

```
try:
    yield tuple(next(i) for i in iters)
except StopIteration:
    return
```

but something subtle happens here. This code is exactly the same as:

```
try:
    temp = []
    iters_iter = iter(iters)
    while True:
        try:
            i = next(iters_iter)
            temp.append(next(i))
        except StopIteration:
            break
    yield tuple(temp)
except StopIteration:
    return
```

In the comprehension, the `try-except` statement is attempting to catch the `StopIteration` exception that might be raised when `next` runs an iterator dry. Unfortunately, the same `StopIteration` exception is used to terminate the comprehension, so the result is that a tuple is always constructed, with those values that were successfully collected before the terminated iterator. The principle, then is to avoid, whenever possible, writing comprehensions whose expressions may throw exceptions; the results of comprehensions constructed in this way can be unexpected.

Exercise 5.1 *What happens when you call* `zip` *with one iterable object?*

[8] Actually, there is no *tuple comprehension* mechanism. Instead, a parenthesized comprehension expression indicates the construction of a *generator* which, of course, is iterable, and may be used in the `tuple` initializer. When a generator is used in this way in a method call, the (seemingly redundant) parentheses may be eliminated. In fact, the reader is encouraged to retain these parentheses to make clear the intermediate generator comprehension.

Exercise 5.2 *What happens when you provide* `zip` no *iterable objects? Is this a reasonable behavior? If not, is there an elegant fix?*

Python provides, through its `itertools` module, a versatile collection of utilities, like `zip`, that manipulate iterators and generators efficiently. A few of them are listed in Table 5.1.

Tool	**Purpose**
`count([start=0[, step=1]])`	A stream of values counting from `start` by step.

Figure 5.1 Iterator manipulation tools from the `itertools` package.

Chapter 6

Linear Structures

The classes that implement the `List` interface support a large number of operations with varied efficiency. Sometimes when we design new data structures we *strip away* operations in an attempt to *restrict access*. In the case of *linear* structures, the most important operations in the interface are those that add and remove values. Because it is not necessary, for example, to support random access, the designer of linear structures is able to efficiently implement add and remove operations in novel ways.

The two main linear structures are the *stack* and the `queue`. These two general structures are distinguished by the relationship between the ordering of values placed within the structure and the ordering of the values removed (rewrite). Because of their general utility, linear structures are central to the control of the flow of data in many important algorithms.

6.1 The `Linear` Interface

The abstract base class that is the root of all linear data structures is the `Linear` class. This method provides very rudimentary access through the two primary methods, `add` and `remove`:

```
class Linear(Iterable, Sized, Freezable):
    """
    An abstract base class for classes that primarily support organization
    through add and remove operations.
    """

    __slots__ = ["_frozen", "_hash"]

    def __init__(self):
        """Construct an unfrozen Linear data structure."""
        self._hash = None
        self._frozen = False

    @abc.abstractmethod
    def add(self, value):
        """Insert value into data structure."""
        ...
```

```
@abc.abstractmethod
def remove(self):
    """Remove next value from data structure."""
    ...

@abc.abstractmethod
def __len__(self):
    """The number of items in the data structure."""
    ...

@property
def empty(self):
    """Data structure contains no items."""
    return len(self) == 0

@property
def full(self):
    """Data structure has no more room."""
    return False;

@abc.abstractmethod
def peek(self):
    """The next value that will be removed."""
    ...

@abc.abstractmethod
def __iter__(self):
    """An iterator over the data structure."""
    ...

def __str__(self):
    """Return a string representation of this linear."""
    return str(list(self));

def __repr__(self):
    """Parsable string representation of the data structure."""
    return self.__class__.__name__ + "(" + str(self) + ")"

def __eq__(self, other):
    """Data structure has equivalent contents in the same order as other."""
    if not isinstance(other,Linear) or len(self) != len(other):
        return False
    for x,y in zip(self, other):
        if x != y:
            return False
    return True
```

Clearly, any value that is removed from a linear structure has already been added. The order in which values appear through successive remove operations, however, is not specified. For example, a *stack* object always returns the value which was added to the stack most recently. We might decide, however, to return the value that was added least recently (a *queue*), or the value that is largest or smallest (a *priority queue*). While an algorithm may only make use of the add and remove methods, the possible re-orderings of values as they pass through the linear structure typically plays a pivotal role in making the algorithm efficient. The limitations placed on the `Linear` interface help to guarantee that the relationship between the `add`s and `remove`s is not tainted by other less important operations.

6.2 The Stack

A *stack* is a linear structure that returns the value that was most recently added. For this reason, we formally call this structure a *last in, first out* or *LIFO* (pronounced *lie-foe*) structure. Stacks play an important role in algorithms that focus attention on subproblems as part of the process of solving more general problems. Recursion, typically, is like this as well: general problems are solved by *first* solving subproblems. As a result, a *call stack* plays an important role in keeping track of the progress of methods as they call each other: calling a method *adds*, for example, its *context* (local variables and a pointer to the currently executing statement) to the call stack, and returning from a procedure causes its context to be removed and discarded.

We use two different interfaces to describe a particular implementation of a `Stack`. The first interface provides aliases for `Linear` methods that are particular to LIFO structures: `push`, `pop`, and `top`:

```
@checkdoc
class LIFO(metaclass=abc.ABCMeta):
    """
    A base class for stack-like objects.
    """

    def push(self, value):
        """Insert value into data structure."""
        self.add(value)

    def pop(self):
        """Remove value from data structure."""
        return self.remove()

    def top(self):
        """Show next value to be removed from structure."""
        return self.peek()
```

In addition to these alternative methods, the LIFO interfaces suggests that any class that implements the interface abides by the reordering policy that the last items placed in the linear are the first to come out. Nothing in the interface *actually* guarantees that fact, but the implementation of the interface provides a type-oriented way for checking whether a structure implements the policy. A subtle aspect of this interface is the class header, which declares metaclass=abc.ABCMeta. When we wish to implement abstract base classes that are roots—that do not extend another class—it is necessary to declare the class in this manner to include it in the abstract base class hierarchy.[1]

It is natural to think of storing the values of a stack in some form of List. Indeed, the List interface provides a routine, pop, that suggests that the implementors had imagined this use. Since the list version of pop defaults to removing the element at the far end of the list (at index -1), we will place the elements of the stack in that orientation. This has the advantage, of course, that push/add and pop/remove are performed where they are most efficiently implemented in lists, in constant ($O(1)$) time. This implementation also has the feature that the structure can grow to be arbitrarily large; it is unbounded. We can actually implement, as an abstract class (i.e. a class with unspecified, abstract methods), much of the implementation as an UnboundedLinear structure. Here are the details:

```
class UnboundedLinear(Linear):
    __slots__ = ["_data"]

    def __init__(self, container=list):
        """Partially construct a linear."""
        super().__init__()
        self._data = container()
        # subclasses of this type must perform further initialization

    def add(self, value):
        """Add a value to the linear structure."""
        self._data.append(value)

    @abc.abstractmethod
    def remove(self):
        """Remove a value from the linear strucuture."""
        ...

    @abc.abstractmethod
```

[1] This is not ideal. It reflects the development of Python over many years. Since hierarchies of incompletely specified classes (abc hierarchy) are a relatively new addition to Python, the default behavior (for backwards compatibility) is not to place new classes in this hierarchy. For technical reasons, it is not ideal to mix old-style and new-style class definitions, so we opt to include all our structure classes in the abc hierarchy by directly specifying the metaclass, as we have here, or by extending an existing structure (as we do much of the the time).

```
    def peek(self):                                        20
        """The next value that will be popped.  Equivalent to top."""
        ...

    def __len__(self):
        """The number of items in the linear structure."""
        return len(self._data)
```

Though not all the details of the implementation can be predicted at this point, because we were motivated by the particular efficiencies of a `list`, we commit to using some form of `List` implementation as the internal container for our unbounded `Linear` classes. Here, the initializer takes a parameter, `container`, that is bound to the class associated with the desired container object. In Python, classes are, themselves, objects. Calling these class-objects creates instances of that class. Here, the variable `container` is simply a formal-parameter, a stand-in, for the desired class. On line 7 the underlying `List` is allocated.

While we cannot commit to a reordering policy, here, without loss of generality, we decide to `add` elements to the tail of the underlying list. We could have made the decision to `add` them to the head of the list, but then, in the case of `Stack` implementations, we would be forced to remove or `pop` values from the head, not the tail, which seems contrary to the default behavior of `list`'s `pop` method. Because we can't commit to a remove implementation, `peek` (which allows us to see the about-to-be-`removed` value) must also remain abstract. The special method `__len__` simply returns the length of the underlying container.

Finally, we can develop an implementation of a stack. This structure is an unbounded LIFO structure, so we will extend *two* abstract base classes, `UnboundedLinear` and `LIFO`, as we have just seen. You should note that these two interfaces do not overlap—they declare methods with different signatures—therefore any reference to a method specified by either class is uniquely defined.

```
class Stack(UnboundedLinear,LIFO,Freezable):

    __slots__ = ["_data", "_frozen", "_hash"]

    def __init__(self, data=None, frozen=False, container=list):
        """Construct a Stack from an iterable source."""
        super().__init__(container=container)
        if data:
            for v in reversed(data):
                self.push(v)
        self._frozen = frozen
```

The initializer provides a mechanism for declaring the container type, as discussed earlier. This is simply passed to the `UnboundedLinear` class for allocation, and then the structure is extended by pushing any initial values. While, internally, we imagine the stack placed in the list with the top on the right, externally, we represent it with the top on the left. This requires us to *reverse* the initializing values to ensure they appear in the correct order.

Most important are the implementation of `remove` and `peek`. These values, for stacks, must work on the tail end of the list.

```
def remove(self):
    """Remove and return a value from the linear structure."""
    return self._data.pop(); # pop tail (a list method)

def peek(self):
    """The next value that will be popped.  Equivalent to top."""
    assert not self.empty
    return self._data[-1]
```

Notice that `remove` calls `pop` on the underlying `List`, the entire motivation for our particular orientation of the stack in the underlying container.

Note that extending `LIFO` causes the `push` and `pop` methods and the `top` property to be defined.

While we want to hide the implementation from the user, it would be nice to be able to iterate through the values of the `Stack` in the order that we would expect them to appear from successive remove operations. For this reason, we construct an iterator that traverses the underlying list in reverse order:

```
def __iter__(self):
    """An iterator over the Stack."""
    return reversed(self._data)
```

The `reversed` iterator is an `itertool` that simply takes an underlying iterator (here, the one provided by the `list self._data`) and reverses the order that the elements appear. This requires (possibly) substantial storage to maintain the intermediate, in-order list. A more efficient technique might write a `Stack`-specific generator that directly yields the values of the underlying list in reverse order. We leave that as an exercise for the reader.

Now that we have defined an iterator for the stack we can appreciate the general implemenation of the special `__str__` method provided by the `Linear` interface. It simply returns a string that represents a list of values in iteration order:

```
def __str__(self):
    """Return a string representation of this linear."""
    return str(list(self));
```

The special `__repr__` method, recall, depends directly on the meaningful representation of `__str__`. We must always be observant that the special methods `__init__`, `__iter__`, `__str__`, and `__repr__` work consistently, especially when they are defined across many related classes.

6.3 The Queue

The *queue* is a linear structure that always returns that value that has been resident in the structure the longest. Of all the elements in the structure, that element was the *first* added, thus we call queue-like structure first-in, first-out, or

FIFO, structures. Queues are typically used to solve problems in the order that they were first encountered. “To do” lists and agenda-driven algorithms typically have some form of queue as the fundamental control mechanism. When removed values are reinserted (as, obviously, the least recently added item), the structure provides a *round robin* organization. Thus, queues are commonly found in systems that periodically provide exclusive access to shared resources. Operating systems often turn over computational resources to processes in approximate round-robin order.

The ordering concept of FIFO structures is so important we provide a base class implementation of three alias commonly used with queue-like structures: the `enqueue` and `dequeue` methods, and the `next` property.

```
@checkdoc
class FIFO(metaclass=abc.ABCMeta):
    """
    A base class for queue-like objects.
    """

    def enqueue(self, value):
        """Insert value into data structure."""
        self.add(value)

    def dequeue(self):
        """Remove value from data structure."""
        return self.remove()

    def next(self):
        """Return next removed value from data structure."""
        return self.peek()
```

Again, we use these methods when we wish to emphasize the importance of FIFO ordering of values stored within in the linear structure.

Paralleling our construction of the `Stack` class, the `Queue` class is based on the `UnboundedLinear` and `FIFO` classes. Between these two base classes, most of the `Queue` methods have been implemented.

```
class Queue(UnboundedLinear,FIFO):

    __slots__ = ["_data", "_frozen", "_hash"]

    def __init__(self, data=None, frozen=False, container=CircularList):
        """Construct a Queue from an iterable source."""
        super().__init__(container=container)
        if data:
            for v in data:
                self.enqueue(v)
        self._frozen = frozen
```

In this implementation, we have chosen to use the `CircularList` implementation, which provided efficient access for adding to the tail and removing from the head of a list. When queues are long, the `list` class may induce significant penalties for removing from the head of the structure. If we know the `Queue` will hold, ever, only a few elements, we can always pass a `container=list` parameter when we initialize the `Queue`. These are the subtle choices that go beyond simply meeting correctness requirements.

What remains, recall, is the `remove` method, which determines the first-in, first-out policy. Elements are added at the tail of the list, and are removed from the head. Since elements only appear at the head after all longer-resident values have been removed, the policy is FIFO. The `peek` method allows us to see this value before it is removed from the structure:

```
@mutatormethod
def remove(self):
    """Wrapper method for dequeue to comply with ABC Linear."""
    assert not self.empty
    return self._data.pop(0);

def peek(self):
    """The next value that will be dequeued.  Equivalent to next."""
    assert not self.empty
    return self._data[0]
```

Here, the use of the unaesthetic `pop` operation is hidden by our implementation. We are thankful.

Because the natural order of the elements to be removed from the queue parallels the natural order of elements that appear in the underlying list, the central `__iter__` method can simply fall back on iterator for the underlying list.

```
def __iter__(self):
    """An iterator over the Queue."""
    return iter(self._data)
```

Notice, by the way, that we have to explictly ask (via `iter`) for the iterator from the list class. In the `Stack` iterator, we used `reversed`, an iterator that asked for the subordinate iterator from the `List` class on our behalf.

6.4 Bounded Implementations: `StackBuffer` and `QueueBuffer`

We had hinted, in the previous section, that we might provide a slightly different implementation if we felt that the maximum size of the `Queue` was going to be small. This type of analysis is often possible and important to the efficient implementation of computationally intensive algorithms. We might, for example, know *a priori* that we would never have more than a dozen values stored in a stack. This may be a limitation of underlying hardware[2], or it may be a lim-

[2] For example, many "smart card" technologies allow the embedding of limited computation in a low-cost portable device. It is reasonable, in these situations, to impose a limit on certain resources, like stack or queue sizes, fully disclosing them as part of the technology's API. Programming for

itation that allows us to make further efficiency-enabling assumptions "down stream" from the structure at hand. Whatever the reason, when we provide strict limits on the size of a linear structure, we often call these implementations *buffer-based*. Here, we quickly consider the `StackBuffer` and `QueueBuffer`.

Because the storage requirements of these structures is bounded, and often small, we can allocate the entire store, once, for use throughout the life of the structure. The size of the store is called its *extent*, or *allocation*, and it provides an upper bound on the logical size of the structure. Because the extent and the size are not the same, we must keep track of the size independently.

Because of the inherent speed and size efficiency of built-in lists, we opt to make the container class a `list`. In our implementation, here, this cannot be overridden. Because of our single allocation strategy, we hope to avoid movement of data within the list. This will require us, therefore, to think of the top or front of the logical structure as "moving", over time, through the underlying list. While the logical extremes of the structure move (sometimes wrapping around to the other end), it's important to realize that data within the linear does not, itself, actually get copied from one location to another.

Keeping track of the location of the top of the linear, along with its size is sufficient to represent the $n + 1$ different sized linears that can be stored in a n element `list`. If, instead, we kept track of, say, the index of the next element as well as the index of the last element, we could not directly distinguish between empty and full structures. This is worth thinking about very carefully. Keeping track of a logical size of the structure is an easy solution to the problem, but there are others. For example, we could maintain a boolean flag that is `True` precisely when the structure is empty. Adding a value to the structure sets this flag to `False`. If removing a value makes the structure appear either empty or full, it must, of course, be empty, so the flag is set to `True`. This one extra bit of information makes use of an information-theoretic argument to help convince ourselves that the structure is in state that is unambiguous.

Here, then, are the details of our `BoundedLinear` structure. In contrast to the `UnboundedLinear` we focus on the implementation of the `remove` and `peek` methods, and promise to implement `add` in specific extensions to this class.

```
class BoundedLinear(Linear):
    __slots__ = ["_data","_next","_size","_extent"]

    def __init__(self, extent):
        """Partically construct a linear."""
        super().__init__()
        assert extent>0
        self._extent = extent
        self._data = self._extent*[None]
        self._next = 0
        self._size = 0
```

these devices can be challenging, but it is often rewarding to realize how efficient computation can occur within these imposed limits.

```
        # subclasses of this type must perform further initialization

    @property
    def empty(self):
        """Data structure is empty."""
        return self._size == 0

    @property
    def capacity(self):
        """Return the size of the underlying buffer."""
        return self._extent

    @property
    def full(self):
        """Data structure cannot accept more items."""
        return self._size == self._extent

    @abc.abstractmethod
    def add(self, value):
        """Add a value to the linear structure."""
        ...

    def remove(self):
        """Remove a value from the linear strucuture."""
        if self.empty:
            raise EmtpyError()
        else:
            result = self._data[self._next]
            self._data[self._next] = None
            self._next = (self._next+1) % self._extent
            self._size -= 1
            return result

    def peek(self):
        """The next value that will be popped.  Equivalent to top."""
        if self.empty:
            raise EmptyError()
        else:
            return self._data[self._next]

    def __len__(self):
        """The number of items in the linear structure."""
        return self._size
```

By implementing the `remove` operation a this point in the class hierarchy, we are making the (arbitrary) decision that all buffer-based linear structures will be oriented in the same direction—the *head* of the structure is found at index _next and, as we move right to through higher indices, we encounter values that will be removed farther into the future. The task of remove, then, is simply to remove the value at _next, and return it. A subtle point at line 40 is the assignment of None to cell logically freed up by this operation. When an object, result, is logically removed from the linear structure, it is important to make sure that the actual reference is removed. If we don't remove it, the garbage collector in Python has no way to verify that the reference is unnecessary. If the object referred to here by result would otherwise be unreferenced in the future, a reference here would make it impossible to reclaim the space used by the object. This is not a problem in our unbounded implementations of linear structures because the dereferencing of these objects happens naturally as part of the List pop method. A failure to dereference values is not a *logical* error, it's simply an space performance optimization that, in some cases, can yield important space savings through garbage collection.

The details of StackBuffer and QueueBuffer are very similar, with their only significant differences appearing in the workings of the add method. In the StackBuffer, the addition of values appears near the _next index:

```
class StackBuffer(BoundedLinear,LIFO,Freezable):
    def __init__(self, extent=10, data=None, frozen=False):
        """Construct a Stack from an iterable source."""
        super().__init__(extent)
        # super().__init__() sets frozen false
        if data:
            for v in reversed(data):
                self.push(v)
        self._frozen = frozen

    @mutatormethod
    def add(self,value):
        if self.full:
            raise FullError()
        else:
            self._next = (self._next-1+self._extent)%self._extent;
            self._data[self._next] = value
            self._size += 1

    def __iter__(self):
        """An iterator over the Stack."""
        return (self._data[i%self._extent] for i in range(self._next,self._next+self._size))
```

Notice that we update the _next index, not by subtracting 1, but by *adding* the extent less 1. This ensures that all the intermediate values are non-negative, where the behavior of the modulus operator (%) is well-behaved.[3] Thus, the val-

[3] The concern, here, is that the definition of n%m for n<0 is different for different languages, though most strive to have the remainder computation meet the constraint that n%m == n-(n/m)*m. In this particular application, we would like to have the result be a valid index, a value that should never

ues produced by the `next` method are found at indices that increase (wrapping around to index 0, if necessary), over time. The iterator, then, must carefully return the values starting at index `_next`.

The implementation of `QueueBuffer` differs only in its `add` method, which adds the new value to the *tail* of the logical list, that is, at `_size` elements past `_next`.

```
class QueueBuffer(BoundedLinear,FIFO,Freezable):
    def __init__(self, extent=10, data=None, frozen=False):
        """Construct a Queue from an iterable source."""
        super().__init__(extent)
        # super().__init__() sets frozen false
        if data:
            for v in data:
                self.enqueue(v)
        self._frozen = frozen

    @mutatormethod
    def add(self,value):
        if self.full:
            raise FullError()
        else:
            last = (self._next+self._size)%self._extent
            self._data[last] = value
            self._size += 1

    def __iter__(self):
        """An iterator over the Queue."""
        return (self._data[i%self._extent] for i in range(self._next,self._next+self._size)
```

Here, of course, the list is made longer simply by increasing `_size`, provided that the queue is not full. There is no need to worry about the magnitude of `_size` since, if the queue was not full, `_size` must have been less than `_extent`.

Because we have oriented the buffer-based stack and queue the same way (from the perspective of the `remove` operation), the iterators are precisely the same. It is reasonable, at this point, to consider whether the `__iter__` method should be pushed back up into the super-class, `BoundedLinear`, where it might serve to motivate a similar orientation in other subclasses of `BoundedLinear`.

be negative. In Python, the sign of the result is always the same as the sign of the divisor (`m`, above). In Java, the sign of the result is always the same as the dividend (`n`). In C the result is defined by the underlying hardware. From our point of view, Python computes the correct result, but the answer is never incorrect if we take the extra step described in the text, to guarantee the dividend is always non-negative.

6.5 The Subtleties of Iteration and Equality Testing

We saw, in our previous discussion of linear structures, that traversal of linear structures is best motivated by the order of values that are encountered when successive `remove` operations are performed. In fact, since the underlying structure is hidden, it is important that the traversal order be motivated by the *semantics* of how the structure acts over time, as opposed to the *mechanics* of how the structure is implemented.

Another important reason to carefully think about the ordering of traversals is that they often provide the mechanism necessary to help us implement higher-level operations like equality testing. In Python, recall, two objects are compared using the `eq` method. For simple, non-collection objects, two objects are typically equal if, say, all of their components are, themselves, equal. For collection objects, testing is more complex. Suppose, for example, that we compare two buffer-based queues. It is important *not* to implement equality testing by simply handing the operation of to the underlying `list` object because (1) the `lists` may be different lengths or (2) the logical structures may be mapped to different locations within their respective `lists`. The problem becomes more complex when we want to support the comparison of structures implemented in fundamentally different ways, for example, two FIFO structures implemented using bounded and unbounded linear structures, respectively.

The solution is to base the special method `__eq__` on parallel traversals of the two structures. Now we understand our implementation of `__eq__` from the `Linear` interface:

```
def __eq__(self, other):
    """Data structure has equivalent contents in the same order as other."""
    if not isinstance(other,Linear) or len(self) != len(other):
        return False
    for x,y in zip(self, other):
        if x != y:
            return False
    return True
```

This is the structure of equality testing for most container classes.

6.6 Python's `deque` Collection

Python's collections package includes a *deque* or *double ended queue*. This builtin type provides all the features of a `list`, but efficiently implements insertion of values at low indices. Because of this, it supports new methods, `appendleft`, `extendleft`, and `popleft`, that act like `append`, `extend`, and `pop`, but that act on the low-index end of the structure, efficiently. Random access indexing is also available, but it works in time linear in the distance from the nearest list end.

The technique is, essentially, to support the structure with a sufficient number small arrays or *blocks* (typically about 64 elements in length) that are linked together to form a doubly linked list. The first element of the deque is located somewhere within the first block, and the last element is located somewhere within the last block.[4] All other elements are stored contiguously from after the first element, through successive blocks, and until the last element in the last block. The deque also quietly supports a bounded version of the structure by allowing the user to specify a maximum length associated with the structure. Appending values to either end causes excess elements to be lost off the other.

Our version of deque maintains several state variables, including _list, the underlying doubly linked list; _first and _len, that determine the span of the deque across the blocks of store; the _maxlen variable is either None if the deque is unbounded, or the maximum length of the buffer. Here's how our implementation of the deque class begins:

```
from collections import Iterable
from structure.doubly_linked_list import DoublyLinkedList
from structure.decorator import checkdoc, hashmethod, mutatormethod
# size of the blocks of contiguous elements.

@checkdoc
class deque(Iterable):
    blocksize = 64

    __slots__ = [ "_list", "_first", "_len", "_maxlen" ]

    def __init__(self, initial = None, maxlen=None):
        """Initialize deque instance."""
        self._maxlen = maxlen
        self._list = DoublyLinkedList()
        self._list.append(deque.blocksize*[None])
        self._first = deque.blocksize//2
        self._len = 0
        if initial is not None:
            for v in initial:
                self.append(v)
```

The DoublyLinkedList maintains a collection of blocks of store, each of which is a fixed length list of size blocksize (here, 64). These structures never change size, they simply provide potential storage locations.The length of the structure, _len, is zeroed, and we choose a value of _first that will allow append operations on either end of the structure to expand into adjacent cells of the current block.

The location of the last element of the deque is determined by the private property, _last:

```
@property
def _last(self):
```

[4] For simplicity, we assume there is always at least one block backing the deque.

```
        return self._first + self._len - 1
```

This property computes the index of the last elements, assuming that it is found somewhere within the first block. Clearly, if `_len` becomes large, then this is not a valid index. With a little ingenuity, we can compute the block index (the index of the element in `_list`) and the offset within that block. The private method `_addr` takes one of these "virtual" block offsets and returns a 2-tuple that contains the actual block index and offset:

```
    def _addr(self,offset):
        """Return the block and line pair associated with an offset."""
        return (offset//deque.blocksize,offset%deque.blocksize)
```

A typical use of these methods is the implementation of `__getitem__` and `__setitem__` that allow random access and update of elements within the `deque`:

```
    def __getitem__(self,index):
        """Get a value stored at a position in the deque indicated by index."""
        if index >= len(self):
            raise IndexError("Index {} out of range.".format(index))
        (block,line) = self._addr(self._first + index)
        return self._list[block][line]

    def __setitem__(self,index,value):
        """Assign the element of the deque at index to value."""
        if index > len(self):
            raise IndexError("Index {} out of range.".format(index))
        (block,line) = self._addr(self._first + index)
        self._list[block][line] = value
```

The reader may find it useful to walk through these methods with `index` set to zero: it simply accesses the `_first` entry of block 0 of the `_list`. It is important to observe, here, that the performance of this routine is $O(1)$ and optimized to search for the appropriate block from the nearest end of the structure (see page **??**).

We now consider the operations that increase the size of the structure: `append` and `appendleft`. The append operation checks to see if the last block in the `_list` is full, potentially extends the list, and then writes the desired value at the block and index indicated by the new value of `_last`:

```
    def append(self,value):
        """Append value to the far end of the deque."""
        (block,line) = self._addr(self._last+1)
        if block >= len(self._list):
            self._list.append(deque.blocksize*[None])
        self._len += 1
        self._list[block][line] = value
        if (self._maxlen is not None) and (self._len > self._maxlen):
            self.popleft()
```

Here, we note that increasing the size of the list simply requires incrementing `_len`; `_first` stays where it is and `_last` is a property whose value depends on _first and _size. The appendleft operation is symmetric, save for the need to update _first:

```
def appendleft(self,value):
    """Append value to the near end of the list."""
    self._first = (self._first-1)%deque.blocksize
    if self._first == deque.blocksize-1:
        self._list.insert(0,deque.blocksize*[None])
    self._len += 1
    self._list[0][self._first] = value
    if (self._maxlen is not None) and (self._len > self._maxlen):
        self.pop()
```

An important part of both of these methods is to enforce any buffer limits that may have been imposed at the construction of the deque.

All other growth operations are cast in terms of one of the append methods. For example, extend simply (and efficiently) extends the list through multiple appends:

```
def extend(self,iterable):
    """Add elements from iterable to the far end of the deque."""
    for v in iterable:
        self.append(v)
```

The extendleft method is the same, but performs an appendleft, instead, reversing the order of the elements placed at the beginning of the deque.

There are no surprises in the pop and popleft methods. The reader is encouraged to sketch out and verify possible implementations.

A single rotate method allows the elements of the deque to shifted, end-around, to the right (for positive n) or left (for negative n). With a little thought it becomes clear that these rotates are related and that smaller left (or right) rotates are preferred over larger right (or left) rotates:

```
def rotate(self, n=1):
    """Pop elements from one end of list and append on opposite.

    If n > 0, this method removes n elements from far end of list and
    successively appends them on the near end; a rotate right of n.
    If n < 0, the opposite happens: a rotate left of n.

    Args:
       n: number of elements to rotate deque to the right.
    """
    l = len(self)
    l2 = l//2
    n = ((n + l2)%l)-l2
    while n < 0:
        self.append(self.popleft())
        n += 1
    while n > 0:
        self.appendleft(self.pop())
```

```
            n -= 1
```

While not a feature of Python's deque objects, insertion and deletion of values at arbitrary locations can be accomplished by combinations of rotations, appends, and pops.

Finally, because deque structures support `__len__` and `__getitem__` methods, Python constructs a default iterator. Unfortunately, `__getitem__` is a linear operation, so a traversal of the structure with the default iterator leads to $O(n^2)$ performance in time. This can be improved if we use a generator to manage a long-running traversal over the underlying doubly linked list. One implementation is as follows:

```
def __iter__(self):
    """Traverse the elements of the deque from head to tail."""
    lit = iter(self._list)
    offset = self._first
    block = None
    while offset <= self._last:
        index = offset%deque.blocksize
        if block is None:
            block = next(lit)
        yield block[index]
        offset += 1
        if index == deque.blocksize-1:
            block = None
```

This generator is quite efficient, but it is not as pretty as one might expect, largely because care has to be taken when `_first` or `_last` fall at one of the ends of an underlying block.

Python's deque, which is implemented in a manner similar to this, provides a general alternative to linear structure implementation. Because of a versatility that provides many features that we may not want accessible in our linear structures, it may allow users to violate desired FIFO or LIFO behaviors we desire. It remains, however, a suitable container atop which we may implement more restrictive interfaces that support only FIFO or LIFO operations. Because, for example, the deque does not provide a general insert method, it cannot be classified as a list. Such a method would be easy to construct, provided we think carefully about *which* element should be removed if the insert grows the deque beyond its desired maximum length.

6.7 Conclusions

Often, one of the most important features of a container class is to *take away* features of an underlying container. For linear structures, we seek to guarantee certain order relationships between the values inserted into the linear structure and those extracted from it. In our examples and, as we shall continue to see in later chapters, these relationships are important to making sure that algorithms achieve a desired performance or, even more critically, compute the correct result.

Chapter 7

Sorting

Computers spend a considerable amount of their time keeping data in order. When we view a directory or folder, the items are sorted by name or type or modification date. When we search the Web, the results are returned sorted by "applicability." At the end of the month, our checks come back from the bank sorted by number, and our deposits are sorted by date. Clearly, in the grand scheme of things, sorting is an important function of computers. Not surprisingly, data structures can play a significant role in making sorts run quickly. This chapter begins an investigation of sorting methods.

7.1 Approaching the Problem

For the moment we assume that we will be sorting an unordered list of integers (see Figure 7.1a).[1] The problem is to arrange the integers so that every adjacent pair of values is in the correct order (see Figure 7.1b). A simple technique to sort the list is to pass through the list from left to right, swapping adjacent values that are out of order (see Figure 7.2). The exchange of values is accomplished with a parallel assignment statement:

```
(a,b) = (b,a)
```

This is equivelent to the packing and unpacking of a 2-tuple, or a sequential series of statements:

```
temp = a
a = b
b = temp
```

but the temporary variable (`temp`) is implied.

After a single pass the largest value will end up "bubbling" up to the high-indexed side of the list. The next pass will bubble up the next largest value, and so forth. The sort—called *bubble sort*—must be finished after $n-1$ passes. Here is how we might write bubble sort in Python:

```
def bubble(l,n):
    for sorted in range(0,n):
        for i in range(1,n-sorted):
            if l[i] < l[i-1]:
```

[1] We focus on list of integers to maintain a simple approach. These techniques, of course, can be applied to other container classes, provided that some relative comparison can be made between two elements. This is discussed in Section 7.9.

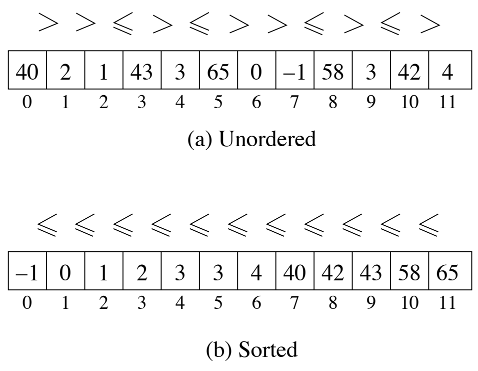


Figure 7.1 The relations between entries in unordered and sorted lists of integers.

```
(l[i-1],l[i]) = (l[i],l[i-1])
```

Observe that the only potentially time-consuming operations that occur in this sort are comparisons (on line 4) and exchanges (on line 5). While the cost of comparing integers is relatively small, if each element of the list were to contain a long string (for example, a DNA sequence) or a complex object (for example, a Library of Congress entry), then the comparison of two values might be a computationally intensive operation. Similarly, the cost of performing an exchange is to be avoided.[2] We can, therefore, restrict our attention to the number of comparison and exchange or data movement operations that occur in sorts in order to adequately evaluate their performance.

In bubble sort each pass of the bubbling phase performs $n-1$ comparisons and as many as $n-1$ exchanges. Thus the worst-case cost of performing bubble sort is $O(n^2)$ operations. In the best case, none of the comparisons leads to an exchange. Even then, though, the algorithm has quadratic behavior.[3]

Most of us are inefficient sorters. Anyone having to sort a deck of cards or a stack of checks is familiar with the feeling that *there must be a better way to do this*. As we shall see, there probably is: most common sorting techniques used in day-to-day life run in $O(n^2)$ time, whereas the best single processor comparison-based sorting techniques are expected to run in only $O(n \log n)$ time. (If multiple processors are used, we can reduce this to $O(\log n)$ time, but that algorithm is beyond the scope of this text.) We shall investigate some

[2] In languages like Python, where objects are manipulated through references, the cost of an exchange of even large objects is usually fairly trivial. In some languages, however, the cost of exchanging large values stored directly in the list is a real concern.

[3] If, as we noted in Figure 7.2, we stopped when we performed a pass with no exchanges, bubble sort would run in $O(n)$ time on data that were already sorted. Still, the average case would be quadratic.

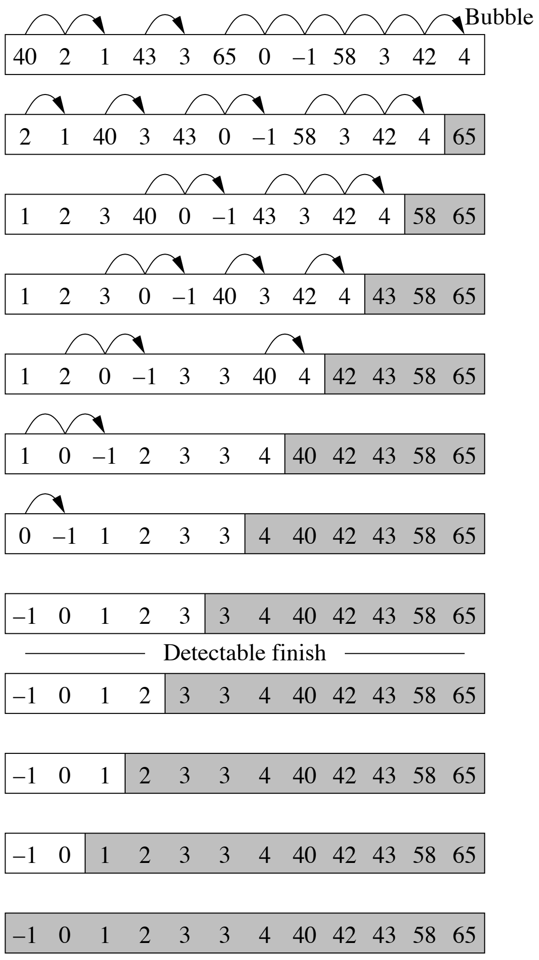


Figure 7.2 The passes of bubble sort: hops indicate “bubbling up” of large values. Shaded values are in sorted order. A pass with no exchanges indicates sorted data.

other sorting techniques that run in $O(n^2)$ time, on average, and some that run in $O(n \log n)$ time. In the end we will attempt to understand what makes the successful sorts successful.

Our first two sorting techniques are based on natural analogies.

7.2 Selection Sort

Children are perhaps the greatest advocates of *selection sort*. Every October, Halloween candies are consumed from best to worst. Whether daily sampling is limited or not, it is clear that choices of the next treat consumed are based on "the next biggest piece" or "the next-most favorite," and so on. Children consume treats in decreasing order of acceptability. Similarly, when we select plants from a greenhouse, check produce in the store, or pick strawberries from the farm we seek the best items first.

This selection process can be applied to an list of integers. Our goal is to identify the index of the largest element of the list. We begin by *assuming* that the first element is the largest, and then form a competition among all the remaining values. As we come across larger values, we update the index of the current maximum value. In the end, the index must point to the largest value. This code is idiomatic, so we isolate it here:

```
max = 0 # assume element 0 is the largest
for loc in range(1,unsorted):
    if l[max] < l[loc]:
        # bigger value found
        max = loc
# l[max] is largest value
```

(Notice that the maximum is not updated unless a *larger* value is found.) Now, consider where this maximum value would be found if the data were sorted: it should be clear to the right, in the highest indexed location. All we need to do is swap the last element of the unordered elements of the list with the maximum. Once this swap is completed, we know that at least that one value is in the correct location—the one on the right—and we logically reduce the size of the problem—the number of unsorted values—by one. If we correctly place each of the $n - 1$ largest values in successive passes (see Figure 7.3), we have selection sort. Here is how the entire method appears in Python:

```
def selection(l,n):
    for sorted in range(n):
        unsorted = n-sorted
        max = 0 # assume element 0 is the largest
        for loc in range(1,unsorted):
            if l[max] < l[loc]:
                # bigger value found
                max = loc
        # l[max] is largest value
        target = unsorted-1 # where to stick l[max]
        if max != target:
            (l[max],l[target]) = (l[target],l[max])
```

We can think of selection sort as an optimized bubble sort, that simply makes one exchange—the one that moves the maximum unsorted value to its final location. Bubble sort performs as many as $n-1$ exchanges on each pas. Like bubble sort, however, selection sort's performance is dominated by $O(n^2)$ time for comparisons.

The performance of selection sort is independent of the order of the data: if the data are already sorted, it takes selection sort just as long to sort as if the data were unsorted. For this reason, selection sort is only an interesting thought experiment; many other sorts perform better on all inputs. We now think about sorting with a slightly different analogy.

7.3 Insertion Sort

Card players, when collecting a hand, often consider cards one at a time, inserting each into its sorted location. If we consider the "hand" to be the sorted portion of the list, and the "table" to be the unsorted portion, we develop a new sorting technique called *insertion sort*.

In the following Python implementation of insertion sort, the sorted values are kept in the low end of the list, and the unsorted values are found at the high end (see Figure 7.4). The algorithm consists of several "passes" of inserting the lowest-indexed unsorted value into the list of sorted values. Once this is done, of course, the list of sorted values increases by one. This process continues until each of the unsorted values has been incorporated into the sorted portion of the list. Here is the code:

```
def insertion(l,n):
    for i in range(1,n): # 1..n-1
        item = l[i]
        loc = None
        for pos in reversed(range(i)):
            if item < l[pos]:
                loc = pos
                l[pos+1] = l[pos]
            else:
                break
        if loc is not None:
            l[loc] = item

def _is(l,low,high):
    for i in range(low+1,high+1): # 1..n-1
        item = l[i]
        loc = None
```

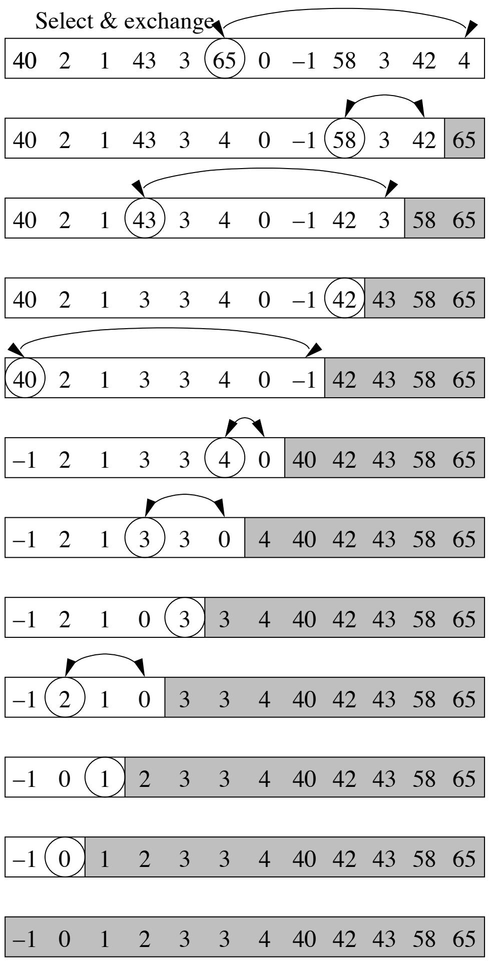


Figure 7.3 Profile of the passes of selection sort: shaded values are sorted. Circled values are maximum among unsorted values and are moved to the low end of sorted values on each pass.

```
        for pos in reversed(range(low,i)):
            if item < l[pos]:
                loc = pos
                l[pos+1] = l[pos]
            else:
                break
        if loc is not None:
            l[loc] = item
```

A total of $n-1$ passes are made over the list, with a new unsorted value inserted each time. The value inserted is likely to be neither a new minimum or maximum value. Indeed, if the list was initially unordered, the value will, on average, end up near the middle of the previously sorted values, causing the inner loop to terminate early. On random data the running time of insertion sort is expected to be dominated by $O(n^2)$ compares and data movements (most of the compares will lead to the movement of a data value).

If the list is initially in order, one compare is needed at every pass to verify that the value is as big as all the previously sorted value. Thus, the inner loop is executed exactly once for each of $n-1$ passes. The best-case running time performance of the sort is therefore dominated by $O(n)$ comparisons (there are no movements of data within the list). Because of this characteristic, insertion sort is often used when data are very nearly ordered (imagine the cost of adding a small number of new cards to an already sorted poker hand).

In contrast, if the list was previously in reverse order, the value must be compared with *every* sorted value to find the correct location. As the comparisons are made, the larger values are moved to the right to make room for the new value. The result is that each of $O(n^2)$ compares leads to a data movement, and the worst-case running time of the algorithm is $O(n^2)$.

7.4 Quicksort

Since the process of sorting numbers consists of moving each value to its ultimate location in the sorted list, we might make some progress toward a solution if we could move *a single value* to its ultimate location. This idea forms the basis of a fast sorting technique called *quicksort*.

One way to find the correct location of, say, the leftmost value—called a *pivot*—in an unsorted list is to rearrange the values so that all the smaller values appear to the left of the pivot, and all the larger values appear to the right. One method of partitioning the data is shown here. It returns the final location for what was originally the leftmost value:

```
def _partition(l,left,right):
    while left < right:
        while left < right and l[left] < l[right]:
            right -= 1
        if left != right:
```

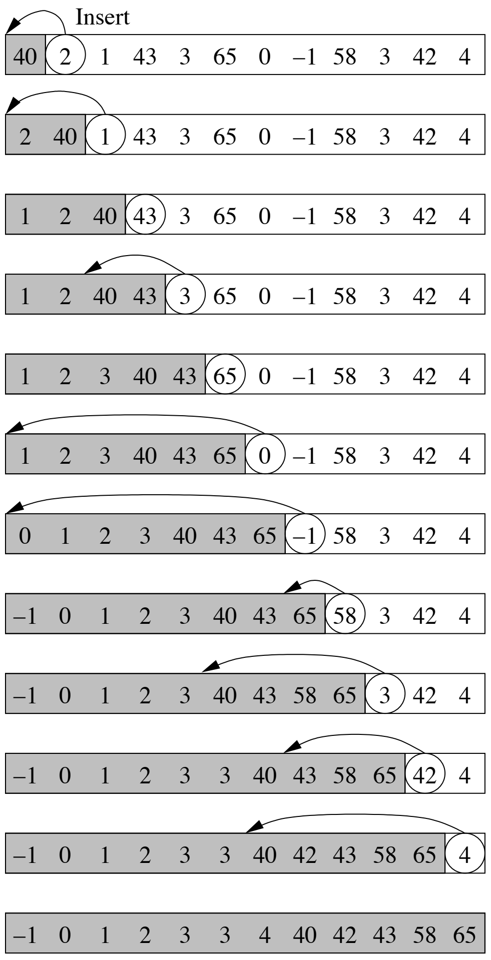


Figure 7.4 Profile of the passes of insertion sort: shaded values form a "hand" of sorted values. Circled values are successively inserted into the hand.

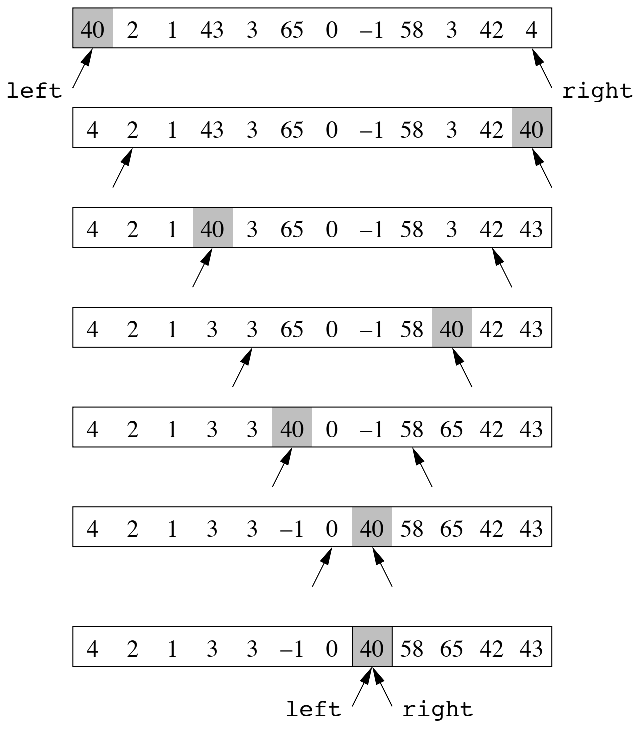


Figure 7.5 The partitioning of a list's values based on the (shaded) pivot value 40. Snapshots depict the state of the data after the `if` statements of the `partition` method.

```
                (l[left],l[right]) = (l[right],l[left])
                left += 1
            while left < right and l[left] < l[right]:
                left += 1
            if left != right:
                (l[left],l[right]) = (l[right],l[left])
                right -= 1
        return left
```

The indices `left` and `right` start at the two ends of the list (see Figure 7.5) and move toward each other until they coincide. The pivot value, being leftmost in the list, is indexed by `left`. Everything to the left of `left` is smaller than the pivot, while everything to the right of `right` is larger. Each step of the main loop compares the left and right values and, if they're out of order, exchanges them. Every time an exchange occurs the index (`left` or `right`) that references the pivot value is alternated. In any case, the nonpivot variable is moved toward

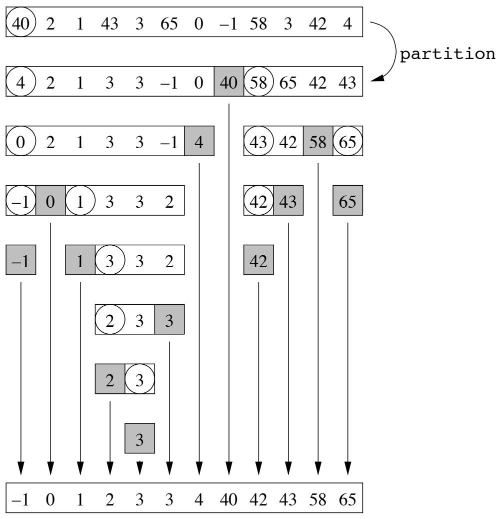


Figure 7.6 Profile of quicksort: leftmost value (the circled *pivot*) is used to position value in final location (indicated by shaded) and partition list into relatively smaller and larger values. Recursive application of partitioning leads to quicksort.

the other. (Failing to do this simply introduces redundant comparisons.) Since, at each step, `left` and `right` move one step closer to each other, within n steps, `left` and `right` are equal, and they point to the current location of the pivot value. Since only smaller values are to the left of the pivot, and larger values are to the right, the pivot must be located in its final location. Values correctly located are shaded in Figure 7.6.

Because the pivot segregates the larger and smaller values, we know that none of these values will appear on the opposite side of the pivot in the final arrangement. This suggests that we can reduce the sorting of a problem of size n to two problems of size approximately $\frac{n}{2}$. To finish the sort, we need only recursively sort the values to the left and right of the pivot:

```
def quicksort(l,n):
    _qs(l,0,n-1)

def _qs(l,low,high):
    if low < high:
        i = _partition(l,low,high)
        _qs(l,low,i-1)
        _qs(l,i+1,high)
```

In practice, of course, the splitting of the values is not always optimal (see the placement of the value 4 in Figure 7.6), but a careful analysis suggests that even with these "tough breaks" quicksort takes only $O(n \log n)$ time.

When either sorted or reverse-sorted data are to be sorted by quicksort, the results are disappointing. This is because the pivot value selected (here, the leftmost value) finds its ultimate location at one end of the list or the other. This reduces the sort of n values to $n - 1$ values (and *not* $n/2$), and the sort requires $O(n)$ passes of an $O(n)$ step partition. The result is an $O(n^2)$ sort. Since nearly sorted data are fairly common, this result is to be avoided.

Notice that picking the leftmost value is not special. If, instead, we attempt to find the correct location for the middle value, then other arrangements of data will cause the degenerate behavior. In short, for any specific or *deterministic* partitioning technique, a degenerate arrangement exists. The key to more consistent performance, then, is a *nondeterministic* partitioning that correctly places a value selected at random (see Problem **??**). There is, of course, a very unlikely chance that the data are in order *and* the positions selected induce a degenerate behavior, but that chance is small and successive runs of the sorting algorithm on the same data are exceedingly unlikely to exhibit the same behavior. So, although the worst-case behavior is still $O(n^2)$, its expected behavior is $O(n \log n)$.

Improving Quicksort

While the natural implementation strategy for quicksort is recursive, it's important to realize that Python has a limited number of method calls that can be outstanding at one time. A typical value for this limit is 1000. That means, for example, sorting a list of 1024 values that is *already sorted* not only incurs a performance penalty, it will actually *fail* when the recursive calls cascade beyond 1000 deep. We consider how this limit can be avoided.

When data is randomly ordered, there can be as many as $O(\log n)$ outstanding calls to quicksort. The limit of 1000 levels of recursion is not a problem in cases where the partitioning of the list is fairly even. However, quicksort can have degenerate behavior when the correct location for the selected pivot is at one end of the list or the other. The reason is that the larger half of the partitioned list is nearly as large as the list itself. This can lead to $O(n)$ levels of recursion.

There are two ways that this can be dealt with. First, we can choose, at each partitioning, a *random* element as the pivot. Our partitioning strategy is easily adapted to this approach: we simply begin the partition by swapping the left element with some other element picked at random. In this way, the selection of extreme values at each partitioning stage is quite unlikely. It is also possible to simply shuffle the data at the beginning of the sort, with similar expected runtime behavior. Randomization leads to expected $O(n \log n)$ running time and $O(\log n)$ stack depth.

A second optimization is to note that once the partition has been completed, the remaining tasks in quicksort are recursive calls. When a procedure ends with a recursive call it is called *tail recursion*, which can be converted into a simple call-free loop. Quicksort ends with two calls: one can be converted into a loop, and the other cannot be eliminated in any direct way. Thus the following procedure is equivalent to our previous formulation:

```
def _qs(l,low,high):
    while low < high:
        i = _partition(l,low,high)
        _qs(l,low,i-1)
        low = i+1
```

The order of the recursive calls in quicksort is, of course, immaterial. If we choose to convert one of the recursive calls, we should choose to eliminate the call *that sorts the large half of the list*. The following code eliminates half of the calls *and* ensures the depth of the stack never grows larger than $O(\log n)$ call frames:

```
def _qs(l,low,high):
    while low < high:
        i = _partition(l,low,high)
        if i < (low+high)//2:
            # sort fewer smaller numbers recursively
            _qs(l,low,i-1)
            low = i+1
        else:
            # sort fewer larger numbers recursively
            _qs(l,i+1,high)
            high = i-1
```

Quicksort is an excellent sort when data are to be sorted with little extra space. Because the speed of partitioning depends on the random access nature of lists, quicksort is not suitable when used with structures that don't support efficient indexing. In these cases, however, other fast sorts are often possible.

7.5 Mergesort

Suppose that two friends are to sort a list of values. One approach might be to divide the deck in half. Each person then sorts one of two half-decks. The sorted deck is then easily constructed by combining the two sorted half-decks. This careful interleaving of sorted values is called a *merge*.

It is straightforward to see that a merge takes at least $O(n)$ time, because every value has to be moved into the destination deck. Still, within $n - 1$ comparisons, the merge must be finished. Since each of the $n - 1$ comparisons (and potential movements of data) takes at most constant time, the merge is no worse than linear.

There are, of course, some tricky aspects to the merge operation—for example, it is possible that all the cards in one half-deck are smaller than all the cards in the other. Still, the performance of the following merge code is $O(n)$:

```
def _merge(l,l0,h0,l1,h1,temp):
    # save left sub-array into temp
    for i in range(l0,l1):
        temp[i] = l[i]
    target = l0
    while l0 <= h0 and l1 <= h1:
        if l[l1] < temp[l0]:
            l[target] = l[l1]
            l1 += 1
        else:
            l[target] = temp[l0]
            l0 += 1
        target += 1
    while (l0 <= h0):
        l[target] = temp[l0]
        target += 1
        l0 += 1
```

This code is fairly general, but a little tricky to understand (see Figure 7.7). We assume that the data from the two lists are located in the two lists—we copy the lower half of the range into `temp` and the upper half of the range remains in `l` (see Figure 7.7a). The first loop compares the first remaining element of each list to determine which should be copied over to `l` first (Figure 7.7b). That loop continues until one of the two sublists is emptied (Figure 7.7c). If `l` is the emptied sublist, the remainder of the `temp` list is transferred (Figure 7.7d). If the `temp` list was emptied, the remainder of the `l` list is already located in the correct place—in `l`!

Returning to our two friends, we note that before the two lists are merged each of the two friends is faced with sorting half the cards. How should this be done? If a deck contains fewer than two cards, it's already sorted. Otherwise, each person could recursively hand off half of his or her respective deck (now one-fourth of the entire deck) to a new individual. Once these small sorts are finished, the quarter decks are merged, finishing the sort of the half decks, and the two half decks are merged to construct a completely sorted deck. Thus, we might consider a new sort, called *mergesort*, that recursively splits, sorts, and reconstructs, through merging, a deck of cards. The logical "phases" of mergesort are depicted in Figure 7.8.

```
def _ms(l,low,high,temp):
    if high <= low:
        return
    mid = (low+high)//2
    if low < mid:
        _ms(l,low,mid,temp)
    if mid+1 < high:
        _ms(l,mid+1,high,temp)
    _merge(l,low,mid,mid+1,high,temp)
```

Note that this sort requires a temporary list to perform the merging. This tem-

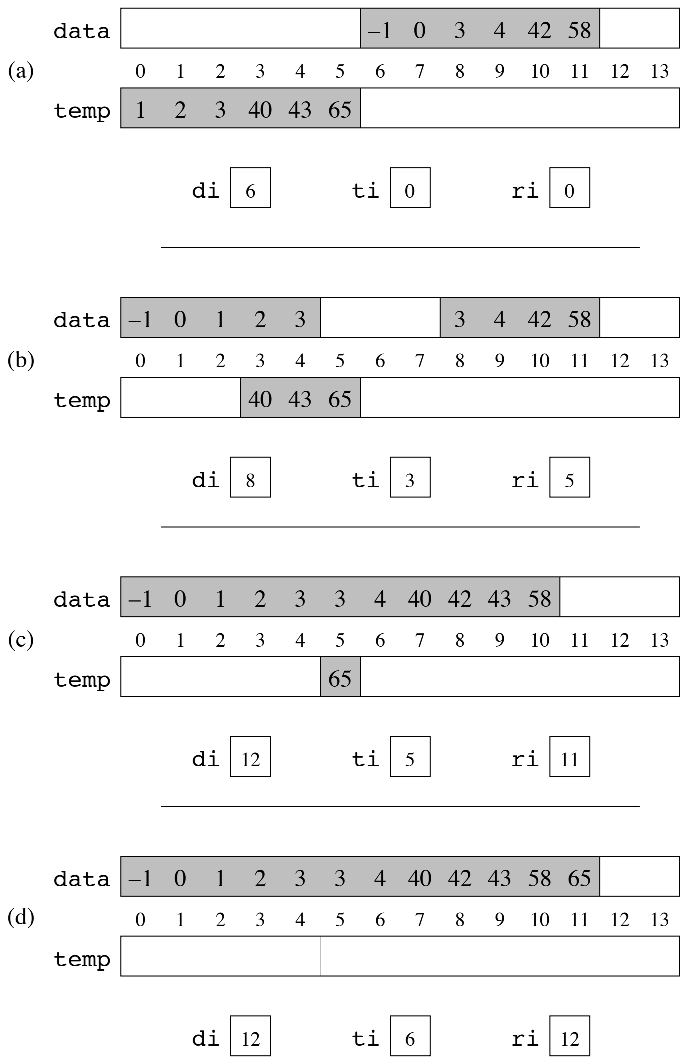


Figure 7.7 Four stages of a merge of two six element lists (shaded entries are participating values): (a) the initial location of data; (b) the merge of several values; (c) the point at which a list is emptied; and (d) the final result.

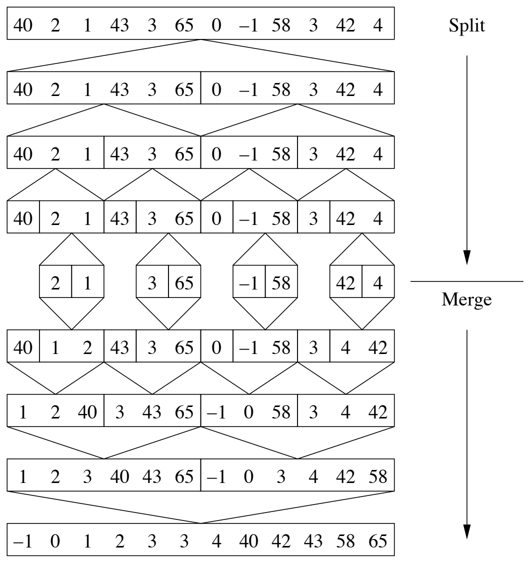


Figure 7.8 Profile of mergesort: values are recursively split into unsorted lists that are then recursively merged into ascending order.

porary list is only used by a single merge at a time, so it is allocated once and garbage collected after the sort. We hide this detail with a public wrapper procedure that allocates the list and calls the recursive sort:

```
def mergesort(l,n):
    temp = len(l)*[None]
    _ms(l,0,n-1,temp)
```

Clearly, the depth of the splitting is determined by the number of times that n can be divided in two and still have a value of 1 or greater: $\log_2 n$. At each level of splitting, every value must be merged into its respective sublist. It follows that at each logical level, there are $O(n)$ compares over all the merges. Since there are $\log_2 n$ levels, we have $O(n \cdot \log n)$ units of work in performing a mergesort.

One of the unappealing aspects of mergesort is that it is difficult to merge two lists without significant extra memory. If we could avoid the use of this extra space without significant increases in the number of comparisons or data movements, then we would have an excellent sorting technique.

Recently, there has been renewed interest in designing sorts that are combinations of the sorts we have seen so far. Typically, the approach is to select an appropriate sort based on the characteristics of the input data. In our next section, we see how this *adaptation* can be implemented.

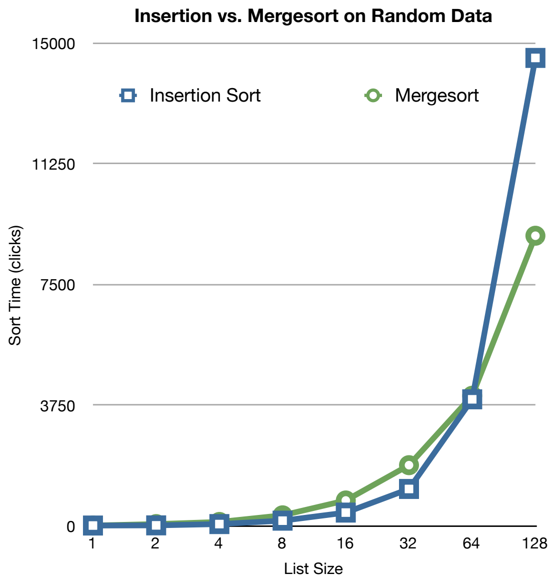


Figure 7.9 The relative performance of mergesort and insertion sort for lists of length 1 through 256. Note that mergesort is more efficient than insertion sort only for lists larger than 32 elements.

7.5.1 Adaptive Sorts

We have seen a number of different types of sorting, each with slightly different performance characteristics. We also noted, earlier, that the main concern about performance was the number of comparisons and memory movements, it is sometimes necessary to consider, more carefully, the actual time it takes to perform all parts of the algorithm. For example, mergesort, which is recursive and involves some fairly large control overhead in the `_merge`routine, is in the best case $O(n \log n)$, but the *constant* associated with the performance curve is, undoubtedly, much larger than the constant associated with insertion sort. This means, for example, that for small values of n, insertion sort may be expected to perform better than mergesort.

Since mergesort (and quicksort) naturally approach the problem through recursive sorting, we have the opportunity to substitute in a more efficient sort when the subarray becomes sufficiently short. Mergesort can, for example, call insertion sort when that technique appears to be more efficient on a subarray. To determine the appropriate list length to adapt to insertion sort, we can measure the relative performance of sorting methods. In Figure 7.9 we see the relative performance of mergesort and insertion sort for array sizes up to 128 elements.

Since insertion sort (on random data) appears to be more efficient than merge sort when a list has 64 or fewer elements, we can modify our mergesort algorithm to take advantage of that fact:

```
def adaptivemergesort(l,n):
    temp = n*[None]
    _ams(l,0,n-1,temp)

def _ams(l,low,high,temp):
    n = high-low+1
    if n < 64:
        _is(l,low,high)
    else:
        if high <= low:
            return
        mid = (low+high)//2
        if low < mid:
            _ams(l,low,mid,temp)
        if mid+1 < high:
            _ams(l,mid+1,high,temp)
        _merge(l,low,mid,mid+1,high,temp)
```

Clearly, mergesort is really insertion sort for smaller list sizes, but for larger lists the mergesort call tree is trimmed of 6 levels of recursive calls in favor of the relatively fast insertion sort technique. In Figure **??**, the adaptive merge sort takes on the characteristics of insertion sort for lists of length 64 or less, and the shape of the curve is determined primarily by the characteristics of mergesort for larger lists.

Quicksort can benefit from the same type of analysis and we find that, similarly, for lists of length 64 or less, insertion sort is preferred. To introduce the optimization we note that partitioning the values into sublists that are relatively small and large compared to the pivot, *we know the values in the sublist must occupy that space in the sorted list*. We could sort small sublists immediately, with insertion sort, or we can delay the sorting of small sublists until after the quicksort has otherwise completed its activity. A single insertion sort on the entire list will be fairly efficient: each element is *close* to its final destination (and thus quickly slid into place with an insertion pass) and elements that were pivots are in *precisely* the correct location (so insertion sort will not move them at all). Our optimized adaptive quicksort appears, then, as follows:

```
def quicksorta(l,n):
    shuffle(l,n)
    _qsa(l,0,n-1)
    insertion(l,0,n-1)

def _qsa(l,low,high):
    while low+64 < high:
        i = _partition(l,low,high)
        if i < (high+low)//2:
```

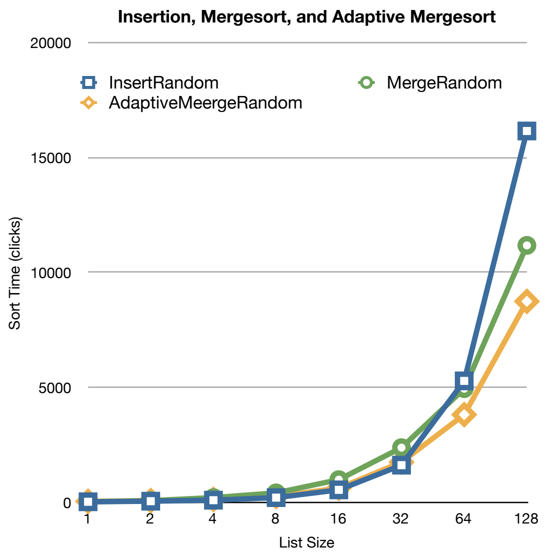


Figure 7.10 The relative performance of insertion, merge, and adaptive merge sorts.

```
        _qsa(l,low,i-1)
        low = i+1
    else:
        _qsa(l,i+1,high)
        high = i-1
```

7.6 Stability

When the objects to be sorted are complex, we frequently determine the ordering of the data based on a *sort key*. Sort keys either appear directly in the object, or are derived from data found within the object. In either case, the keys for two objects may be equal even if the objects are not. For example, we may perform a sort of medical records based on the billing id, a value that may be shared among many individuals in a family. While the billing id for family members is the same, their medical histories are not.

Of course, some objects may have an option of several different sort keys. Voters, for example, may be sorted by surname, or by the street or precinct where they live. Having the choice of keys allows us to combine or *pipeline* two or more sorts on different keys. The result is a full sort that distinguishes individuals in useful ways. In order for combined sorts to be effective, it is important that each sort be *stable*, that is, that when sort keys are equal for several data values in a list, their *relative ordering* is preserved. If, for example, we sort a voter registry first by name and then, using a stable sort, by precinct, we can be certain that all individuals in Precinct 1 appear in alphabetical order.[4]

To design a stable sort, we need to make sure that the sorting algorithm does not have the potential of exchanging two equal items. In the simple comparison sorts, like bubble and insertion sort, this is accomplished by only swapping two adjacent values when a "less than" comparison indicates they are out of order; when keys are equal, they cannot be swapped. In selection sort, we must make sure that when we select a largest value that it is the *rightmost* maximum value found. In mergesort, reordering can only happen during a merge operation. If we assume the two halves of the list were sorted in a stable manner, the only way two equal values could exchange position is if a value from the "right" list could be placed before an equal counterpart from the "left" list. Again, a careful analysis of our merging operation demonstrates that when equal values could be compared (merge only ever compares values from the two lists), the value from the left list is merged-in first.

In our version of quicksort, the ordering of values is accomplished by the `_partition` function. This function finds the "correct" location for the pivot by a swapping the pivot with a smaller or larger value. This swapping, however, causes the pivot to jump over equal values, causing them to be shuffled compared to their original ordering. Quicksort, in this implementation, is not a stable sort. Partitioning in a stable way will require the use of more space or time.

Fortunately, an unstable sort can always be turned into a stable sort, through a simple transformation of the input data. For each value v_i at the i^{th} position in the input list, we construct a pair, (v_i, i) to be sorted in its place. When the sort is finished, the list element at j is replaced with the first element of the pair found at location j.

```
for i in range(len(l)):
    l[i] = (l[i],i)
    sort(l)
    for j in range(len(l)):
        l[j] = l[j][0]
```

This approach is an instance of the idiom commonly referred to as the *decorate-sort-undecorate* or *DSU*. Why does this work? First, it's important to observe that tuples (and in this particular case, a pair) are ordered in "dictionary order": the relative ordering of two pairs is determined by the relative ordering of their first components. If these values are equal, the comparison falls through to the

[4] Indeed, before the advent of computer-based sorting, names were sorted using punched cards, with several passes of a stable mechanical sort based on the character punched in a particular column. A stable sort was made on each column, from *right to left*, leaving the punch cards in alphabetical order at the end of all passes.

next component of the tuple. This continues until an ordering is determined. If no ordering is determined through this process, then the tuples are determined to be equal. Now, if all the original values of the list are distinguishable by value (i.e. no pair of the values are equal), then this wrapping process produces pairs that can be distinguished, just by looking at their first component. In particular, this wrapping process has no ill-effect on the sort. The other extreme, where all the values are the same, produces pairs that are all distinguishable *by their second component*, and that ordering represents the original order that they appeared within the array. Whenever two original values are equal, their order is determined by their original index order.

The overhead of decorating and undecorating the data is a relatively simple $O(n)$ operation in time (and space) that is unlikely to have a significant impact on the performance of most sorting techniques. Still, when stable sorts can be promised, or where stability is not important, the decoration-based paradigm is unnecessary.

7.7 Key Functions: Supporting Multiple Sort Keys

When data items support multiple sort keys, we must have a mechanism that allows us to tell a sorting method *how* the key is found for each data item. The approach that Python uses is to provide a *key function*. This is a general function that, given a data element, returns the key associated with that element. This function should be *deterministic*: the key function should return the same key each time the data value is provided.

One obvious advantage of the use of key functions is that the choice *how data is ordered* is independent of *how data is encoded*. When we design a structure to hold information about baseball players, we needn't worry about whether they primary way they will be ordered is by their name or their batting average. Instead, we can provide a variety of key functions that generate the key required. Notice that the key function need not *directly* find the data, but may actually *compute*, say, a baseball statistic on-the-fly.

Provided the key function, it is tempting to modify one's sorts so that instead of comparing data elements, we compare the values returned by the key function:

```
if keyFun(a) < keyFun(b):
    ...
```

but this means that the key functions are called many times. In the best case of comparison-based sorts, this is *at least* 1 time per value, but could be as many as $O(n)$. When the complexity of the key function is great, we would like to place firm limits on this overhead.

One approach is to use the decorate-sort-undecorate paradigm to generate a list of key-value pairs. The `KV` item stored at location `i` has a `value` of `l[i]` and a `key` that is the key associated with i^{th} original element. The decoration is accomplished as follows:

```
    for i in range(len(l)):
        l[i] = KV(keyfun(l[i]),l[i])
```

This list can then be sorted independently of the original list since KV values are compared based just on their keys. Once the sort is finished, the list is undecorated with:

```
    for i in range(len(l)):
        l[i] = kvlist[i].value
```

Notice that these decorate operations are *not* list comprehension since the list is to be sorted *in place*. As with the process of stabilizing a sort, we could have used tuples instead of key-value pairs, but because the key function may return equal key values for list elements that are not equal, it is important to make sure that comparisons never fall through to the component that carries the original value. The correct approach, then, is a decoration strategy that used 3-tuples:

```
    for i in range(len(l)):
        l[i] = (keyfun(l[i]), i, l[i])
```

Because equal key values reduce to the comparison of original list index values, which are necessarily distinguishable, the 3-tuples are clearly compared without accessing the original value, `l[i]`. A pleasant side-effect of this time-saving strategy is the introduction of stability in any otherwise unstable sort. The undecorate appears as

```
    for i in range(len(l)):
        l[i] = l[i][2]
```

7.8 Radix and Bucket Sorts

After investigating a number of algorithms that sort in $O(n^2)$ or $O(n \log n)$ time, one might wonder if it is possible to sort in linear time. If the right conditions hold, we can sort certain types of data in linear time. First, we must investigate a pseudogame, 52 pickup!

Suppose we drop a deck of 52 cards on the floor, and we want to not only pick them up, but we wish to sort them at the same time. It might be most natural to use an insertion sort: we keep a pile of sorted cards and, as we pick up new cards, we insert them in the deck in the appropriate position. A more efficient approach makes use of the fact that we know what the sorted deck looks like. We simply lay out the cards in a row, with each position in the row reserved for the particular card. As we pick up a card, we place it in its reserved location. In the end, all the cards are in their correct location and we collect them from left to right.

Exercise 7.1 *Explain why this sorting technique always takes $O(n)$ time for a deck of n cards.*

Such an approach is the basis for a general sorting technique called *bucket sort*. By quickly inspecting values (perhaps a word) we can approximately sort them into different buckets (perhaps based on the first letter). In a subsequent pass we can sort the values in each bucket with, perhaps a different sort. The buckets of sorted values are then accumulated, carefully maintaining the order of the buckets, and the result is completely sorted. Unfortunately, the worst-case behavior of this sorting technique is determined by the performance of the algorithm we use to sort each bucket of values.

Exercise 7.2 *Suppose we have n values and m buckets and we use insertion sort to perform the sort of each bucket. What is the worst-case time complexity of this sort?*

Such a technique can be used to sort integers, especially if we can partially sort the values based on a single digit. For example, we might develop a support function, `digit`, that, given a number `n` and a decimal place `d`, returns the value of the digit in the particular decimal place. If `d` was 0, it would return the units digit of `n`. Here is a recursive implementation:
Here is the code for placing a list of integer values among 10 buckets, based on the value of digit `d`. For example, if we have numbers between 1 and 52 and we set `d` to 2, this code almost sorts the values based on their 10's digit.

We now have the tools to support a new sort, *radix sort*. The approach is to use the `bucketPass` code to sort all the values based on the units place. Next, all the values are sorted based on their 10's digit. The process continues until enough passes have been made to consider all the digits. If it is known that values are bounded above, then we can also bound the number of passes as well. Here is the code to perform a radix sort of values under 1 million (six passes):
After the first `bucketPass`, the values are ordered, based on their units digit. All values that end in `0` are placed near the front of `data` (see Figure 7.11), all the values that end in `9` appear near the end. Among those values that end in `0`, the values appear *in the order they originally appeared in the list.* In this regard, we can say that `bucketPass` is a *stable* sorting technique. All other things being equal, the values appear in their original order.

During the second pass, the values are sorted, based on their 10's digit. Again, if two values have the same 10's digit, the relative order of the values is maintained. That means, for example, that 140 will appear before 42, because after the first pass, the 140 appeared before the 42. The process continues, until all digits are considered. Here, six passes are performed, but only three are necessary (see Problem **??**).

There are several important things to remember about the construction of this sort. First, `bucketPass` is stable. That condition is necessary if the work of previous passes is not to be undone. Secondly, the sort is unlikely to work if the passes are performed from the most significant digit toward the units digit. Finally, since the number of passes is independent of the size of the `data` list, the speed of the entire sort is proportional to the speed of a single pass. Careful

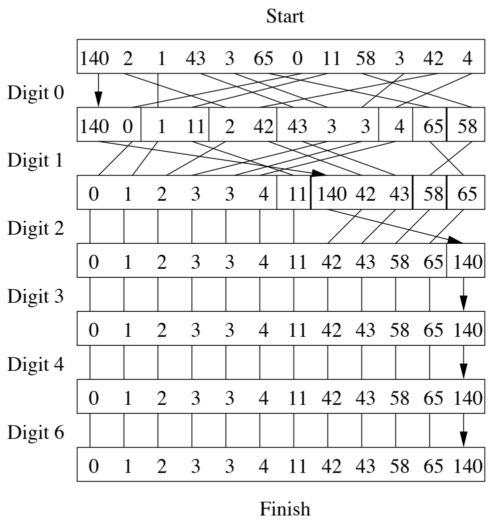


Figure 7.11 The state of the `data` list between the six passes of `radixSort`. The boundaries of the buckets are identified by vertical lines; bold lines indicate empty buckets. Since, on every pass, paths of incoming values to a bucket do not cross, the sort is stable. Notice that after three passes, the `radixSort` is finished. The same would be true, no matter the number of values, as long as they were all under 1000.

design allows the `bucketPass` to be accomplished in $O(n)$ time. We see, then, that `radixSort` is a $O(n)$ sorting method.

While, theoretically, `radixSort` can be accomplished in linear time, practically, the constant of proportionality associated with the bound is large compared to the other sorts we have seen in this chapter. In practice, `radixSort` is inefficient compared to most other sorts.

7.9 Sorting Objects

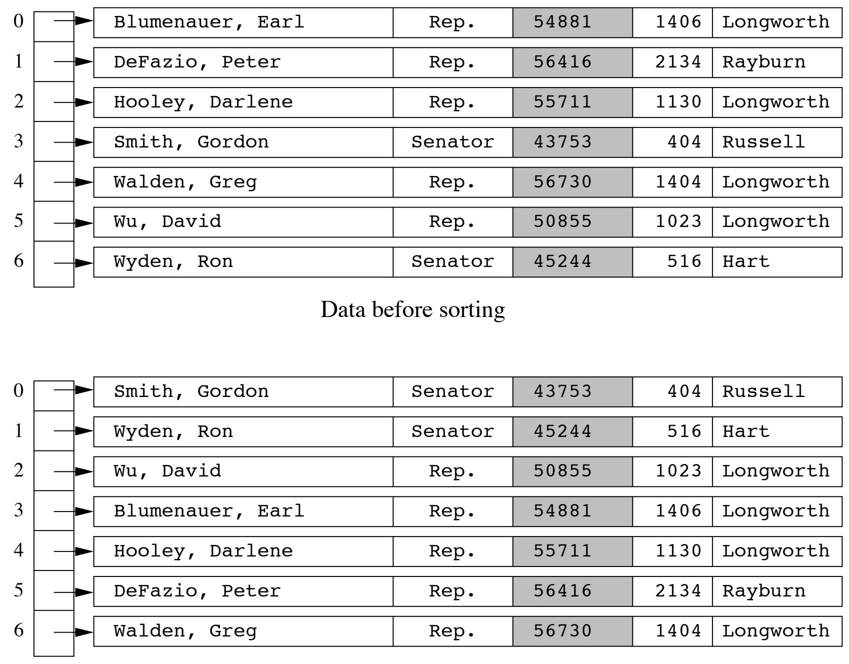
Sorting lists of integers is suitable for understanding the performance of various sorts, but it is hardly a real-world problem. Frequently, the object that needs to be sorted is an `Object` with many fields, only some of which are actually used in making a comparison.

Let's consider the problem of sorting the entries associated with an electronic phone book. The first step is to identify the structure of a single entry in the phone book. Perhaps it has the following form:
We have added the `compareTo` method to describe the relation between two entries in the phone book (the shaded fields of Figure 7.12). The `compareTo` method returns an integer that is less than, equal to, or greater than 0 when `this` is logically less than, equal to, or greater than `other`. We can now modify any of the sort techniques provided in the previous section to sort a list of phone entries:
Careful review of this insertion sort routine shows that all the $<$ operators have been replaced by checks for negative `compareTo` values. The result is that the phone entries in the list are ordered by increasing phone number.

If two or more people use the same extension, then the order of the resulting entries depends on the stability of the sort. If the sort is stable, then the relative order of the phone entries with identical extensions in the sorted list is the same as their relative order in the unordered list. If the sort is not stable, no guarantee can be made. To ensure that entries are, say, sorted in increasing order by extension and, in case of shared phones, sorted by increasing name, the following `compareTo` method might be used:
Correctly specifying the relation between two objects with the `compareTo` method can be difficult when the objects cannot be *totally ordered*. Is it always possible that one athletic team is strictly less than another? Is it always the case that one set contains another? No. These are examples of domains that are *partially ordered*. Usually, however, most types may be totally ordered, and imagining how one might *sort* a collection of objects forces a suitable relation between any pair.

7.10 Ordering Objects Using Comparators

The definition of the `compareTo` method for an object should define, in a sense, the natural ordering of the objects. So, for example, in the case of a phone



0	Blumenauer, Earl	Rep.	54881	1406	Longworth
1	DeFazio, Peter	Rep.	56416	2134	Rayburn
2	Hooley, Darlene	Rep.	55711	1130	Longworth
3	Smith, Gordon	Senator	43753	404	Russell
4	Walden, Greg	Rep.	56730	1404	Longworth
5	Wu, David	Rep.	50855	1023	Longworth
6	Wyden, Ron	Senator	45244	516	Hart

Data before sorting

0	Smith, Gordon	Senator	43753	404	Russell
1	Wyden, Ron	Senator	45244	516	Hart
2	Wu, David	Rep.	50855	1023	Longworth
3	Blumenauer, Earl	Rep.	54881	1406	Longworth
4	Hooley, Darlene	Rep.	55711	1130	Longworth
5	DeFazio, Peter	Rep.	56416	2134	Rayburn
6	Walden, Greg	Rep.	56730	1404	Longworth

Data after sorting by telephone

Figure 7.12 A list of phone entries for the 107th Congressional Delegation from Oregon State, before and after sorting by telephone (shaded).

book, the entries would ideally be ordered based on the name associated with the entry. Sometimes, however, the `compareTo` method does not provide the ordering desired, or worse, the `compareTo` method has not been defined for an object. In these cases, the programmer turns to a simple method for specifing an external comparison method called a *comparator*. A comparator is an object that contains a method that is capable of comparing two objects. Sorting methods, then, can be developed to apply a comparator to two objects when a comparison is to be performed. The beauty of this mechanism is that different comparators can be applied to the same data to sort in different orders or on different keys. In Java a comparator is any class that implements the `java.util.Comparator` interface. This interface provides the following method:
Like the `compareTo` method we have seen earlier, the `compare` method returns an integer that identifies the relationship between two values. Unlike the `compareTo` method, however, the `compare` method is not associated with the compared objects. As a result, the comparator is not privy to the implementation of the objects; it must perform the comparison based on information that is gained from accessor methods.

As an example of the implementation of a `Comparator`, we consider the implementation of a case-insensitive comparison of `Strings`, called `CaselessComparator`. This comparison method converts both `String` objects to uppercase and then performs the standard `String` comparison:
The result of the comparison is that strings that are spelled similarly in different cases appear together. For example, if a list contains the words of the children's tongue twister:
we would expect the words to be sorted into the following order:
This should be compared with the standard ordering of `String` values, which would generate the following output:

To use a `Comparator` in a sorting technique, we need only replace the use of `compareTo` methods with `compare` methods from a `Comparator`. Here, for example, is an insertion sort that makes use of a `Comparator` to order the values in a list of `Objects`:
Note that in this description we don't see the particulars of the types involved. Instead, all data are manipulated as `Objects`, which are specifically manipulated by the `compare` method of the provided `Comparator`.

7.11 Vector-Based Sorting

We extend the phone book example one more time, by allowing the `PhoneEntrys` to be stored in a `Vector`. There are, of course, good reasons to use `Vector` over lists, but there are some added complexities that should be considered. Here is an alternative Java implementation of `insertionSort` that is dedicated to the sorting of a `Vector` of `PhoneEntrys`:
Recall that, for `Vectors`, we use the `get` method to fetch a value and `set` to store. Since any type of object may be referenced by a vector entry, we verify the type expected when a value is retrieved from the vector. This is accomplished

through a parenthesized *cast*. If the type of the fetched value doesn't match the type of the cast, the program throws a *class cast exception*. Here, we cast the result of `get` in the `compareTo` method to indicate that we are comparing `PhoneEntrys`.

It is unfortunate that the `insertionSort` has to be specially coded for use with the `PhoneEntry` objects.

Exercise 7.3 *Write an* `insertionSort` *that uses a* `Comparator` *to sort a* `Vector` *of objects.*

7.12 Conclusions

Sorting is an important and common process on computers. In this chapter we considered several sorting techniques with quadratic running times. Bubble sort approaches the problem by checking and rechecking the relationships between elements. Selection and insertion sorts are based on techniques that people commonly use. Of these, insertion sort is most frequently used; it is easily coded and provides excellent performance when data are nearly sorted.

Two recursive sorting techniques, mergesort and quicksort, use recursion to achieve $O(n \log n)$ running times, which are optimal for comparison-based techniques on single processors. Mergesort works well in a variety of situations, but often requires significant extra memory. Quicksort requires a random access structure, but runs with little space overhead. Quicksort is not a stable sort because it has the potential to swap two values with equivalent keys.

We have seen with radix sort, it is possible to have a linear sorting algorithm, but it cannot be based on compares. Instead, the technique involves carefully ordering values based on looking at portions of the key. The technique is, practically, not useful for general-purpose sorting, although for many years, punched cards were efficiently sorted using precisely the method described here.

Sorting is, arguably, the most frequently executed algorithm on computers today. When we consider the notion of an *ordered structure*, we will find that algorithms and structures work hand in hand to help keep data in the correct order.

Self Check Problems

Solutions to these problems begin on page **??**.

Chapter 8

Instrumentation and Analysis

An important aspect of data structure design is analysis of performance. The data structures we use in our programs and algorithms can play a dominant role in performance. As an informed engineer, it is important that we take the time to carefully instrument our data structures and programs to allow us to collect information about the performance of our data structures. It's also important that we be able to identify when the performance we measure suggests that there may be ways that we can improve our design.

This chapter develops the tools and strategies that are important to developing not only correct programs but programs that perform as well as possible. We begin with an understanding of *decoration*, an important tool for tool builders we then follow with a discussion of *performance measurement* and *profiling* strategies. We end with the development of a few lightweight tools that can improve the performance of many programs.

8.1 Decoration

A large portion of this chapter is about the development of programs that help us instrument and measure characteristics about other programs. One of the strengths of Python is its ability to support this engineering process. The first step along this path is an understanding of a process called *decoration*, or the annotation of functions and classes.

Suppose we have designed a data structure that provides a feature that is used in many different placeses in a program. An example might be the `append` method of a `list` implementation. How might we keep track of the number of times we call this method? It is likely to be error-prone to find the places where the method is called, and count those; we might not even have access to all of those call sites. Clearly, it would be necessary to modify the *code itself*.

We might imagine, however, that the process of gathering these statistics could be complex—perhaps at least as complex as the code associated with `append`. It is likely, also, that success in analyzing the frequency of `append` methods will lead, ultimately, to the analysis of other methods of the `list` class as well. In all of this, it is important that we not confuse the analysis tool with the method itself. For this reason, it is frequently useful to "abstract away" the details of the analysis in the hope that it lowers the barrier to use and that it does not obscure the object being analyzed.

Python provides a very simple syntactic mechanism, called a method called a *decorator*, that can be used to *augment our definitions of other methods and classes*. Remember, in Python, functions are first-class objects. Once a function is defined, references to it can be passed around and operated on like any other object.[1] A decorator is a *higher order function* that is called with another function as its only parameter. The purpose of the decorator is to return an alternative function to be used it its place. Typically the alternative function is simply a new function that *wraps* the original: it performs some preprocessing of the function's arguments, calls the function with its arguments, and then performs some post processing of the results. Once defined, the higher order function is applied by mentioning it as a decoration just before the original function is defined:

```
@decorator
def f(x):
    print(x)
```

is the same as

```
def f(x)
    print(x)

f = decorator(f)
```

8.1.1 Example: Counting method calls

As a simple example, we will write a decorator that allows us to keep track of the number of times a method is called. Such a decorator might be used to identify how deep recursion runs, or to count the number of times a comparison occurs in a sorting routine.

To begin, we think about how a decorator might be used, syntactically. In a class, a method might be decorated as:

```
@accounting
def fun(self,x):
    ...
```

Again, this is equivalent to the sequence of statements

```
def fun(self,x):
    ...

fun = accounting(fun)
```

Our job is to write the decorator method, `accounting`.

Our first version is as follows:

```
def accounting(f):
```

[1] As we shall see, shortly, the same holds true of classes. When we define a class, it becomes a *callable object*, just like a method.

```
    """Keep track of calls to function f."""
    @wraps(f)
    def wrapped_f(*args,**kargs):
        wrapped_f.counter += 1
        return f(*args,**kargs)
    wrapped_f.counter = 0
    return wrapped_f
```

When the `accounting` decorator is referenced, the method definition that follows it is passed as `f`. Again, this happens when we are *defining* `f`, well before the method is ever called. Within the `accounting` decorator, a new method is constructed and referred to as `wrapped_f`. Looking ahead, this is the return value for the decorator, so this method is what will *actually* be called, whenever we imagine we're calling the method being decorated; when we define `f`, this is the actual value of that binding. Because we don't know anything about the parameters used in `f`, we must be ready to accept all the positional parameters (`*args`) and all of the keyword parameters (`**kargs`). If `wrapped_f` is ever called, it is really a request to call method `f` with the particular parameters. Thus, we increment the number of times this method is called. This integer is stored as an *attribute* of the `wrapped_f` function called `counter`. Before we return `wrapped_f` as our wrapped definition of `f`, we add the `counter` attribute to the newly synthesized method and initialize it to zero.

Because we have provided an alternate definition of the function `f`, any effort to look at the `f`'s documentation will fail. In addition, if an error is generated within the wrapper function, the wrapper call is exposed in the traceback. This is because the documentation associated with the method `f` is stored in the docstring (`__doc__`) that is an attribute of `f` and not `wrapped_f`. A decorator `wraps`, imported from Python's `functools` package copies over all of the special variables that are useful in giving `wrapped_f` the identity of its wrapped function, `f`. The reader is encourage to think about how the `wraps` decorator might be written.

As an excercise in understanding how our `accounting` decorator might be used, we think about the straightforward recursive implementation of a function to compute the n^{th} member of the Fibonacci sequence,

$$f = \{0, 1, 1, 2, 3, 5, \ldots\}$$

where $f_n = f_{n-1} + f_{n-2}$. A common first effort in Python is:

```
def fibo(n):
    if n <= 1:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

Testing out our new function quickly demonstrates its potential computational complexity:

```
>>> fibo(3)
3
```

```
>>> fibo(5)
8
>>> fibo(10)
89
>>> fibo(2000)
(significant time passes, humanity fades)
```

So, what is happening? Since the amount of computation directly attributed to a single call to `fibo` is small (a condition test and a possible addition), it must be the case that *many* function calls are being performed. To see how many, we can decorate `fibo` with our newfound `accounting` decorator:

```
@accounting
def fibo(n):
    if n <= 1:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

The effect of this is to quietly wrap `fibo` with code that increments the counter each time the function is called. We can now use this to count the actual calls to the `fibo` method:

```
>>> fibo(3)
3
>>> fibo.counter
5
>>> fibo.counter = 0; fibo(5)
8
>>> fibo.counter
15
>>> fibo.counter = 0; fibo(20)
10946
>>> fibo.counter
21891
```

Clearly, things get out of hand rather quickly. Our next decorator demonstrates how we can dramatically reduce the complexity of our recursive implementation without losing its beauty.

Exercise 8.1 *Write a new method,* `fibocalls(n)`*, that computes the number of calls that* `fibo(n)` *will make.*

8.1.2 Example: Memoizing Functions

Functions like `fibo` of the previous section perform lots of redundant computation. If we consider what is required to compute `fibo(4)`, we see it is based on `fibo(3)` and `fibo(2)`. But notice that the computation of `fibo(3)`, itself, must call `fibo(2)`. Looking at the computation involved with `fibo(5)`, we see that we compute `fibo(2)` twice for `fibo(4)`, and third time for `fibo(3)`. Figure 8.1 demonstrates the intractability of our naïve implementation of `fibo`.

n	fibocalls(n)
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67
9	109
10	177
20	21891
100	1146295688027634168201
200	907947388330615906394593939394821238467651

Figure 8.1 The number of calls made by `fibo` for various values of `n`.

It would help considerably if we could record, perhaps in a table, the results of any computation that we perform. If a computation is requested later, we can look it up in our table and return the result previously computed. When we compute, on behalf of `fibo(5)`, the values of `fibo(4)` and `fibo(3)`, the value of `fibo(3)` is computed exactly once, as a part of the computation of `fibo(4)`. A little thought will convince us that this will make the computation of `fibo(n)` a linear operation. This table-driven process is called *memoization*.

Ideally, the memoization of `fibo` should be relatively transparent. For this reason we opt to make use of a decorator, `memoize`, instead of modifying the `fibo` function itself. Our approach makes use of a *dictionary*, which allows us to look up a value associated with an arbitrary constant key. In Python, dictionaries are implemented as `dict` objects. The value associated with a key, `k`, in `dict` `d` may be obtained as `d[k]`. To enter or update a key-value pair in the dictionary, we use simple assignment: `d[k] = v`. In our implementation of `memoize`, we think of our dictionary as a `cache`:

```
def memoize(f):
    """Wrap f with a cache of prior computations."""
    @wraps(f)
    def wrapped_f(*args):
        a = tuple(args) # keys must be constant
        if a not in wrapped_f.cache:
            result = f(*args)
            wrapped_f.cache[a] = result
        else:
            result = wrapped_f.cache[a]
        return result
    wrapped_f.cache = dict()
    return wrapped_f
```

The wrapper function, `wrapped_f`, uses a tuple of the particular arguments passed to the function `f` to look up any pre-computed results. If the function call was previously computed, the previous result is returned, avoiding the overhead of redundantly computing it. Otherwise, the result of the new computation is recorded for possible future use and then returned.

Before the wrapper is returned as the new definition, we attach an empty dictionary as the cache attribution on the wrapper. Over time, of course, this dictionary fills with the history of function calls and their respective results.

To make use of the `memoize` decorator, we tuck it between the `accounting` decorator and the function definition so that we may count the number of memoized calls:

```
@accounting
@memoize
def fibo(n):
    if n <= 1:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

This decoration order is a shorthand for

```
def fibo(n):
    ...

fibo = accounting( memoize( fibo ) )
```

Armed with our memoized version of `fibo` we can get answers to questions we had not been able to compute:

```
>>> fibo(50)
20365011074
>>> fibo.counter
99
>>> fibo.counter = 0; fibo(20)
10946
>>> fibo.counter
1
>>> fibo.counter = 0; fibo(100)
573147844013817084101
>>> fibo.counter
199
>>> fibo(200)
453973694165307953197296969697410619233826
```

In our final implementation of `fibo`, of course, we would probably remove the `accounting` decorator—which was only there for as a performance indicator—and keep the `memoize` decorator—which is a performance optimization.

8.1.3 Example: Decorate-Sort-Undecorate Idiom

When discussing sorting we introduced the idiom *decorate-sort-undecorate* (DSU). Recall that this programming pattern pre-processed the list to be sorted in a way that reduced the number of sort key computations and guaranteed stability for otherwise unstable sorts. The notion of *wrapping* the underlying sort technique was natural, and so we are motivated to provide a decorator for sort functions that provides a sorting interface that Python users expect in their sorting routines. We call this decorator `sortoptions`:

```
def sortoptions(f):
    """Decorator for sorting methods that adds key and reverse options."""
    @wraps(f)
    def wrapper(l,n=None,key=None,reverse=False):
        if n is None:
            n = len(l)
        if key is None:
            key = lambda x: x
        for i in range(n):
            l[i] = (key(l[i]),i,l[i])
        f(l,n)
        for i in range(n):
            l[i] = l[i][-1]
        if reverse:
            l[:n] = reversed(l[:n])
    return wrapper
```

The technical details of the DSU approach were discussed earlier (see page **??**), but we point out a few decoration-specific details, here. Python's `sorted` method supports the keyword arguments `key` and `reverse`. Because they were not important to our discussion of particular sorting techniques, our sort methods did not support these keyword arguments. Instead, all of our sort methods took a list and the number of elements from the list we wished to sort. It is the job of the DSU wrapper, then, to interpret the keyword arguments and, if the number of elements is not provided to have it default to the length of the list. The author tried to develop a wrapper that was efficient enough to have negligible impact on the underlying sort. If further optimizations can be identified, however, modifying the decorator ensures that *all* the sorts are improved simultaneously.

As a glimpse of the power of this interface, we experiment with using `quicksort` to sort the first 20 integers based on the length of their Syracuse sequence (see page **??**):

```
>>> l = list(range(20))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> [syrlen(i) for i in l]
[1, 1, 2, 8, 3, 6, 9, 17, 4, 20, 7, 15, 10, 10, 18, 18, 5, 13, 21, 21]
>>> quicksort(l,key=syrlen)
>>> l
```

```
[0, 1, 2, 4, 8, 16, 5, 10, 3, 6, 12, 13, 17, 11, 7, 14, 15, 9, 18, 19]
>>> l = list(reversed(range(20)))
>>> l
[19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> quicksort(l,key=syrlen)
>>> l
[1, 0, 2, 4, 8, 16, 5, 10, 3, 6, 13, 12, 17, 11, 7, 15, 14, 9, 19, 18]
```

Notice that the relative ordering of values that have the same Syracuse trajectory length is determined by the initial ordering of the list to be sorted. Because the `sortoptions` decorator is used for each of our sorting techniques, the results of the experiment are the same no matter which sort we use.

8.2 Timing and Tracing

Python, like most languages, provides a simple mechanism to get the current *wall clock* time: the `time` method imported from the `time` module is called with no arguments and returns the number of seconds since the 1960's.[2] This, itself, is not particularly useful, but differences in wall clock time can help us establish a fairly accurate measure of the time elapsed. On most systems the measurement of time is accurate to the microsecond.

Let's imagine we would like to measure the time it takes to `quicksort` 1000 values in a randomly ordered list `l`. A good first guess would code similar to

```
starttime = time()
quicksort(l)
stoptime = time()
elapsedtime = stoptime-starttime
print("The elapsed time is {} seconds.".format(elapsedtime))
```

Another approach would be to develop a `timeaccounting` decorator that does much the same thing:

```
def timeaccounting(f):
    """Keep track of elapsed time for last call to function f."""
    @wraps(f)
    def wrapped_f(*args,**kargs):
        starttime = time()
        result = f(*args,**kargs)
        stoptime = time()
        wrapped_f.elapsedtime = stoptime-starttime
        return result
    wrapped_f.elapsedtime = 0
    return wrapped_f
```

This version keeps track of the elapsed time associated with the most recently

[2] Thursday, January 1, 1970 is known as the Unix Epoch.

finished call to the function `f`. Decorating the `quicksort` method would then allow us to perform the following:

```
>>> quicksort(l)
>>> quicksort.elapsedtime
0.004123442259382993
```

This tells us that it took three fourths of a second to perform the sort. If we run the experiment ten times, we get the following output:

```
>>> for i in range(10):
...     k = list(l)
...     quicksort(k)
...     print("ime is {} seconds.".format(quicksort.elapsedtime))
...
Time is 0.008211135864257812 seconds.
Time is 0.006258964538574219 seconds.
Time is 0.004125833511352539 seconds.
Time is 0.003949880599975586 seconds.
Time is 0.004060029983520508 seconds.
Time is 0.003949165344238281 seconds.
Time is 0.004272937774658203 seconds.
Time is 0.0040531158447265625 seconds.
Time is 0.003945112228393555 seconds.
Time is 0.004277944564819336 seconds.
```

How are we to interpret this data? Clearly there is some variability in the elapsed time. Most modern computers run tens or even hundreds of processes that, at various times, may interrupt your experiment as it progresses. For example, the system clock needs to be updated, mail is checked, the mouse image is moved, garbage is collected, and the display is updated. If any of these distractions happens during your experiment, the elapsed time associated with `quicksort` is increased. When multiple experiments are timed, each may be thought of as an upper bound on the length of time. It is appropriate, then to take the minimum time as a reasonable estimate of the time consumed. Further measurements may be larger, but over many experiments, the estimate becomes more accurate. Here, for example, we see that `quicksort` can sort 1000 integers in approximately 3922 microseconds.

Some experiments, of course, are finished in well under the resolution of the clock. In this book, all times are measured in units informally called a *click*, the length of time it takes Python to increment an integer. On the author's machine, a click is approximately 34 nanoseconds. In these cases it can be useful to repeat the experiment enough times to make the total time significantly greater than the resolution of the clock. If, for example, we measure 1000 repeated experiments that take a total of 1000 microseconds, the time of each individual experiment is about 1 microsecond. One must take care, of course, to consider the overhead associated with running the experiments 1000 times. Again, on the author's machine, that may be as much as 63 nanoseconds per iteration.

On most modern unix-based operating systems, the system allows a waiting process to run largely uninterrupted, until the next scheduling event. That quantum of time is called a *jiffy* and is in the range of $\frac{1}{100}$ to $\frac{1}{10}$ of a second. If possible, we try to measure elapsed times that are smaller than a jiffy. Certain actions will cause our Python process to give up the machine earlier than a jiffy. For example, if we perform a read or write operation it is likely that some process serving the keyboard, display, or file, will have to be woken up. For this reason, we try to avoid doing any I/O until *after* all of our experiments have been performed. Because writing to I/O devices can be slow, and often causes many interruptions to concurrently executing processes, we may even see the effects of I/O that happened *before* our experiments.

8.2.1 The `timeit` Module

Timing snippets of code is a common task of analyzing the performance of code we write. For this reason, Python provides a module of timing tools, called `timeit`. These tools allow you to perform scripted, interactive, or standalone timing experiments. We will investigate, here, how to perform standalone timing experiments.

One of the most important uses of the `timeit` module is to investigate the timing of Python code. The approach is very similar to that of the previous section, but has the great advantage that it can be used as a quick, off-the-shelf tool precisely when needed. Often, the results are quite surprising. As an ongoing example to motivate our tour of `timeit`, we'll investigate the time it takes to perform an operation, the increment of a small integer. As mentioned, we use this quantum of time throughout this text as a basic unit that is relatively independent of the type of machine. Presenting timing information in this way helps us to focus on what is important: algorithmic resource utilization trends.

From the point-of-view of `timeit`, all experiments are performed as part of a generic Python loop:

```
setup-statment
for trial in range(trials):
    experiment
```

When us use the `timeit` utility, you can specify the `setup-statement` and one or more `experiment` statements. The setup is exectuted precisely once, while the experiment—the statement we are trying to measure—is executed `trials` times. The particular value of `trials` can be specified directly or, by default, the `timeit` module will determine an appropraite value for trials.

Now, let's imagine we would like to measure the length of time it takes to perform the computation

```
i += 1
```

The variable `i`, of course, we take to be an integer. We can ensure that by initializing `i` to zero as our setup statement. Now it is time to figure out how to actually run `timeit` as a utility.

Many modules have a dual personality: when imported for use in a program, the provide a set of classes and functions for the programmer's use. When run as a standalone program, however, they can be used directly as an analysis tool. This latter approach is how we will use the `timeit` module here. The `timeit` tool is generally called as:

```
python3 -m timeit ...
```

In the past, when we've wanted to run standalone Python programs, those programs have been located in the current directory. The `-m` switch tells Python to use the standard module-locating techniques to find the indicated module, and then to run it as a standalone program.

The `timeit` module provides a number of *switches* that configure and control the timing experiment. Some of these switches are presented in Table 8.2. For our simple purposes, here, we use the following command line:

```
python3 -m timeit -s 'i=0' -- 'i+=1'
```

and the result is:

```
10000000 loops, best of 3: 0.0469 usec per loop
```

Here, `timeit` has determined that 10 million increments were needed to make the timing sufficiently accurate—the total experiment time was about a half a second. `timeit` also reports the best measurement: the entire measurement process was repeated 3 times, and the values reported here were the *minimum*. The reasons for this have been discussed in the previous section.

Switch	**Purpose**
`-n`	Directly specify the number of trials to be timed
`-s`	Specify the setup statement
`-r`	Specify the number of times to repeat the entire experiment (default is 3)
`--`	Separate switches from statements to be timed

Figure 8.2 Switches available as part of the standalone `timeit` module.

By specifying multiple statements to be timed, we can perform even more intereseting experiments. For example,

```
python3 -m timeit -s 'i=0' -- 'i+=1' 'i+=1'
10000000 loops, best of 3: 0.086 usec per loop
```

This is slightly strange: why does repeating the statement cause the total time measured to increase to *less* than twice the previous time? The reason is subtle. Because Python has arbitrary precision arithmetic, the notion of "incrementing an integer" depends on how big the integer is. Because 10 million experiments are performed, the value of `i` takes on values that approach 20 million. The cost of incrementing larger integers is (slightly) different than incrementing smaller integers.

We can test out our theory by timing increasingly complex experiments. Our simplest is to increment an integer variable.

```
    i += 1
```

The `timeit` module reports this as taking about 45 nanoseconds:

```
    10000000 loops, best of 10: 0.0445 usec per loop
```

Our second experiment is to measure the time it takes to increment `i`, immediately followed by a decrement:

```
    i += 1
    i -= 1
```

This snippet of code takes about 61 nanoseconds:

```
10000000 loops, best of 10: 0.0609 usec per loop
```

These two experiments lead us to a small surprise: the timing of two similar statments is less than twice the time for just one. A possible reason for this is that incrementing `i` 10 million times takes `i` from 0 to nearly 10 million. In Python, which supports arbitrary precision arithmetic, the cost of performing an addition on a small integer takes less time than on a larger value. The second experiment, then, leaves `i` as a value that is either zero or one, so the two operations are much simpler, on average, than the single operation on a large value.

We can test our hypothesis by initializing `i` to 5 million:

```
    python3 -m timeit -s 'i=5000000' -- 'i+=1' 'i-=1'
```

which, not surprisingly, takes more than 80 nanoseconds:

```
    10000000 loops, best of 3: 0.0822 usec per loop
```

We will take, as our definition of the time it takes to increment an integer, half the time of our increment-decrement experiment, or about 30 nanoseconds. Again, it is important to remember that these results will differ, significantly, on different platforms; the reader is encouraged to perform these experiments on their own machines.

One might be tempted, as an alternative, to time the initialization of `i`:

```
    python3 -m timeit -- 'i=0'
    100000000 loops, best of 3: 0.0137 usec per
loop
```

and then use that to expose the time of an increment that follows:

```
    python3 -m timeit -- 'i=0' 'i+=1'
    10000000 loops, best of 3: 0.0408 usec
per loop
```

which yields a somewhat smaller time of about 27 nanoseconds. The reason for this is the fact that the Python interpreter can optimize these two instructions into a single simpler operation that directly loads 1 into `i`.

The ability to perform very precise timings allows for careful experimental design and evaluation of algorithms. **Expand on this.**

Chapter 9

Sorted Structures

One important use of data structures is to help keep data sorted—the smallest value in the structure might be stored close to the front, while the largest value would be stored close to the rear. Once a structure is kept sorted it also becomes potentially useful as a sorting method: we simply insert our possibly unordered data into the structure and then extract the values in sorted order. To do this, however, it is necessary to *compare* data values to see if they are in the correct order. In this chapter we will discuss approaches to the various problems associated with maintaining ordered structures. First we review material we first encountered when we considered sorting.

9.1 Comparable Objects

In Python, most types that are native to the language can be compared. This ordering is determined by a group of *rich comparison methods* that correspond to the various comparisons. These are indicated in Table 9.1.

Operator	**Special method**
`<`	`__lt__`
`<=`	`__le__`
`==`	`__eq__`
`!=`	`__ne__`
`>=`	`__ge__`
`>`	`__gt__`

Figure 9.1 Comparison operators and their rich comparision methods.

Python has a very complex scheme to make sure that ordering can be determined, even when specific comparison methods have not been defined. In addition, Python, does not make assumptions about the consistency of the relationships between the comparison methods; for example, it's not necessary that `==` and `!=` return opposite boolean values. While it may sometimes be useful to define these operations in inconsistent ways, most classes defined in Python benefit from the definition of one or more of these methods, in a consistent manner.

Let's see a particular example. Suppose, for example, that we're interested in determining the relative ordering of Ratio values. (Recall, we first met the Ratio class in Chapter **??**, on page **??**.) The relative ordering of Ratio types depends not indirectly on the ordering of values held internally, but the ordering of the products of opposite numerators and denominators of the Ratios involved. For this reason the implementation of the special ordering methods is nontrivial. As examples, we consider the definition of the __lt__ and __eq__ methods from that class:

```
def __lt__(self,other):
    """Compute self < other."""
    return self._top*other._bottom < self._bottom*other._top

def __le__(self,other):
    """Compute self < other."""
    return self._top*other._bottom <= self._bottom*other._top

def __eq__(self,other):
    """Compute self == other."""
    return (self._top == other._top) and \
           (self._bottom == other._bottom)
```

When two Ratio values are compared, the appropriate products are computed and compared, much as we learned in grade school. Note that the implementation of the __lt__ special method involves a < comparison of the products—integer values. This, of course, calls the special method __lt__ for integers. The implementation of the <= operator follows the same pattern. The implementation of the __eq__ method ultimately performes a == test of numerators and denominators, and depends on the ratios always being represented in their least terms, which, as you may recall, they are. Other operators can be defined, for the Ratio class, in a similar manner.

Some classes, of course, represent values that cannot be ordered in a particular manner (Complex values, for example). In these cases, the rich comparison methods are not defined and an error is thrown if you attempt to compare them. In the context of our current thinking, of course, it is meaningless, then to place values of this type in a sorted structure.

In implementations similar to Ratio, of course, it is important that we define these methods in a consistent manner. Careful attention, of course, will make this possible, but given there are a half dozen methods that are related, it is important that maintainers of these classes be vigilant: a change to one method should be reflected in the implementation of *all* the methods.

Python provides, as part of the functools package, a class decorator, total_ordering, that, given the implementation of one or more of the comparison methods, defines each of the others in a consistent manner. For example, the Ratio class is decorated with the total_ordering decorator to guarantee that each of the rich comparison methods is correctly defined. Typically, when efficient operations cannot be inherited, specific definitions are given for the __lt__, __le__, and __eq__, and the remaining rich comparison functions are provided by the total_ordering class decorator.

9.2 Keeping Structures Ordered

We can make use of the natural ordering of classes suggested by the `__lt__` method to organize our structure. Keeping data sorted, however, places significant constraints on the type of operations that should be allowed. If a comparable value is added to a structure that orders its elements, the relative position of the new value is determined by the data, not the structure. Since this placement decision is predetermined, ordered structures have little flexibility in their interface. It is not possible, for example, to insert data at random locations. While simpler to use, these operations also tend to be more costly than their sorted counterparts. Typically, the increased *energy* required is the result of an increase in the friction associated with decisions needed to accomplish `add` and `remove`.

The implementation of the various structures we see in the remainder of this chapter leads to simpler algorithms for sorting, as we will see in Section **??**.

9.2.1 The Sorted Interface

Recall that a container is any traversable structure that allows us to add and remove elements and perform membership checks (see Section **??**). Since the `Structure` interface also requires the usual size-related methods (e.g., `size`, `isEmpty`, `clear`), none of these methods actually requires that the data within the structure be kept in order. To ensure that the structures we create order their data (according to their native ordering), we make them abide by an extended interface, `Sorted`:

```
@checkdoc
class Sorted(Iterable, Sized):

    """
    An abstract base class for SortedList, SkipList, SearchTree, SplayTree,
    and RedBlackSearchTree.

    For more information on sorted data structures, see sorted.py.
    """

    @abc.abstractmethod
    def add(self, value):
        """Insert value into data structure."""
        ...

    @abc.abstractmethod
    def remove(self, value):
        """Remove value from data structure."""
        ...

    def __eq__(self, other):
        """Data structure has equivalent contents to other."""
        if len(self) != len(other):
            return False
        for x,y in zip(self, other):
            if x != y:
                return False
        return True
```

This interface demands that implementors ultimately provide methods that `add` values to the structure and `remove` them by value. Since the values are always held within the structure in fully sorted ordering, the equality test for these containers can perform pairwise comparisons of the values that occur in the traversals.

9.2.2 The SortedList and Binary Search

We can now consider the implementation of a sorted list of values. Since it implements the `Sorted` interface, we know that the order in which elements are added does not directly determine the order in which they are ultimately removed. Instead, when elements are added to the `SortedList`, they are kept ascending in their natural order.

Constructing an sorted list requires little more than allocating the underlying list:

```
def __init__(self, data=None, frozen=False):
    """Construct a SortedList from an iterable source."""
    self._data = []
    if data:
        for x in data:
            self.add(x)
```

Rather obviously, if there are no elements in the underlying `Vector`, then all of the elements are in order. Initially, at least, the structure is in a consistent state. We must always be mindful of consistency.

Because finding the correct location for a value is important to both adding and removing values, we focus on the development of an appropriate search technique for `SortedLists`. This process is much like looking up a word in a dictionary, or a name in a phone book (see Figure 9.2). First we look at the value halfway through the list and determine if the value for which we are looking is bigger or smaller than this *median*. If it is smaller, we restart our search with the left half of the structure. If it is bigger, we restart our search with the right half of the list. Whenever we consider a section of the list consisting of a single element, the search can be terminated, with the success of the search dependent on whether or not the indicated element contains the desired value. This approach is called *binary search*.

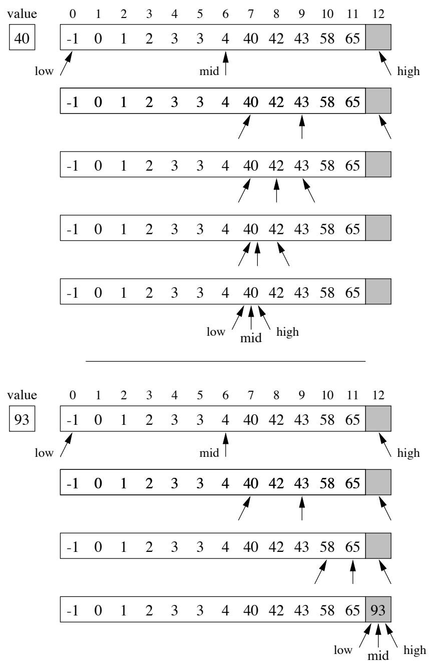


Figure 9.2 Finding the correct location for a comparable value in a sorted list. The top search finds a value in the array; the bottom search fails to find the value, but finds the correct point of insertion. The shaded area is not part of the list during search.

We present here the code for determining the index of a value in an `SortedList`. Be aware that if the value is not in the `list`, the routine returns the ideal location to insert the value. This may be a location that is just beyond the bounds of the `list`.

```
def _locate(self, target):
    """Find the correct location to insert target into list."""
    low = 0
    high = len(self._data)
    mid = (low + high) // 2
    while high > low:
        mid_obj = self._data[mid]
        if target < mid_obj:
            # if the object we’re trying to place is less than mid_obj, look left
            # new highest possible index is mid
            high = mid
        else:
            # otherwise, lowest possible index is one to the right of mid
            low = mid + 1
        # recompute mid with new low or high
        mid = (low + high) // 2
    # if target is in the list, the existing copy is located at self._data[low - 1]
    return low
```

For each iteration through the loop, `low` and `high` determine the bounds of the list currently being searched. `mid` is computed to be the middle element (if there are an even number of elements being considered, it is the leftmost of the two middle elements). This middle element is compared with the parameter, and the bounds are adjusted to further constrain the search. The `_locate` imagines that you’re searching for an ideal insertion point for a new value. If values in the list are equal to the value searched for, the `_locate` method will return the index of the first value *beyond* the equal values. This approach makes it possible to implement stable sorting relatively easily. Since the portion of the list participating in the search is roughly halved each time, the total number of times around the loop is approximately $O(\log_2 n)$. This is a considerable improvement over the implementation of the `__contains__` method for lists of arbitrary elements—that routine is *linear* in the size of the structure.

Notice that `_locate` is declared with a leading underscore and is thought of as a private member of the class. This makes it difficult for a user to call directly, and makes it hard for a user to write code that depends on the underlying implementation. To convince yourself of the utility of this, several `Sorted` containers of this chapter have exactly the same interface (so these data types can be interchanged), but they are completely different structures. If the `_locate` method were made public, then code could be written that makes use of this list-specific method, and it would be impossible to switch implementations.

Implementation of the `_locate` method makes most of the nontrivial `Sorted` methods more straightforward. The `__contains__` special method is implemented by searching for the appropriate location of the value in the sorted list and returns true if this location actually contains an equivalent value. Because `_locate` would return an index one beyond any equivalent values, we would expect to find equivalent values at the location *just prior* to the one returned by `_locate`.

```
def __contains__(self, value):
    """Determine if a SortedList contains value."""
    index = self._locate(value) - 1
    # if list is empty, value cannot be in it
    if index == -1:
        return False
    return self._data[index] == value
```

The `add` operator simply adds an element to the `SortedList` in the position indicated by the `_locate` operator:

```
def add(self, value):
    """Insert value at the correct location in the SortedList."""
    index = self._locate(value)
    self._data.insert(index, value)
```

Notice that if two equal values are added to the same `SortedList`, the last value added is inserted later in the list.

To remove a value, we first attempt to locate it. If it is found, we capture the value in the list for return, and then remove it from the list. In all other cases, the operation does not return a value:

```
def remove(self, value):
    """Remove and return one instance of value list, if present."""
    index = self._locate(value) - 1
    if index >= 0:
        target = self._data[index]
        if target == value:
            del self._data[index]
            return target
    return None
```

Notice that we use the `del` operation to remove a specific location in the underlying `list`; the `remove` operation from `list` removes by value, not location. We also return the value stored in the list, even though we know it to be equal to the value that was to be removed. The reason we do this is because, though the *values* may be equal, they may, in fact, be substantially different. For example, we may remove a driver from a driver data base using a driver specification that is sufficient only for the purposes of comparison. Other features that play no role in the comparison—say, dates of license renewal and driving record—are only features of the record stored within the database.

9.3 Skip Lists

The sorted list structure provides an improvement for applications that require many queries of the data structure. Because the structure is still, effectively, a `list`, the time it takes to add or remove a value from the structure is still linear. When we studied linked list structures we realized that insertion and removal could be fast, but searching the structure was slow. Is there some approach that is fast for *both* lookup and modification of the structure? We will see many structures—most of them not obviously "list-like" in their structure—that have the promise of fast queries and modification. At this point, however, it is useful to investigate a list-like linked structure that is *likely* to provide efficient operations, the *skip list*.

Linked lists are slow to traverse because it is not easy to make great strides over many elements: each element has one reference to the value that follows. If, however, we had a *second* reference that skipped the next node and referenced a node *two* away, one could imagine the search for a relatively large value in a list could be accomplished by first taking strides of two and then possibly a single stride of one. Adding this second pointer would improve the lookup time by a factor of two. Removal of a value would have to readjust, of course, more pointers. By adding a third pointer, finding a value would take only $\mathrm{th}\frac{1}{4}$ the time. Indeed, adding $O(\log_2 n)$ pointers to each node would allow one to find a value in $O(\log_2 n)$ time. The number of pointers that must be adjusted, however, begins to make modification considerably more difficult.

One might imagine the approach used by mass transit authorities in larger cities. Busses or trains for a single route might be either "local" and "express". Local busses stop at every stop along the route, while express busses are faster because they only ever stop at, say, every fifth stop. Most riders looking to get home quickly with such an arrangement can expect to have to transfrer between express and local busses. Express stops service see both types of busses, while other most local stops only see local busses. The system is fast not because *every* stop has an express option, but just that *many* stops have that option.

In skip lists, there are several linked paths through the list. Some paths visit a only a few nodes, allowing for a quick traverals, while other paths are longer, allowing for more accurate local searching of values. Since a single node may appear on several paths, it is convenient to think of the "height" of a node as the number of paths it supports. All nodes have height at least 1 (linked as a linked list), but a few nodes are height 2, and fewer still are higher. The height of a node is determined when it is created, and it remains fixed throughout the its life. This allows the linkage between nodes to remain stable most of the time. An important aspect of the performance of skip lists is that the height is determined randomly, and is independent of the actual data stored in the list. This allows for us to make bold statements about the *expected* behavior of operations, even though, in most cases, *worst case* behavior linear and rare.

To support the implementation of skip lists, we store data in nodes that have one or more `next` pointers. Because this is a structure that is not exported, we allow for direct access to both fields from the list itself:

```
class SkipListNode():
    """A node for use in SkipList."""
    __slots__ = ["data", "next"]
    def __init__(self, data=None, next=None):
        self.data = data
        # pointer to next node in SkipList
        # next[i] is the next node at level i
        self.next = [] if next is None else next
```

The `SkipList` is, essentially, a linked list of `node` structures. It keeps explicit track of its size, a probability associated with computing the node height, and a list of next pointers, one for each path through the list. It is initialized in the following manner:

```
@checkdoc
class SkipList(Sorted):

    __slots__ = [ "_p", "_size", "_head", "_frozen", "_hash" ]

    def __init__(self, data=None, p=0.5, frozen=False):
        """Construct a SkipList from an iterable source.
        p is a probability that a node will have more than one linkage pointer"""
        # pointer to first node in SkipList at each level
        self._head = SkipListNode(None)
        self._head.next = [None]
        self._size = 0
        # probability that a node goes to the next level
        self._p = p
        if data:
            for x in data:
                self.add(x)
```

The number of paths through the list is the *height* of the list. As mentioned previously, all nodes in the list have height at least 1. The probability `p` is the percentage of nodes that have height at least two. Typically, we set `p` to be 0.5; each path is composed of approximately half the length of the path at the layer below. The `SkipList` maintains the heads of each of the paths through the list.

We will make use of a number of properties associated with skip lists:

```
@property
def p(self):
    """The probability that a node goes to the next level."""
    return self._p

@property
def height(self):
    """The highest level of an actual node in the SkipList."""
    return len(self._head.next)
def __len__(self):
    """The length of the list at level 0."""
    return self._size
```

Now, let's assume that there are several elements in the list. What is necessary to locate a particular value? The search begins along the highest level, shortest path. This path, `i`, connects only a few nodes, and helps us get to the approximate location. We walk along the path, finding the largest value on the path that is less than the value sought. This is the predecessor of our value (if it is in the list) at level `i`. Since this node is in every lower level path, we transfer the search to the next lower level, `i-1`. As we find the predecessor at each level, we save a reference to each node. If the list has to be modified—by removing the value sought, or adding a copy if it is missing—each of these nodes must have their respective paths updated.

```
    def _find(self, value):
        """Create a list of the nodes immediately to the left of where value should be located
        node = self._head
        predecessors = []
        for l in reversed(range(self.height)):
            # search for predecessor on level l
            nextNode = node.next[l]
            while (nextNode is not None) and (nextNode.data < value):
                (node,nextNode) = (nextNode,nextNode.next[l])
            # we've found predecessor at level l
            predecessors.append(node)
            # note that we continue from node's lower pointers
        predecessors.reverse()
        return predecessors
```

Notice that it is important that the search begin with the highest level path; going from the outer path to the innermost path guarantees that we find the predecessor nodes quickly. If we consider the paths from lower levels upward, we must begin the each of each path at the head; this does not guarantee a sub-linear performance.

How can we evaluate the performance of the `_find` operation? First, let's compute the approximate length of each path. The lowest level path (level 0) has n nodes. The second lowest path has pn nodes. Each successively higher path is reduced in length to $p^i n$ for the path at level i. Given this, the number of paths through the list is approximately $\log_{\frac{1}{p}} n$, the number of times that n can be reduced by p. Another way to think about this is that between each pair of nodes at level i, there are approximately $\frac{1}{p}$ nodes at level $i-1$. Now, a `_find` operation searches, in the expected case, about half the way through the top-most path. Once the appropriate spot is found, the search continues at the next level, considering some portion of the $\frac{1}{p}$ nodes at that level. This continues $\log_{\frac{1}{p}} n$ times. The result is a running time of $\frac{1}{p}\log_{\frac{1}{p}} n$, or $O(\log n)$ steps.

The result of `_find` is a list of references to nodes that are the largest values smaller than the value sought. Entry 0 in this list is a reference to the predecessor of the value sought. If the value that follows the node referenced by entry 0 in this list is not the value we seek, the value is not in the list.

```
def __contains__(self,value):
    """value is in SkipList."""
    location = self._find(value)
    target = location[0].next[0]
    return (target is not None) and (target.data == value)
```

Another way to think about the result is find is that it is a list of nodes whose links might be updated if the node sought was to be inserted. In the case of removals, these nodes are the values that must be updated when the value is removed.

The `add` method performs a `_find` operation which identifies, for each path, the node that would be the predecessor in a successful search for the value. A new node is constructed, its height is randomly selected (that is, the number of paths that will include the node). The insertion of the node involves constructing the `next` points for the new node; these pointers are simply the next references currently held by the predecessor nodes. For each of these levels it is also necessary to update the `next` pointers for the predecessors so they each point the our new node. Here is the code:

```
@mutatormethod
def add(self, value):
    """Insert a SkipListNode with data set to value at the appropriate location."""
    # create node
    node = SkipListNode(value)
    nodeHeight = self._random_height()
    for l in range(self.height,nodeHeight):
        self._head.next.append(None)
    # list of nodes whose pointers must be patched
    predecessors = self._find(value)
    # construct the pointers leaving node
    node.next = [predecessors[i].next[i] for i in range(nodeHeight)]
    for i in range(nodeHeight):
        predecessors[i].next[i] = node
    self._size += 1
```

It is possible to improve, slightly, the performance of this operation by merging the code of `_find` and `add` methods, but the basic complexity of the operation is not changed.

The `remove` operation begins with code that is similar to the `__contains__` method. If the value-holding node is found, `remove` replaces the `next` fields of the predecessors with references from the node targeted for removal:

```
@mutatormethod
def remove(self, value):
    """Remove value from SkipList at all levels where it appears."""
    predecessors = self._find(value)
    target = predecessors[0].next[0]
```

```
if (target is None) or (target.data != value):
    return None
height = len(target.next)
for i in range(height):
    predecessors[i].next[i] = target.next[i]
while (len(self._head.next) > 1) and (self._head.next[-1] is None):
    self._head.next.pop()
self._size -= 1
```

9.4 Performance Analysis

We have seen two different implementations of lists that keep their elements in order. One is based on Python's built-in `list` class, the other based on a linked structure. Although the linked structure has the overhead of possibly many pointers, the number of pointers per node asymptotically approaches $\frac{1}{1-p}$ for a skip list whose probability is p. For example, when $p = 2$, the skip list has a space utilization of a doubly linked list. The advantage of these extra pointers is that insertion and removal operations is reduced to the expected complexity of finding the appropriate location, $O(\log_{\frac{1}{p}} n)$. This is an *expected* cost because the structure of the list depends on the selection of nodes of random heights.

Insertion of values into either type of list involved identifying the appropriate location for the value to be inserted. In both of these structures, this operation takes $O(\log n)$ time. Insertion in a list requires the shifting of a potentially large number of values, but the `SkipList` requires the updating of several pointers. Not surprisingly, the performance of the `SkipList` is no better than the `SortedList` for small values of n, but as n increases, the $O(n)$ insertion time of `SortedList` loses out to the `SkipList`. We can see this in the tables of Figure **??**.

9.5 Conclusions

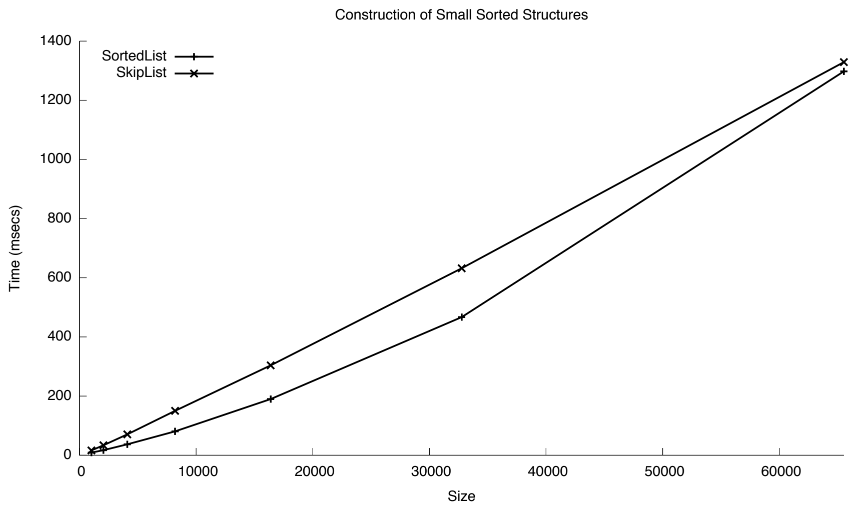


Figure 9.3 For small list sizes, `SortedLists` are constructed more quickly than `SkipLists`.

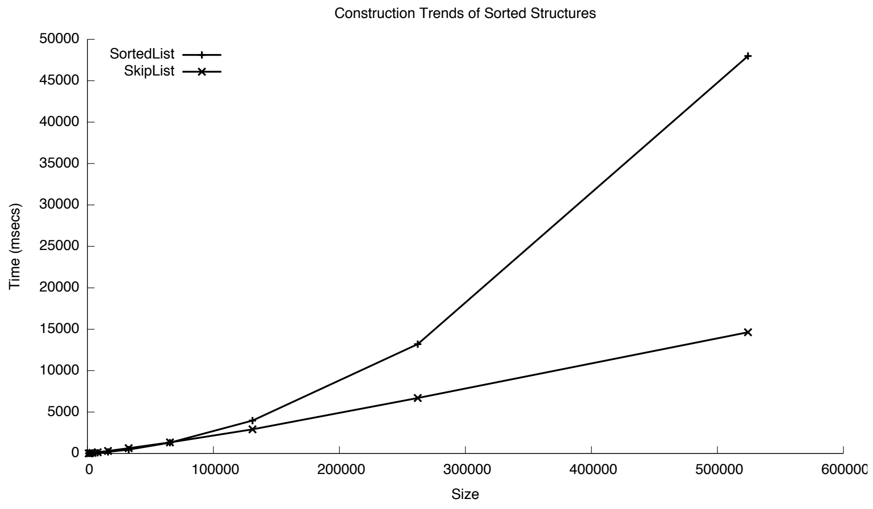


Figure 9.4 Based on insertion sort, `SortedList` construction is a $O(n^2)$ process, while `SkipList` construction is $O(n \log n)$

Chapter 10

Binary Trees

RECURSION IS A BEAUTIFUL APPROACH TO STRUCTURING. We commonly think of recursion as a form of structuring the *control* of programs, but self-reference can be used just as effectively in the structuring of program *data*. In this chapter, we investigate the use of recursion in the construction of branching structures called *trees*.

Most of the structures we have already investigated are *linear*—their natural presentation is in a line. Trees branch. The result is that where there is an inherent ordering in linear structures, we find choices in the way we order the elements of a tree. These choices are an indication of the reduced "friction" of the structure and, as a result, trees provide us with the fastest ways to solve many problems.

Before we investigate the implementation of trees, we must develop a concise terminology.

10.1 Terminology

A tree is a collection of elements, called *nodes*, and relations between them, called *edges*. Usually, data are stored within the nodes of a tree. Two trees are *disjoint* if no node or edge is found common to both. A *trivial tree* has no nodes and thus no data. An isolated node is also a tree.

From these primitives we may recursively construct more complex trees. Let r be a new node and let $T_1, T_2, \ldots, T_n$ be a (possibly empty) set—a *forest*—of distinct trees. A new tree is constructed by making r the root of the tree, and establishing an edge between r and the root of each tree, T_i, in the forest. We refer to the trees, T_i, as *subtrees*. We draw trees with the root above and the trees below. Figure **??**g is an aid to understanding this construction.

The *parent* of a node is the adjacent node appearing above it (see Figure **??**). The *root* of a tree is the unique node with no parent. The *ancestors* of a node n are the roots of trees containing n: n, n's parent, n's parent's parent, and so on. The root is the ancestor shared by every node in the tree. A *child* of a node n is any node that has n as its parent. The *descendants* of a node n are those nodes that have n as an ancestor. A *leaf* is a node with no children. Note that n is its own ancestor and descendant. A node m is the *proper ancestor* (*proper descendant*) of a node n if m is an ancestor (descendant) of n, but not vice versa. In a tree T, the descendants of n form the *subtree* of T rooted at n. Any node of a tree T that is not a leaf is an *interior node*. Roots can be interior nodes. Nodes

m and n are *siblings* if they share a parent.

A *path* is the unique shortest sequence of edges from a node n to an ancestor. The *length* of a path is the number of edges it mentions. The *height of a node* n in a tree is the length of any longest path between a leaf and n. The *height of a tree* is the height of its root. This is the maximum height of any node in the tree. The *depth* (or *level*) of a node n in its tree T is the length of the path from n to T's root. The sum of a node's depth and height is no greater than the height of the tree. The *degree of a node* n is the number of its children. The *degree of a tree* (or its *arity*) is the maximum degree of any of its nodes. A *binary tree* is a tree with arity less than or equal to 2. A 1-ary tree is termed *degenerate*. A node n in a binary tree is *full* if it has degree 2. In an *oriented tree* we will call one child the *left child* and the other the *right child*. A *full binary tree* is a tree whose internal nodes are all full. A *perfect binary tree* (or a *maximally heapable tree*) is a full tree of height h that has leaves only on level h. The addition of a node to a perfect binary tree causes its height to increase. A *complete binary tree* (or a *heapable tree*) of height h is a perfect binary tree with 0 or more of the rightmost leaves of level h removed.

10.2 Example: Pedigree Charts

With the growth of the Internet, many people have been able to make contact with long-lost ancestors, not through some new technology that allows contact with spirits, but through genealogical databases. One of the reasons that genealogy has been so successful on computers is that computers can organize treelike data more effectively than people.

One such organizational approach is a pedigree chart. This is little more than a binary tree of the relatives of an individual. The root is an individual, perhaps yourself, and the two subtrees are the pedigrees of your mother and father.[1] They, of course, have two sets of parents, with pedigrees that are rooted at your grandparents.

Steve points out: relationships in these trees is upside down!

To demonstrate how we might make use of a BinTree class, we might imagine the following code that develops the pedigree for someone named George Bush:[2]

```
JSBush = BinTree("Rev. James");
HEFay = BinTree("Harriet");
SPBush = BinTree("Samuel",JSBush,HEFay);
RESheldon = BinTree("Robert");
MEButler = BinTree("Mary");
FSheldon = BinTree("Flora",RESheldon,MEButler);
PSBush = BinTree("Prescott",SPBush,FSheldon);
DDWalker = BinTree("David");
```

[1] At the time of this writing, modern technology has not advanced to the point of allowing nodes of degree other than 2.

[2] This is the Texan born in Massachusetts; the other Texan was born in Connecticut.

```
MABeaky = BinTree("Martha");
GHWalker = BinTree("George",DDWalker,MABeaky);
JHWear = BinTree("James II");
NEHolliday = BinTree("Nancy");
LWear = BinTree("Lucretia",JHWear,NEHolliday);
DWalker = BinTree("Dorothy",GHWalker,LWear);
GHWBush = BinTree("George",PSBush,DWalker);
```

For each person we develop a node that either has no links (the parents were not included in the database) or has references to other pedigrees stored as `BinTree`s. Arbitrarily, we choose to maintain the father's pedigree on the left side and the mother's pedigree along the right. We can then answer simple questions about ancestry by examining the structure of the tree. For example, who are the direct female relatives of the President?

```
person = GHWBush
while not person.right.empty:
    person = person.right
    print(person.value)
```

The results are

```
Dorothy
Lucretia
Nancy
```

Exercise 10.1 *These are, of course, only some of the female relatives of President Bush. Write a program that prints* all *the female names found in a* `BinTree` *representing a pedigree chart.*

One feature that would be useful, would be the ability to add branches to a tree after the tree was constructed. For example, we might determine that James Wear had parents named William and Sarah. The database might be updated as follows:

```
JHWear.left = BinTree("William")
SAYancey = BinTree("Sarah")
JHWear.right = SAYancey
```

A little thought suggests a number of other features that might be useful in supporting the pedigree-as-`BinTree` structure.

10.3 Example: Expression Trees

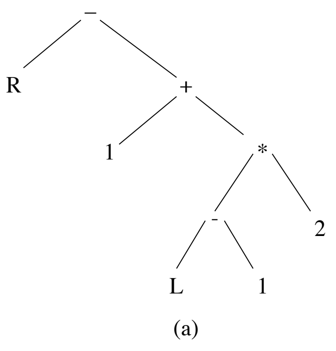
Most programming languages involve mathematical expressions that are composed of binary operations applied to values. An example from Python programming is the simple expression `r = 1 + (l - 1) * 2`. This expression involves four operators (`=`, `+`, `-`, and `*`), and 5 values (`r`, `1`, `l`, `1`, and `2`). Languages often represent expressions using binary trees. Each value in the expression appears as a leaf, while the operators are internal nodes that represent the reduction of two values to one (for example, `l - 1` is reduced to a single value for use on the left side of the multiplication sign). The *expression tree* associated with our expression is shown in Figure 10.1a. We might imagine that the following code constructs the tree and prints -1:

```
# two variables, l and r, both initially zero
l = BinTree(["l",0])
r = BinTree(["r",0])

# compute r = 1+(l-1)*2
t = BinTree(operator('-'),l,BinTree(1))
t = BinTree(operator('*'),t,BinTree(2))
t = BinTree(operator('+'),BinTree(1),t)
t = BinTree(operator('='),r,t)
print(eval(t))

# evaluate and print expression
print(r.value[1])
```

Once an expression is represented as an expression tree, it may be evaluated by *traversing* the tree in an agreed-upon manner. Standard rules of mathematical precedence suggest that the parenthesized expression (`l-1`) should be evaluated first. (The `l` represents a value previously stored in memory.) Once the subtraction is accomplished, the result is multiplied by 2. One is then then added to the product. The result of the addition is assigned to `r`. The assignment operator is treated in a manner similar to other common operators; it just has lower *precedence* (it is evaluated later) than standard mathematical operators. Thus an implementation of binary trees would be aided by a traversal mechanism that allows us to manipulate values as they are encountered.

10.4 Implementation

We now consider the implementation of binary trees. As with our linked list implementations, we will construct a self-referential binary tree class, `BinTree`. The recursive design motivates implementation of many of the `BinTree` operations as recursive methods. However, because the base case of recursion often involves an empty tree we will make use of a dedicated “sentinel” node that represents the empty tree. This simple implementation will be the basis of a large number of more advanced structures we see throughout the remainder of the text.



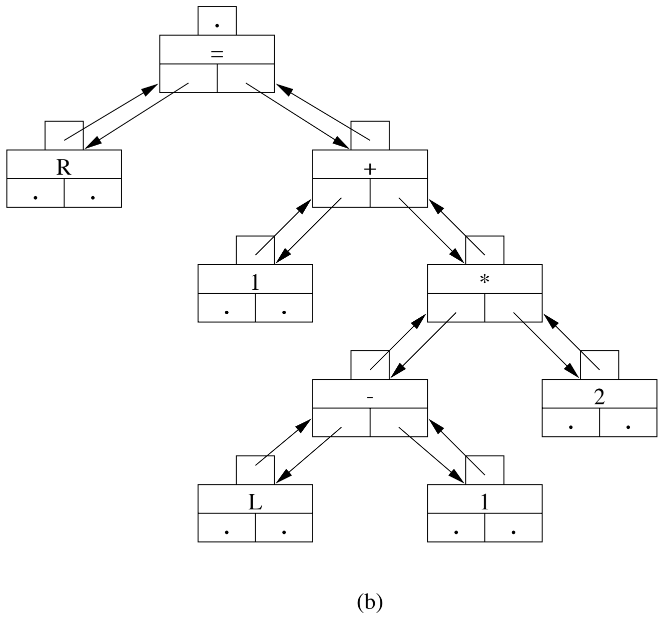


Figure 10.1 Expression trees. (a) An abstract expression tree representing `r=1+(l-1)*2`. (b) A possible connectivity of an implementation using references.

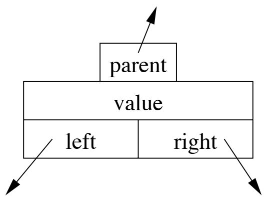


Figure 10.2 The structure of a `BinTree`. The parent reference opposes a left or right child reference in parent node.

10.4.1 The BinTree Implementation

Our first step toward the development of a binary tree implementation is to represent an entire subtree as a reference to its root node. The node will maintain a reference to user data and related nodes (the node's parent and its two children) and directly provides methods to maintain a subtree rooted at that node. All empty trees will be represented by a single instance of a `BinTree` called `BinTree.Empty`. This approach is not unlike the "dummy nodes" provided in our study of linked lists. Allowing the empty tree to be an object allows programs to call methods on trees that are empty. If the empty tree were represented by `None`, it would be impossible to apply methods to the structure.

Figure 10.1b depicts the use of `BinTree`s in the representation of an entire tree. We visualize the structure of a `BinTree` as in Figure 10.2. To construct such a node, we require three pieces of information: a reference to the data that the user wishes to associate with this node, and left and right references to binary tree nodes that are roots of subtrees of this node. The parent reference is determined implicitly from opposing references. The `BinTree` is initialized as follows:

```
@checkdoc
class BinTree(Iterable, Sized, Freezable):
    # This object represents the empty tree
    # All empty references are to self.Empty
    # Initialization happens after class definition
    Empty = None

    __slots__ = ["_data", "_left", "_right", "_parent", "_frozen", "_hash"]

    def __init__(self,data=None,left=None,right=None,dummy=False,frozen=False):
        """Construct a binary tree node."""
        self._hash = None
        if dummy:
            # this code is typically used to initialize self.Empty
            self._data = None
            self._left = self
            self._right = self
            self._parent = None
            self._frozen = True
        else:
            # first, declare the node as a disconnected entity
            self._left = None
            self._right = None
            self._parent = None
            self._data = data
            self._frozen = False
            # now, nontrivially link in subtrees
            self.left = left if left is not None else self.Empty
            self.right = right if right is not None else self.Empty
            if frozen:
                self.freeze()
```

We call the initializer with `dummy` set to `True` when an empty `BinTree` is needed. The result of this initializer is the single empty node that will represent members of the fringe of empty trees found along the edge of the binary tree. The most common initialization occurs when `dummy` is `False`. Here, the initializer makes calls to "setting" routines. These routines allow one to set the references of the left and right subtrees, but also ensure that the children of this node reference this node as their parent. This is the direct cost of implementing forward and backward references along every link. The return, though, is the considerable simplification of other code within the classes that make use of `BinTree` methods.

Principle 6 *Don't let opposing references show through the interface.*

When maintenance of opposing references is left to the user, there is an opportunity for references to become inconsistent. Furthermore, one might imagine implementations with fewer references (it is common, for example, to avoid the parent reference); the details of the implementation should be hidden from the user, in case the implementation needs to be changed.

Here is the code associated with maintaining the left subtree as a property associated with `BinTree`. As is typical, the accessor property method simply returns the internal reference, while the "setter" performs a non-trivial amount of work to maintain a consistent state in the tree.

```
@property
def left(self):
    """This node's left child."""
    return self._left

@left.setter
def left(self,newLeft):
    """Hang new left tree off node."""
    assert isinstance(newLeft,BinTree)
    if self.empty:
        return
    # carefully dissolve any relation to left subtree
    if self.left is not None and self.left.parent is self:
        self.left.parent = None
    self._left = newLeft
    newLeft.parent = self
```

If the setting of the left child causes a subtree to be disconnected from this node, and that subtree considers this node to be its parent (if the subtree is not the empty tree, it should), we disconnect the node by setting its parent to `None`. We then set the left child reference to the value passed in. Any dereferenced node sets its `parent` reference to `None`. We also take care to set the opposite parent reference by calling the `parent` setter method of the root of the associated non-trivial tree. Why? Because it is important that we not set the parent field of `BinTree.Empty`. Because we want to maintain consistency between the downward child references and the upward parent references, we declare the actual reference, `_parent` with a leading underscore to make it difficult for the user to refer to directly:

```
    @property
    def parent(self):
        """This node's parent."""
        return self._parent

    @parent.setter
    def parent(self,newParent):
        if self is not self.Empty:
            self._parent = newParent
```

Once the node has been constructed, its value can be inspected and modified using the value-based functions that parallel those we have seen with other types:

```
    @property
    def value(self):
        """The data stored in this node."""
        return self._data

    @value.setter

    def value(self,newValue):
        self._data = newValue
```

The `BinTree` may be used as the basis for our implementation of some fairly complex programs and structures.

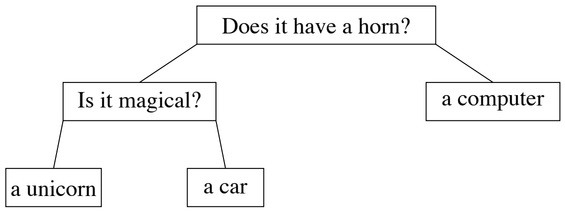


Figure 10.3 The state of the database in the midst of playing `infinite_questions`.

10.5 Example: An Expert System

Anyone who has been on a long trip with children has played the game Twenty Questions. It's not clear why this game has this name, because the questioning often continues until the entire knowledge space of the child is exhausted. We can develop a very similar program, here, called `infinite_questions`. The central component of the program is a database, stored as a `BinTree`. At each leaf is an object that is a possible guess. The interior nodes are questions that help distinguish between two groups of guesses; these nodes are full.

Figure 10.3 demonstrates one possible state of the database. To simulate a questioner, one asks the questions encountered on a path from the root to some leaf. If the response to the question is positive, the questioning continues along the left branch of the tree; if the response is negative, the questioner considers the right.

Exercise 10.2 *What questions would the computer ask if you were thinking of a truck?*

Of course, we can build a very simple database with a single value—perhaps a computer. The game might be set up to play against a human as follows:

```
def main():
    database = BinTree("a computer")
    answer = input("Do you want to play a game? ")
    while answer == "yes":
        print("Think of something...I'll guess it.")
        play(database)
        answer = input("Do you want to play again? ")
    print("Have a good day!")
```

When the game is played, the computer is very likely to lose. The database can still benefit by incorporating information about the losing situation. If the program guessed a computer and the item was a car, we could incorporate the car and a question "Does it have wheels?" to distinguish the two objects. As it turns out, the program is not that difficult.

```
def play(database):
    if not database.left.empty:
        # ask a question
        answer = input(database.value+" ")
        if answer == "yes":
            play(database.left)
        else:
            play(database.right)
    else:
        # make a guess
        answer = input("Is it "+database.value+"? ")
        if answer == "yes":
            print("I guessed it!")
        else:
            answer = input("Darn!  What were you thinking of? ")
            newObject = BinTree(answer)
            oldObject = BinTree(database.value)
            database.left = newObject
            database.right = oldObject
            database.value = \
                input("What question would distinguish {} from {}? "\
                          .format(newObject.value,oldObject.value))
```

The program can distinguish questions from guesses by checking to see there is a left child. This situation would suggest this node was a question since the two children need to be distinguished.

The program is very careful to expand the database by adding new leaves at the node that represents a losing guess. If we aren't careful, we can easily corrupt the database by growing a tree with the wrong topology.

Here is the output of a run of the game that demonstrates the ability of the database to incorporate new information—that is to *learn*:

```
Do you want to play a game?
Think of something...I'll guess it
Is it a computer?
Darn.  What were you thinking of?
What question would distinguish a car from a computer?
Do you want to play again?
Think of something...I'll guess it
Does it have a horn?
Is it a car?
Darn.  What were you thinking of?
What question would distinguish a unicorn from a car?
Do you want to play again?
Think of something...I'll guess it
Does it have a horn?
Is it magical?
Is it a car?
I guessed it!
Do you want to play again?
Have a good day!
```

Exercise 10.3 *Make a case for or against this program as a (simple) model for human learning through experience.*

We now discuss the implementation of a general-purpose `Iterator` for the `BinTree` class. Not surprisingly a structure with branching (and therefore a choice in traversal order) makes traversal implementation more difficult. Next, we consider the construction of several `Iterator`s for binary trees.

10.6 Recursive Methods

Rewrite this section

Trees are recursively defined structures, so it would seem reasonable to consider recursive implementations of iterators. The difficulty is that generators must maintain their state whenever a `yield` occurs. Any recursive approach to traversal would encounter nodes while deep in recursion, and the state of the generator's call stack would have to be preserved. This is so problematic that Python actively discourages the use of recursion in generator specification.

One way around the difficulties of suspending the recursion is to initially perform the entire traversal, generating a `list` of values encountered. Since the entire traversal happens all at once, the list can be constructed during the recursion.

Using this idea, we can reconsider the in-order traversal:

```
def inOrderR(self):
    def listify(tree,lst):
        if not tree.empty:
            listify(tree.left,lst)
            lst.append(tree.value)
            listify(tree.right,lst)

    temp = []
    listify(self,temp)
    return iter(temp)
```

The core of this implementation is the protected method `enqueueInorder`. It simply traverses the tree rooted at its parameter and enqueues every node encountered. Since it recursively enqueues all its left descendants, then itself, and then its right descendants, it is an in-order traversal. Since the queue is a FIFO, the order is preserved and the elements may be consumed at the user's leisure.

Exercise 10.4 *Rewrite the other iteration techniques using the recursive approach.*

It is reassuring to see the brevity of these implementations. Unfortunately, while the recursive implementations are no less efficient, they come at the obvious cost of a potentially long delay whenever the iterator is primed. In addition, if the height of the tree to be traversed is greater than Python's stack limit, the recursion is doomed. For this reason, the iterative approaches seem most resilient. Still, for many applications recursion may be satisfactory.

10.7 Traversals of Binary Trees

We have seen, of course, there is a great industry in selling calculators that allow users to enter expressions in what appear to be arbitrary ways. For example, some calculators allow users to specify expressions in *infix* form, where keys associated with operators (+ and the like) are pressed between the values they operate on, operands. Other calculators advocate a *postfix*[3] form, where the operator is pressed only after the operands have been entered. Considering our representation of expressions as trees, we observe that there must be a similar variety in the ways we traverse a `BinTree` structure. We consider those here.

When designing iterators for linear structures there are usually few useful choices: start at one end and visit each element until you get to the other end. Many of the linear structures we have seen provide an `elements` method that constructs an iterator for traversing the structure. For binary trees, there is no obviously preferred order for traversing the structure. Here are four rather distinct mechanisms:

Preorder traversal. Each node is visited before any of its children are visited. Typically, we visit a node, and then each of the nodes in its left subtree, followed by each of the nodes in the right subtree. A *preorder traversal* of the expression tree in the margin visits the nodes in the order: $=$, r, $+$, 1, $*$, $-$, l, 1, and 2.

In-order traversal. Each node is visited after all the nodes of its left subtree have been visited and before any of the nodes of the right subtree. The *in-order* traversal is usually only useful with binary trees, but similar traversal mechanisms can be constructed for trees of arbitrary arity. An in-order traversal of the expression tree visits the nodes in the order: r, $=$, 1, $+$, l, $-$, 1, $*$, and 2. Notice that, while this representation is similar to the expression that actually generated the binary tree, the traversal has removed the parentheses.

Postorder traversal. Each node is visited after its children are visited. We visit all the nodes of the left subtree, followed by all the nodes of the right subtree, followed by the node itself. A *postorder traversal* of the expression

[3] Reverse Polish Notation (RPN) was developed by Jan Lukasiewicz, a philosopher and mathematician of the early twentieth century, and was made popular by Hewlett-Packard in their calculator wars with Texas Instruments in the early 1970s.

tree visits the nodes in the order: r, 1, l, 1, $-$, 2, $*$, $+$, and $=$. This is precisely the order that the keys would have to be pressed on a "reverse Polish" calculator to compute the correct result.

Level-order traversal. All nodes of level i are visited before the nodes of level $i+1$. In a *Level-order traversal* the nodes of the expression tree are visited in the order: $=$, r, $+$, 1, $*$, $-$, 2, l, and 1. (This particular ordering of the nodes is motivation for another implementation of binary trees we shall consider later and in Problem **??**.)

As these are the most common and useful techniques for traversing a binary tree we will investigate their respective implementations. Traversing `BinTrees` involves constructing an iterator that traverses the entire set of subtrees. For this reason, and because the traversal of subtrees proves to be just as easy, we discuss implementations of iterators for `BinTrees`.

Most implementations of iterators maintain a linear structure that keeps track of the state of the iterator. In some cases, this auxiliary structure is not strictly necessary (see Problem **??**) but may reduce the complexity of the implementation and improve its performance.

10.7.1 Preorder Traversal

For a preorder generator, we wish to yield each node of the tree before any of its proper descendants (recall the node is a descendant of itself). To accomplish this, we keep a local stack of nodes whose right subtrees have not been investigated. In particular, the current node is the topmost element of the stack, and elements stored deeper within the stack are more distant ancestors.

The iterator accepts a single parameter—the root of the subtree to be traversed. An important thing to note is that the traversal from a node should stay within the subtree rooted at that node.

```
def preOrder(self):
    """Traverse subtree by visiting parent before children."""
    def addNode(node,todo):
        if not node.empty:
            todo.add(node)

    todo = Stack()
    addNode(self,todo)
    while not todo.empty:
        tree = todo.pop()
        yield tree.value
        addNode(tree.right,todo)
        addNode(tree.left,todo)
```

As we can see, `todo` is the private stack used to keep track of references to unvisited nodes encountered while visiting other nodes. Another way to think about it is that it is the frontier of nodes encountered on paths from the root that have not yet been visited.

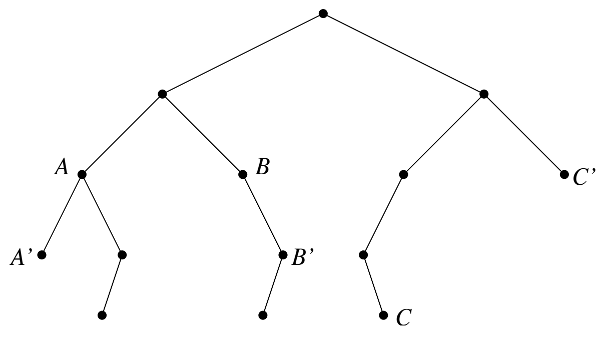


Figure 10.4 Three cases of determining the next current node for preorder traversals. Node A has a left child A' as the next node; node B has no left, but a right child B'; and node C is a leaf and finds its closest, "right cousin," C'.

Whenever we encounter an unvisited node, directly, or as a child of the a visited node, we push it on the stack of nodes to be considered. The traversal, then involves popping nodes off the todo list, and yielding the value. Since the current node has just been visited, we push on any children of the node—first, a possible right child, then a possible left. If the node has a left child (see node A of Figure 10.4), that node (A') is the next node to be visited. If the current node (see node B) has only a right child (B'), it will be visited next. If the current node has no children (see node C), the effect is to visit the closest unvisited right cousin or sibling (C').

It is clear that over the life of the iterator each of the n values of the tree is pushed onto and popped off the stack exactly once; thus the total cost of traversing the tree is $O(n)$. A similar observation is possible for each of the remaining iteration techniques.

10.7.2 In-order Traversal

The inorder iterator is constructed from a generator that visits left children of a node, then the node, then the node's right children. Again, we construct the iterator using a generator that keeps a stack of unvisited ancestors of the node currently being visited. The stack is initialized by pushing on the root of the tree and all the descendants along the leftmost branch.

When a node is popped off the `todo` list it either has no left subtree or all the nodes of the left subtree have been visited. The node is then visited. After the visit, we traverse the right subtree by pushing on its left branch. Here's the process:

```
def inOrder(self):
    def pushLeftDescendants(tree,todo):
        while not tree.empty:
            todo.push(tree)
            tree = tree.left

    todo = Stack()
    pushLeftDescendants(self,todo)
    while not todo.empty:
        tree = todo.pop()
        yield tree.value
        pushLeftDescendants(tree.right,todo)
```

Since the first element considered in an in-order traversal is the leftmost descendant of the root, pushing each of the nodes from the root down to the leftmost descendant places that node on the top of the stack.

When the current node is popped from the stack, the next element of the traversal must be found. We consider two scenarios:

1. If the current node has a right subtree, the nodes of that tree have not been visited. At this stage we should push the right child, and all the nodes down to and including its leftmost descendant, on the stack.

2. If the node has no right child, the subtree rooted at the current node has been fully investigated, and the next node to be considered is the closest unvisited ancestor of the former current node—the node just exposed on the top of the stack.

As we shall see later, it is common to order the nodes of a binary tree so that left-hand descendants of a node are smaller than the node, which is, in turn, smaller than any of the rightmost descendants. In such a situation, the in-order traversal plays a natural role in presenting the data of the tree in order. For this reason, the `__iter__` method returns the iterator formed from the he `inOrder` generator.

10.7.3 Postorder Traversal

Traversing a tree in postorder also maintains a stack of uninvestigated nodes. Each of the elements on the stack is a node whose descendants are currently being visited. Since the first element to be visited is the leftmost descendant of the root, the generator begins by pushing on each of the nodes from the root to the leftmost leaf using the internal method, `pushToLeftmostLeaf`. If this procedure is called on an empty binary tree, no nodes are actually pushed onto the stack. An important observation is that the `todo` stack always contains all of the (unvisited) nodes from the current (visited) node to the root of the traversal.

```
    def postOrder(self):
        def pushToLeftLeaf(tree,todo):
            while not tree.empty:
                todo.push(tree)
                if not tree.left.empty:
                    tree = tree.left
                else:
                    tree = tree.right

        todo = Stack()
        pushToLeftLeaf(self,todo)
        while not todo.empty:
            node = todo.pop()
            yield node.value
            if not todo.empty:
                # parent is on top (todo.peek())
                parent = todo.peek()
                if node is parent.left:
                    pushToLeftLeaf(parent.right,todo)
```

Once the stack has been initialized, the top element on the stack is the leftmost leaf, the first value to be visited. The node is popped from the stack and yielded. Now, if a node has just been traversed (as indicated by the yield statement), it and its descendants must all have been visited thus far in the traversal. Either the todo stack is empty (we're currently at the root of the traversal) or the top of the stack is the parent of the current node. If the current node is its parent's left child, we speculatively continue the search through the parent's right subtree. If there is no right subtree, or if the current node is a right child, no further nodes are pushed onto the stack and the next iteration will visit the parent node.

Since the `todo` stack simply keeps track of the path to the root of the traversal the stack can be dispensed with by simply using the nodes that are singly linked through the parent pointers. One must be careful, however, to observe two constraints on the stackless traversal. First, one must make sure that traversals of portions of a larger tree are not allowed to "escape" through self, the virtual root of the traversal. Second, since there is one `BinTree.Empty`, we have carefully protected that node from being "re-parented". It is important that the traversal not accidentally step off of the tree being traversed into the seemingly disconnected empty node.

10.7.4 Level-order Traversal

A level-order traversal visits the root, followed by the nodes of level 1, from left to right, followed by the nodes of level 2, and so on. This can be easily accomplished by maintaining a queue of the next few nodes to be visited. More precisely, the queue contains the current node, followed by a list of all siblings and cousins to the right of the current node, followed by a list of "nieces and nephews" to the left of the current node.

```
    def levelOrder(self):
        def addNode(node,todo):
            if not node.empty:
                todo.add(node)
        todo = Queue()
        addNode(self,todo)
        while not todo.empty:
            tree = todo.dequeue()
            yield tree.value
            addNode(tree.left,todo)
            addNode(tree.right,todo)
```

The traversal is initialized by adding the root of the search to the queue (presuming the traversed subtree is not empty). The traversal step involves removing a node from the queue. Since a queue is a first-in first-out structure, this is the node that has waited the longest since it was originally exposed and added to the queue. After we visit a node, we enqueue the children of the node (first the left the the right). With a little work it is easy to see that these are either nieces and nephews or right cousins of the next node to be visited; it is impossible for a node *and* its descendants to be resident in the queue at the same time.

Unlike the other iterators, this method of traversing the tree is meaningful regardless of the degree of the tree. On the other hand, the use of the queue, here, does not seem to be optional; the code would be made considerably more complex if it were recast without the queue.

In each of these iterators we have declared utility functions (`addNode`, etc.) that have helped us describe the logical purpose behind the management of the linear `todo` structure. There is, of course, some overhead in declaring these functions, but from a code-clarity point-of-view, these functions help considerably in understanding the motivations behind the algorithm. The method `addNode` is common to two different traversals (`preOrder` and `levelOrder`), but the *type of the linear structure was different*. The use of the `add` method instead of the structure specific equivalents (`push` and `enqueue`) might allow us a small degree of conceptual code reuse.

10.8 Property-Based Methods

At this point, we consider the implementation of a number of property-based methods. Properties such as the height and fullness of a tree are important to guiding updates of a tree structure. Because the binary tree is a recursive structure, the proofs of tree characteristics (and the methods that verify them) often have a recursive feel. To emphasize the point, in this section we allow theorems about trees and methods that verify them to intermingle. Again, the methods described here are written for use on `BinTree`s, but they are easily adapted for use with more complex structures.

Our first method makes use of the fact that the root is a common ancestor of every node of the tree. Because of this fact, given a `BinTree`, we can identify the node as the root, or return the root of the tree containing the node's parent.

```
@property
def root(self):
    """Root of containing tree."""
    result = self
    while result.parent != None:
        result = result.parent
    return result
```

If we count the number of times the `root` routine is recursively called, we compute the number of edges from the node to the root—the depth of the node. A more general purpose method, `pathLength` allows us to count the number of edges along a path from a node to an ancestor.

```
def pathLength(self,ancestor):
    """Count edges between a node and ancestor."""
    if self.empty:
        return -1
    length = 0
    node = self
    while node is not ancestor:
        node = node.parent
        length += 1
    return length
```

Remembering that the root of a tree is an ancestor for every node in the tree, we can determine the depth by computing the length of the path from a node to its root.

```
@property
def depth(self):
    """Depth of the node in the tree."""
    return self.pathLength(self.root)
```

The time it takes is proportional to the depth of the node, though it follows that path twice: once to compute the root and a second time to count the edges. In cases where the root is known, or we are seeking the depth of a node in a rooted subtree, there is a computational advantage to avoiding the root computation. Notice that in the empty case we return a height of -1. This is consistent with our recursive definition, even if it does seem a little unusual. Often, making tough decisions about base cases can play an important role in making your interface useful. Generally, a method is more robust, and therefore more usable, if you handle as many cases as possible.

Principle 7 *Write methods to be as general as possible.*

Having computed the depth of a node, it follows that we should be able to determine the height of a tree rooted at a particular `BinTree`. We know that the height is simply the length of a longest path from the root to a leaf. If we perform a level-order traversal of the tree, the last node encountered is found

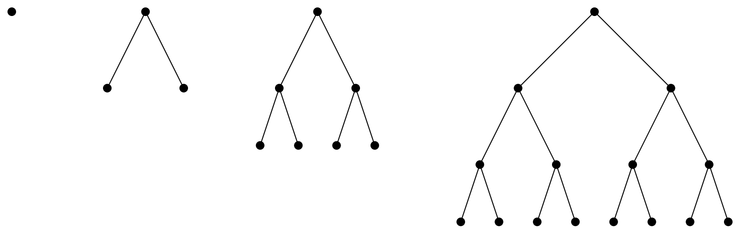


Figure 10.5 Several perfect (and complete) binary trees.

on the last level. The depth of this node below the root of the traversal is the height of the tree.

To facilitate this computation we make use of a private level-order traversal, `_nodes`, that yields not values, but `BinTree` nodes:

```
def _nodes(self):
    """A level-order traversal of the binary tree nodes in this tree."""
    def addNode(node,todo):
        if not node.empty:
            todo.add(node)

    todo = Queue()
    addNode(self,todo)
    while not todo.empty:
        node = todo.remove()
        yield node
        addNode(node.left,todo)
        addNode(node.right,todo)
```

Traversing a tree with n nodes this traversal takes $O(n)$ time and the size of the `todo` structure may grow to include references to as many as $\lceil \frac{n}{2} \rceil$ nodes.

Exercise 10.5 *Rewrite the* `levelOrder` *generator based on the* `_nodes` *traversal.*

Now, with the `_nodes` generator, we can write the `height` property by returning the depth of the last node encountered.

```
@property
def height(self):
    """Height of the tree rooted at this node."""
    if self.empty:
        return -1
    node = list(self._nodes())[-1]
    # how far below self is the last lowest node?
    return node.pathLength(self)
```

Observation 10.1 *A perfect binary tree of height $h \geq 0$ has $2^{h+1} - 1$ nodes.*

Proof: We prove this by induction on the height of the tree. Suppose the tree has height 0. Then it has exactly one node, which is also a leaf. Since $2^1 - 1 = 1$, the observation holds, trivially.

Our inductive hypothesis is that perfect trees of height $k < h$ have $2^{k+1} - 1$ nodes. Since $h > 0$, we can decompose the tree into two perfect subtrees of height $h - 1$, under a common root. Each of the perfect subtrees has $2^{(h-1)+1} - 1 = 2^h - 1$ nodes, so there are $2(2^h - 1) + 1 = 2^{h+1} - 1$ nodes. This is the result we sought to prove, so by induction on tree height we see the observation must hold for all perfect binary trees. ⋄

This observation suggests that if we can compute the height and size of a tree, we have a hope of detecting a perfect tree by comparing the height of the tree to the number of nodes:

```
@property
def perfect(self):
    if self.empty:
        return True
    nodes = list(self._nodes())
    last = nodes[-1]
    height = last.pathLength(self)
    return (1<<(height+1))-1 == len(nodes)
```

Notice the return statement makes use of shifting 1 to the left by `height+1` binary places. This is equivalent to computing `2**(h+1)`. The result is that the function can detect a perfect tree in the time it takes to traverse the entire tree, $O(n)$.

There is one significant disadvantage, though. If you are given a tree with height greater than 100, the result of the return statement may have to compute very large values for reasonably small trees.

Exercise 10.6 *Rewrite the* `perfect` *method so that it avoids the manipulation of large values for small trees.*

We now prove some useful facts about binary trees that help us evaluate performance of methods that manipulate them. First, we consider a pretty result: if a tree has lots of leaves, it must branch in lots of places.

Observation 10.2 *The number of full nodes in a binary tree is one less than the number of leaves.*

Proof: Left to the reader.⋄

With this result, we can now demonstrate that just over half the nodes of a perfect tree are leaves.

Observation 10.3 *A perfect binary tree of height $h \geq 0$ has 2^h leaves.*

Proof: In a perfect binary tree, all nodes are either full interior nodes or leaves. The number of nodes is the sum of full nodes F and the number of leaves l. Since, by Observation 10.2, $F = L - 1$, we know that the count of nodes is $F + L = 2L - 1 = 2^{h+1} - 1$. This leads us to conclude that $L = 2^h$ and that $F = 2^h - 1$. $\diamond$

This result demonstrates that for many simple tree methods half of the time is spent processing leaves.

Using the `_nodes` traversal we can identify full trees by rejecting any tree that contains a node with one child:

```
@property
def full(self):
    """All nodes are parents of 2 children or are leaves"""
    for node in self._nodes():
        if node.degree == 1:
            return False
    return True
```

Since this procedure can be accomplished by simply looking at all of the nodes, it runs in $O(n)$ time on a tree with n nodes.

We can recognize a complete binary tree by performing a level-order traversal and realizing that any full nodes appear before any leaves. Between these two types of nodes is potentially one node with just a left subtree.

```
@property
def complete(self):
    """Leaves of tree at one or two levels, deeper leaves at left."""
    allowableDegree = 2
    for node in self._nodes():
        degree = node.degree
        if degree > allowableDegree:
            return False
        if degree < 2:
            if not node.right.empty:
                return False
            allowableDegree = 0
    return True
```

Again, because all of the processing can be accomplished in a single pass of all of the nodes, a complete tree can be identified in $O(n)$ time.

10.9 Example: Huffman Compression

Information within machines is stored as a series of `bits`, or 1's and 0's. Because the distribution of the patterns of 1's and 0's is not always uniform, it is possible to compress the bit patterns that are used and reduce the amount of storage that is necessary. For example, consider the following 56 character phrase:

```
The mountains, the mountains, we greet them with a song!
```

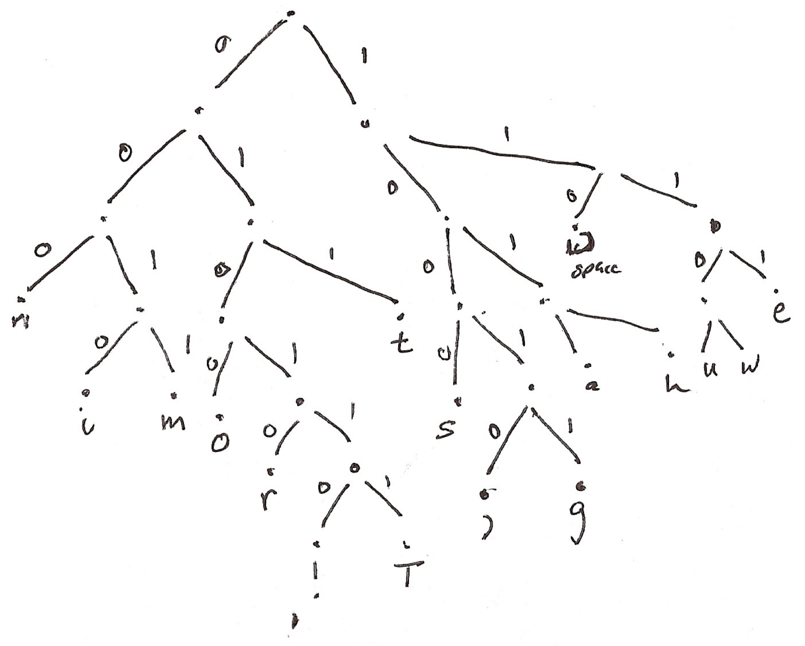


Figure 10.6 The Mountains huffman tree. Leaves are labeled with the characters they represent. Paths from root to leaves provide huffman bit strings.

If each letter in the string is represented by 8 bits (as they often are), the entire string takes 256 bits of storage. Clearly this catchy phrase does not use the full range of characters, and so perhaps 8 bits are not needed. In fact, there are 13 distinct characters so 4 bits would be sufficient (4 bits can represent any of 16 values). This would halve the amount of storage required, to 128 bits.

If each character were represented by a unique *variable-length* string of bits, further improvements are possible. *Huffman encoding* of characters allows us to reduce the size of this string to only 111 bits by assigning frequently occurring letters (like “o”) short representations and infrequent letters (like “a”) relatively long representations.

Huffman encodings can be represented by binary trees whose leaves are the characters to be represented. In Figure 10.6 left edges are labeled 0, while right edges are labeled 1. Since there is a unique path from the root to each leaf, there is a unique sequence of 1’s and 0’s encountered as well. We will use the string of bits encountered along the path to a character as its representation in the compressed output. Note also that no string is a prefix for any other (otherwise one character would be an ancestor of another in the tree). This means that,

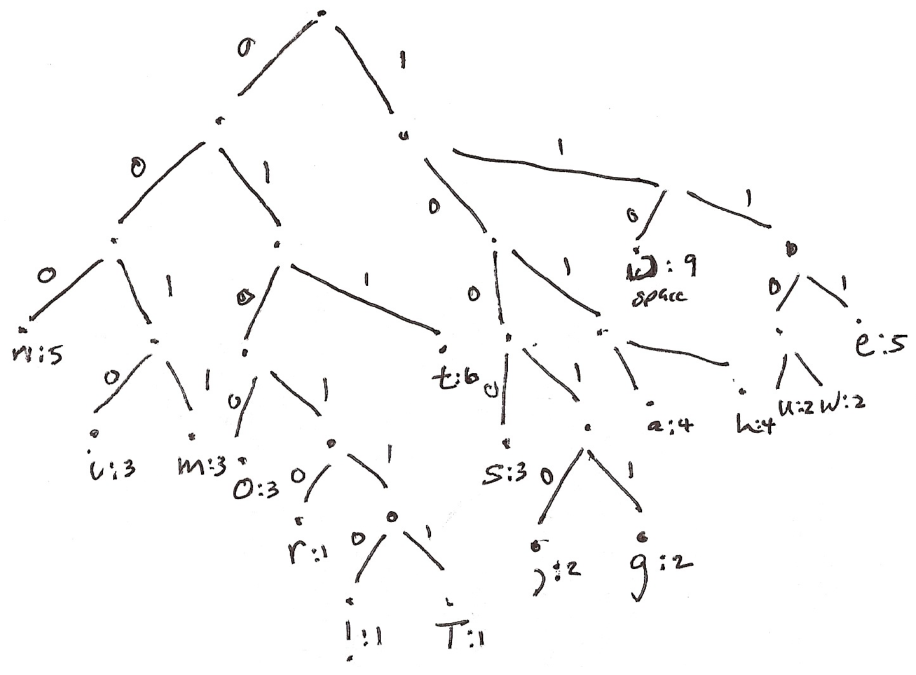


Figure 10.7 The Huffman tree of Figure 10.6, but with nodes labeled by total frequencies of descendant characters.

given the Huffman tree, decoding a string of bits involves simply traversing the tree and writing out the leaves encountered.

The construction of a Huffman tree is an iterative process. Initially, each character is placed in a Huffman tree of its own. The weight of the tree is the frequency of its associated character. We then iteratively merge the two most lightweight Huffman trees into a single new Huffman tree whose weight is the sum of weights of the subtrees. This continues until one tree remains. One possible tree for our example is shown in Figure 10.7.

Our approach is to use `BinTrees` to maintain the structure. This allows the use of recursion and easy merging of trees. Leaves of the tree carry descriptions of characters and their frequencies:

```
@total_ordering
class leaf(object):
    __slots__ = ["frequency","ch"]
    def __init__(self,c):
        self.ch = c
        self.frequency = 1

    def __lt__(self,other):
        return self.ch < other.ch

    def __eq__(self,other):
        return self.ch == other.ch

    def __str__(self):
        return "leaf({} with frequency {})".format(self.ch,self.frequency)
```

Intermediate nodes carry no data at all. Their relation to their ancestors determines their portion of the encoding. The entire tree is managed by a wrapper class, `huffmanTree`:

```
@total_ordering
class huffmanTree(object):
    __slots__ = ["root","totalWeight"]

    def __init__(self, l = None, left = None, right = None):
        if l is not None:
            self.root = BinTree(l)
            self.totalWeight = l.frequency
        else:
            self.totalWeight = left.totalWeight + right.totalWeight
            self.root = BinTree(left.root.value,left.root,right.root)

    def __lt__(self,other):
        return (self.totalWeight < other.totalWeight) or ((self.totalWeight == other.totalW

    def __eq__(self,other):
        return (self.totalWeight == other.totalWeight) and (self.root.value == other.root.v

    def print(self):
        printTree(self.root,"")

    def __str__(self):
        if self.root.height == 0:
            return "huffmanTree({})".format(self.root.value)
        else:
            return "huffmanTree({},{},{})".format(self.totalWeight,self.root.left,self.root
```

This class is an ordered class because it implements the `lt` method and is decorated with the `@totally_ordered` decorator. That method allows the trees to be ordered by their total weight during the merging process. The utility method `print` generates our output recursively, building up a different encoding along every path.

We now consider the construction of the tree:

```
def main():
```

```
s = stdin
freq = LinkedList()

for line in s:
    for c in line:
        if c == '\n':
            continue
        query = leaf(c)
        if not query in freq:
            freq.insert(0,query)
        else:
            item = freq.remove(query)
            item.frequency += 1
            freq.insert(0,item)

trees = SortedList([huffmanTree(l=t) for t in freq])

while len(trees) > 1:
    i = iter(trees)
    smallest = next(i)
    small = next(i)
    trees.remove(smallest)
    trees.remove(small)
    trees.add(huffmanTree(left=smallest,right=small))

if len(trees) > 0:
    ti = iter(trees)
    next(ti).print()
```

There are three phases in this method: the reading of the data, the construction of the character-holding leaves of the tree, and the merging of trees into a single encoding. Several things should be noted:

1. We store characters in a list. Since this list is likely to be small, keeping it ordered requires more code and is not likely to improve performance.
2. The `huffmanTrees` are kept in an `SortedList`. Every time we remove values we must construct a fresh iterator and remove the two smallest trees. When they are merged and reinserted, the wrappers for the two smaller trees can be garbage-collected. (Even better structures for managing these details in Chapter **??**.)
3. The resulting tree is then printed out. In an application, the information in this tree would have to be included with the compressed text to guide the decompression.

When the program is run on the input

```
The mountains, the mountains, we greet them with a song!
```

it generates the output:

```
Encoding of a is 0000 (frequency was 3)
Encoding of i is 0001 (frequency was 3)
Encoding of e is 001 (frequency was 6)
Encoding of m is 0100 (frequency was 3)
Encoding of o is 0101 (frequency was 3)
Encoding of r is 01100 (frequency was 1)
Encoding of ! is 011010 (frequency was 1)
Encoding of T is 011011 (frequency was 1)
Encoding of s is 0111 (frequency was 3)
Encoding of t is 100 (frequency was 6)
Encoding of , is 10100 (frequency was 2)
Encoding of g is 10101 (frequency was 2)
Encoding of h is 1011 (frequency was 4)
Encoding of   is 110 (frequency was 9)
Encoding of u is 11100 (frequency was 2)
Encoding of w is 11101 (frequency was 2)
Encoding of n is 1111 (frequency was 5)
```

Again, the total number of bits that would be used to represent our compressed phrase is only 111, giving us a compression rate of 56 percent. In these days of moving bits about, the construction of efficient compression techniques is an important industry—one industry that depends on the efficient implementation of data structures.

10.10 Example Implementation: Ahnentafel

Having given, in Section 10.2, time to the Republican genealogists, we might now investigate the heritage of a Democrat, William Jefferson Clinton. In Figure 10.8 we see the recent family tree presented as a list. This arrangement is called an *ahnentafel*, or ancestor table. The table is generated by performing a level-order traversal of the pedigree tree, and placing the resulting entries in a table whose index starts at 1.

This layout has some interesting features. First, if we have an individual with index i, the parents of the individual are found in table entries $2i$ and $2i + 1$. Given the index i of a parent, we can find the child (there is only one child for every parent in a pedigree), by dividing the index by 2 and throwing away the remainder.

We can use this as the basis of an implementation of short binary trees. Of course, if the tree becomes tall, there is potentially a great amount of data in the tree. Also, if a tree is not full, there will be empty locations in the table. These must be carefully managed to keep from interpreting these entries as valid data. While the math is fairly simple, our list classes are stored with the first element at location 0. The implementor must either choose to keep location 0 blank or to modify the indexing methods to make use of zero-origin indices.

1	William Jefferson Clinton
2	William Jefferson Blythe III
3	Virginia Dell Cassidy
4	William Jefferson Blythe II
5	Lou Birchie Ayers
6	Eldridge Cassidy
7	Edith Grisham
8	Henry Patton Foote Blythe
9	Frances Ellen Hines
10	Simpson Green Ayers
11	Hattie Hayes
12	James M. Cassidy
13	Sarah Louisa Russell
14	Lemma Newell Grisham
15	Edna Earl Adams

Figure 10.8 The genealogy of President Clinton, presented as a linear table. Each individual is assigned an index i. The parents of the individual can be found at locations $2i$ and $2i+1$. Performing an integer divide by 2 generates the index of a child. Note the table starts at index 1.

Exercise 10.7 *Modify an existing list class so that it keeps track of the index of its first element. This value may be any integer and should default to zero. Make sure, then, that the item-based methods respect this new indexing.*

One possible approach to storing tree information like this is to store entries in key-value pairs in the list structure, with the key being the index. In this way, the tree can be stored compactly and, if the associations are kept in an ordered structure, they can be referenced with only a logarithmic slowdown.

Exercise 10.8 *Describe what would be necessary to allow support for trees with degrees up to eight (called* octtrees*). At what cost do we achieve this increased functionality?*

In Chapter **??** we will make use of an especially interesting binary tree called a *heap*. We will see the ahnentafel approach to storing heaps in a vector shortly.

10.11 Conclusions

The tree is a nonlinear structure. Because of branching in the tree, we will find it is especially useful in situations where decisions can guide the process of adding and removing nodes.

Our approach to implementing the binary tree—a tree with degree 2 or less—is to visualize it as a self-referential structure. This is somewhat at odds with an object-oriented approach. It is, for example, difficult to represent empty self-referential structures in a manner that allows us to invoke methods. To relieve the tension between these two approaches, we represent the empty tree with class instances that represent “empty” trees.

The power of recursion on branching structures is that significant work can be accomplished with very little code. In recursion-limited languages, like Python, it frequently pays to consider efficient implementation that make use of iterative approaches. As we have seen here, traversals often provide a fruitful mechanism for implementing the properties.

Chapter 11

Priority Queues

SOMETIMES A RESTRICTED INTERFACE IS A FEATURE. The *priority queue*, like an ordered structure, appears to keep its data in order. Unlike an ordered structure, however, the priority queue allows the user only to access its smallest element. The priority queue is also similar to the `Linear` structure: values are added to the structure, and they later may be inspected or removed. Unlike their `Linear` counterpart, however, once a value is added to the priority queue it may only be removed if it is the minimum value. It is precisely this restricted interface to the priority queue that allows many of its implementations to run quickly.

Priority queues are used to schedule processes in an operating system, to schedule future events in a simulation, and to generally rank choices that are generated out of order.

Think triage.

11.1 The Interface

Because we will see many contrasting implementations of the priority queue structure, we describe it as abstractly as possible in Python—with an interface:

```
@checkdoc
class PriorityQueue(Sized):
    """
    An abstract base class for Heap and SkewHeap.

    For more information on priority queues, see priority_queue.py.
    """

    @abc.abstractmethod
    def add(self):
        """Insert value into data structure."""
        ...

    @abc.abstractmethod
    def first(self):
        """The next value that will be removed."""
        ...

    @abc.abstractmethod
    def remove(self):
        """Remove next value from data structure."""
        ...

    def values(self):
        """View of all items contained in data structure."""
        return View(self)

    def __eq__(self, other):
        """Data structure has equivalent contents to other."""
        if len(self) != len(other):
            return False
        for x,y in zip(self.values(), other.values()):
            if x != y:
                return False
        return True

    def __hash__(self):
        """Hash value for data structure."""
        # total starting constant from builtin tuplehash function
        total = 1000003
        for v in self.values():
            try:
                x = hash(v)
            except TypeError as y:
                raise y
            total = total + x
        return total

    def __repr__(self):
        """Parsable string representation of data structure."""
        return self.__class__.__name__ + "(" + repr(list(self.values())) + ")"
```

Because they must be kept in order, the elements of a `PriorityQueue` are totally ordered. In this interface the smallest values are found near the front of the queue and will be removed soonest.[1] The `add` operation is used to insert a new value into the queue. At any time a reference to the minimum value can be obtained with the `first` method and is removed with `remove`. A `view` method exposes the elements through a traversal. The remaining methods are similar to those we have seen before.

Notice that the `PriorityQueue` only extends the `Sized` interface. First, as a matter of convenience, `PriorityQueue` methods consume and return values that must be orderable. Most structures we have encountered manipulate unconstrained generic `objects`. Though similar, the `PriorityQueue` is not a

[1] If explicit priorities are to be associated with values, the user may insert a tuple whose first value is an ordered value such as an `int`. In this case, the associated value—the data element—need not be ordered.

Queue. There is, for example, no dequeue method. Though this might be remedied, it is clear that the PriorityQueue need not act like a first-in, first-out structure. At any time, the value about to be removed is the current minimum value. This value might have been the first value inserted, or it might have just recently "cut in line" before larger values. Still, the priority queue is just as general as the stack and queue since, with a little work, one can associate with inserted values a priority that forces any Linear behavior in a PriorityQueue.

The simplicity of the abstract priority queue makes its implementation relatively straightforward. In this chapter we will consider three implementations: one based on use of an SortedStructure and two based on a novel structure called a *heap*. First, we consider an example that emphasizes the simplicity of our interface.

11.2 Example: Improving the Huffman Code

In the Huffman example from Section 10.9 we kept track of a pool of trees. At each iteration of the tree-merging phase of the algorithm, the two lightest-weight trees were removed from the pool and merged. There, we used a SortedList to maintain the collection of trees:

Huffman

```
trees = SortedList([huffmanTree(l=t) for t in freq])

while len(trees) > 1:
    i = iter(trees)
    smallest = next(i)
    small = next(i)
    trees.remove(smallest)
    trees.remove(small)
    trees.add(huffmanTree(left=smallest,right=small))
```

To remove the two smallest objects from the list, we must construct an iterator and remove the first two elements we encounter. This code can be greatly simplified by storing the trees in a PriorityQueue. We then remove the two minimum values:

huffman2

```
trees = PriorityList([huffmanTree(l=t) for t in freq])

while len(trees) > 1:
    smallest = trees.remove()
    small = trees.remove()
    trees.add(huffmanTree(left=smallest,right=small))
```

After the merging is complete, access to the final result is also improved.

A number of interesting algorithms must have access to the minimum of a collection of values, and yet do not require the collection to be sorted. The extra energy required by an Orderedlist to keep all the values in order may, in fact, be excessive for some purposes.

11.3 A List-Based Implementation

Perhaps the simplest implementation of a `PriorityQueue` is to keep all the values in a `SortedList`. Of course, the constructor is responsible for initialization:

```
def __init__(self, data=None, key=None, reverse=False, frozen=False):
    """Construct a priority list from an iterable source."""
    self._data = []
    self._frozen = False
    if data is not None:
        for x in data:
            self.add(x)
    self._frozen = frozen
    self._dirty = False
```

Here, notice that we keep the smallest value at the leftmost index.

The priority queue is very similar to the implementation of the `Orderedlist` structure. In fact, the implementations of the `add` method and the "helper" method `__contains__` can be directly handed off to similar methods for the `SortedList`. Still, values of a priority queue are not removed by value. Instead, method `first` and the parameterless `remove` operate on the `list` element that is smallest (leftmost). The implementation of these routines is straightforward:

```
@property
def first(self):
    """The smallest value in the priority queue."""
    return self._data[-1]

@mutatormethod
def remove(self):
    """Remove the smallest value from the priority queue."""
    self._dirty = True
    return self._data.pop()

@mutatormethod
def add(self,value):
    """Add value to priority queue."""
    self._data.append(None)
    j = len(self._data)-1
    while (j > 0) and not (value < self._data[j-1]):
        self._data[j] = self._data[j-1]
        j -=1
    self._data[j] = value
    self._dirty = True
```

The `first` operation takes constant time. The `remove` operation caches and removes the first value of the `SortedList` with a linear-time complexity. This can be easily avoided by reversing the way that the values are stored in the

`SortedList` (see Problem **??**).

It is interesting to see the evolution of the various types encountered in the discussion of the `PriorityList`. Although the `List` took an entire chapter to investigate, the abstract notion of a list seems to be a relatively natural structure here. Abstraction has allowed us to avoid considering the minute details of the implementation. For example, we assume that `Listss` automatically extend themselves. The abstract notion of an `SortedList`, on the other hand, appears to be insufficient to directly support the specification of the `Prioritylist`. The reason is that the `SortedList` does not support `list` operations like the index-based `get(i)` and `remove(i)`. These could, of course, be added to the `SortedList` interface, but an appeal for symmetry might then suggest implementation of the method `add(i)`. This would be a poor decision since it would then allow the user to insert elements out of order.

Principle 8 *Avoid unnaturally extending a natural interface.*

Designers of data structures spend considerable time weighing these design trade-offs. While it is tempting to make the most versatile structures support a wide variety of extensions, it surrenders the interface distinctions between structures that often allow for novel, efficient, and safe implementations.

Exercise 11.1 *Although the* `SortedList` *class does not directly support the* `PriorityQueue` *interface, it nonetheless can be used, protected inside another class. What are the advantages and disadvantages?*

In Section 11.4 we discuss a rich class of structures that allow us to maintain a loose ordering among elements. It turns out that even a loose ordering is sufficient to implement priority queues.

11.4 A Heap Implementation

In actuality, it is not necessary to develop a complete ranking of the elements of the priority queue in order to support the necessary operations. It is only necessary to be able to quickly identify the *minimum* value and to maintain a relatively loose ordering of the remaining values. This realization is the motivation for a structure called a *heap*.

Definition 11.1 *A* heap *is a binary tree whose root references the minimum value and whose subtrees are, themselves, heaps.*

An alternate definition is also sometimes useful.

Definition 11.2 *A* heap *is a binary tree whose values are in ascending order on every path from root to leaf.*

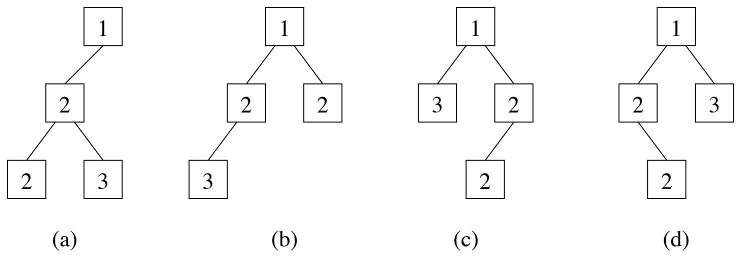


Figure 11.1 Four heaps containing the same values. Note that there is no ordering among siblings. Only heap (b) is complete.

We will draw our heaps in the manner shown in Figure **??**, with the minimum value on the top and the possibly larger values below. Notice that each of the four heaps contains the same values but has a different structure. Clearly, there is a great deal of freedom in the way that the heap can be oriented—for example, exchanging subtrees does not violate the heap property (heaps (c) and (d) are mirror images of each other). While not every tree with these four values is a heap, many are (see Problems **??** and **??**). This flexibility reduces the *friction* associated with constructing and maintaining a valid heap and, therefore, a valid priority queue. When friction is reduced, we have the potential for increasing the speed of some operations.

Principle 9 *Seek structures with reduced friction.*

This is completely obvious.

We will say that a heap is a *complete heap* if the binary tree holding the values of the heap is complete. Any set of n values may be stored in a complete heap. (To see this we need only sort the values into ascending order and place them in level order in a complete binary tree. Since the values were inserted in ascending order, every child is at least as great as its parent.) The abstract notion of a complete heap forms the basis for the first of two heap implementations of a priority queue.

11.4.1 List-Based Heaps

As we saw in Section 10.10 when we considered the implementation of Ahnentafel structures, any complete binary tree (and therefore any complete heap) may be stored compactly in a list. The method involves traversing the tree in level order and mapping the values to successive slots of the list. When we are finished with this construction, we observe the following (see Figure **??**):

1. The root of the tree is stored in location 0. If non-`null`, this location references the minimum value of the heap.

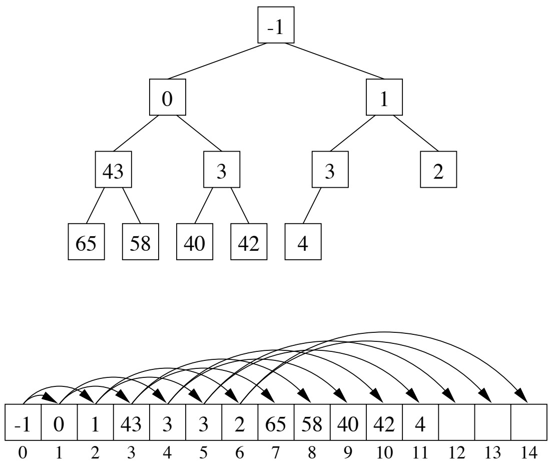


Figure 11.2 An abstract heap (top) and its list representation. Arrows from parent to child are not physically part of the list, but are indices computed by the heap's `left` and `right` methods.

2. The left child of a value stored in location i is found at location $2i + 1$.

3. The right child of a value stored in location i may be found at the location following the left child, location $2(i + 1) = (2i + 1) + 1$.

4. The parent of a value found in location i can be found at location $\lfloor\frac{i-1}{2}\rfloor$. Since division of integers in Java-like languages throws away the remainder for positive numbers, this expression is written `(i-1)/2`.

These relations may, of course, be encoded as functions. In Figure **??** we see the mapping of a heap to a list, with tree relations indicated by arrows in the list. Here are the private `Heap` methods that maintain these relations:

```
def _parent(i):
    """The index of the parent of i."""
    return (i-1)//2

def _left(i):
    """The index of the left child of i."""
    return 2*i+1

def _right(i):
    """The index of the right child of i."""
    return 2*(i+1)
```

The functions `_parent`, `_left`, and `_right` are standalone methods (that is, they're not declared as part of the `Heap` class, to indicate that they do not actually have to be called on any instance of a heap. Instead, their values are functions of their parameters only. Their names contain a leading underscore to emphasize that their use is limited to the `Heap` module.

Principle 10 *Declare standalone object-independent functions when they may be useful to more than one class.*

Notice, in this mapping, that while the list is not maintained in ascending order, any path from the root to a leaf *does* encounter values in ascending order. Initially, the `Heap` is represented by an empty list that we expect to grow. If the list is ever larger than necessary, slots not associated with tree nodes are set to `None`. The `Heap` initialization method is fairly straightforward:

```
@checkdoc
class Heap(PriorityQueue,Freezable):

    __slots__ = ["_data", "_dirty", "_frozen", "_hash"]

    def __init__(self,data=None,frozen=False,key=None,reverse=False):
        """Construct a Heap from an iterable source."""
        self._data = []
```

Now consider the addition of a value to a complete heap. We know that the heap is currently complete. Ideally, after the addition of the value the heap will remain complete but will contain one extra value. This realization forces us to commit to inserting a value in a way that ultimately produces a heap with the desired structure. Since the first free element of the `list` will hold a value, we optimistically insert the new value in that location (see Figure **??**). If, considering the path from the leaf to the root, the value is in the wrong location, then it must be "percolated upward" to the correct entry. We begin by comparing and, if necessary, exchanging the new value and its parent. If the values along the path are still incorrectly ordered, it must be because of the new value, and we continue to percolate the value upward until either the new value is the root or it is greater than or equal to its current parent. The only values possibly exchanged in this operation are those appearing along the unique path from the insertion point. Since locations that change only become smaller, the integrity of other paths in the tree is maintained.

The code associated with percolating a value upward is contained in the private `Heap` method, `_percolateUp`. This function takes an index of an value that is possibly out of place and pushes the value upward toward the root until it reaches the correct location. While the routine takes an index as a parameter, the parameter passed is usually the index of the rightmost leaf of the bottom level.

```
def _percolateUp(self,leaf):
    """Move a (possibly small) leaf value up in a (possibly) nonheap
    to form a min-heap in O(log n) time."""
```

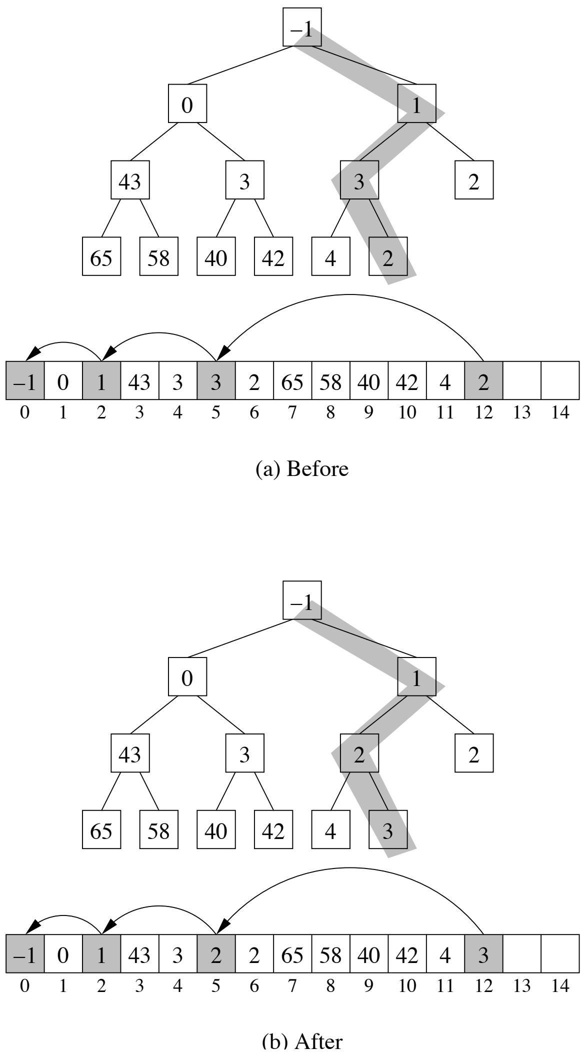


Figure 11.3 The addition of a value (2) to a list-based heap. (a) The value is inserted into a free location known to be part of the result structure. (b) The value is percolated up to the correct location on the unique path to the root.

```
        parent = _parent(leaf)
        value = self._data[leaf]
        while (leaf > 0) and (value < self._data[parent]):
            self._data[leaf] = self._data[parent]
            leaf = parent
            parent = _parent(leaf)
        self._data[leaf] = value
```

Adding a value to the priority queue is then only a matter of appending it to the end of the list (the location of the newly added leaf) and percolating the value upward until it finds the correct location.

```
    @mutatormethod
    def add(self,value):
        """Add a comparable value to the Heap."""
        self._dirty = True
        self._data.append(value)
        self._percolateUp(len(self) - 1)
```

Let us consider how long it takes to accomplish the addition of a value to the heap. Remember that the tree that we are working with is an n-node complete binary tree, so its height is $\lfloor \log_2 n \rfloor$. Each step of the percolateUp routine takes constant time and pushes the new value up one level. Of course, it may be positioned correctly the first time, but the worst-case behavior of inserting the new value into the tree consumes $O(\log n)$ time. This performance is considerably better than the linear behavior of the PriorityList implementation described in Section 11.3. What is the best time? It is constant when the value added is large compared to the values found on the path from the new leaf to the root.

Next, we consider the removal of the minimum value (see Figures **??** and **??**). It is located at the root of the heap, in the first slot of the list. The removal of this value leaves an empty location at the top of the heap. Ultimately, when the operation is complete, the freed location will be the rightmost leaf of the bottom level, the last element of the underlying list. Again, our approach is first to construct a tree that is the correct shape, but potentially not a heap, and then perform transformations on the tree that both maintain its shape and bring the structure closer to being a heap. Thus, when the minimum value is removed, the rightmost leaf on the bottom level is removed and re-placed at the root of the tree (Figure **??**a and b). At this point, the tree is the correct shape, but it may not be a heap because the root of the tree is potentially too large. Since the subtrees remain heaps, we need to ensure the root of the tree is the minimum value contained in the tree. We first find the minimum child and compare this value with the root (Figure **??**a). If the root value is no greater, the minimum value is at the root and the entire structure is a heap. If the root is larger, then it is exchanged with the true minimum—the smallest child—pushing the large value downward. At this point, the root of the tree has the correct value. All but one of the subtrees are unchanged, and the shape of the tree remains correct. All that has happened is that a large value has been pushed down to where it may violate the heap property in a subtree. We then perform any further exchanges

We're "heaping in shape."

recursively, with the value sinking into smaller subtrees (Figure **??**b), possibly becoming a leaf. Since any single value is a heap, the recursion must stop by the time the newly inserted value becomes a leaf.

Here is the private method associated with the pushing down of the root:

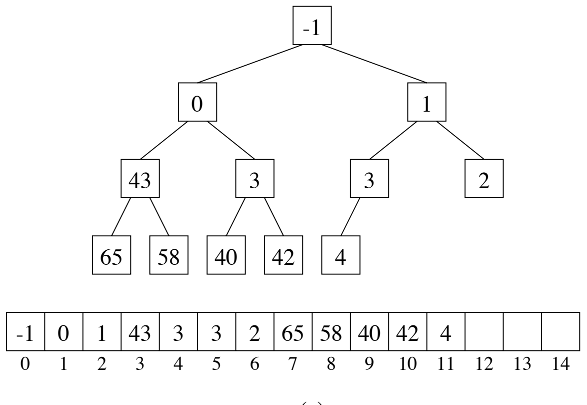
```
def _pushDownRoot(self,root):
    """Take a (possibly large) value at the root and two subtrees that are
    heaps, and push the root down to a point where it forms a single heap in O(log n) time."""
    heapSize = len(self)
    value = self._data[root]
    while root < heapSize:
        childpos = _left(root)
        if childpos < heapSize:
            if (_right(root) < heapSize) and \
               self._data[childpos+1] < self._data[childpos]:
               childpos += 1
            # Assert: childpos indexes smaller of two children
            if self._data[childpos] < value:
                self._data[root] = self._data[childpos]
                root = childpos
            else:
                self._data[root] = value
                return
        else:
            self._data[root] = value
            return
```

The `remove` method simply involves returning the smallest value of the heap, but only after the rightmost element of the list has been pushed downward.

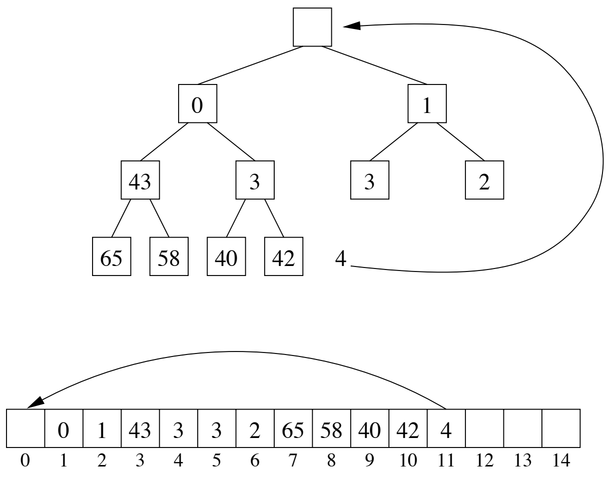
```
@mutatormethod
def remove(self):
    """Remove the minimum element from the Heap."""
    assert len(self) > 0
    self._dirty = True
    minVal = self.first
    self._data[0] = self._data[len(self)-1];
    self._data.pop()
    if len(self) > 1 :
        self._pushDownRoot(0);
    return minVal
```

Each iteration in `pushDownRoot` pushes a large value down into a smaller heap on a path from the root to a leaf. Therefore, the performance of `remove` is $O(\log n)$, an improvement over the behavior of the `Prioritylist` implementation.

Since we have implemented all the required methods of the `PriorityQueue`, the `Heap` implements the `PriorityQueue` and may be used wherever a priority queue is required.



(a)



(b)

Figure 11.4 Removing a value from the heap shown in (a) involves moving the right-most value of the list to the top of the heap as in (b). Note that this value is likely to violate the heap property but that the subtrees will remain heaps.

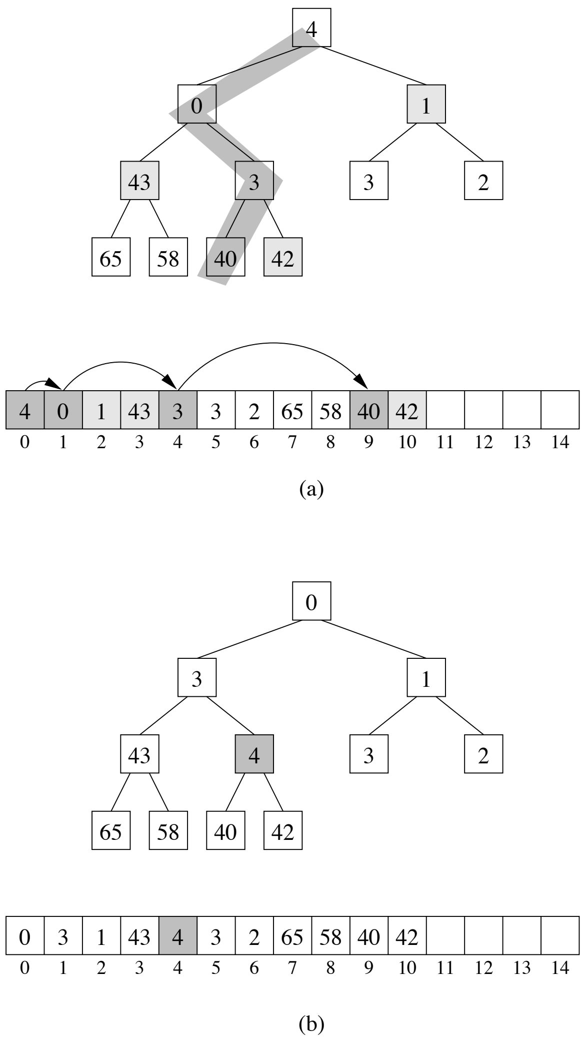


Figure 11.5 Removing a value (continued). In (a) the newly inserted value at the root is pushed down along a shaded path following the smallest children (lightly shaded nodes are also considered in determining the path). In (b) the root value finds, over several iterations, the correct location along the path. Smaller values shift upward to make room for the new value.

The advantages of the Heap mechanism are that, because of the unique mapping of complete trees to the list, it is unnecessary to explicitly store the connections between elements. Even though we are able to get improved performance over the Prioritylist, we do not have to pay a space penalty. The complexity arises, instead, in the code necessary to support the insertion and removal of values.

11.4.2 Example: Heapsort

Any priority queue, of course, can be used as the underlying data structure for a sorting mechanism. When the values of a heap are stored in a list, an empty location is potentially made available when they are removed. This location could be used to store a removed value. As the heap shrinks, the values are stored in the newly vacated elements of the list. As the heap becomes empty, the list is completely filled with values in descending order.

Unfortunately, we cannot make assumptions about the structure of the values initially found in the list; we are, after all, sorting them. Since this approach depends on the values being placed in a heap, we must consider one more operation: a constructor that “heapifies” the data found in an arbitrarily ordered list passed to it:

```
def _heapify(self):
    """Rearrange randomly ordered data into a heap in O(n) time."""
    # Work from lowest parents rootward, making heaps.
    n = len(self._data)
    for i in reversed(range(1+_parent(n-1))):
        self._pushDownRoot(i)
```

The process of constructing a heap from an unordered list obviously takes the time of n add operations, each of which is $O(\log n)$. The worst-case cost of “heapifying” is, therefore, $O(n \log n)$. (This can be improved—see Problem **??**.)

To demonstrate the utility of this standalone method, we can use it in our Heap constructor to quickly add the initial data elements, which are likely to be disordered.

```
def __init__(self,data=None,frozen=False,key=None,reverse=False):
    """Construct a Heap from an iterable source."""
    self._data = []
    # bulk addition is most efficient using heapify
    if data:
        self._data.extend(data)
        self._heapify()
```

Now, returning to heapsort, we observe gathering the values from smallest to largest requires n remove operations. This phase also has worst-case complexity $O(n \log n)$. We have, therefore, another sorting algorithm with $O(n \log n)$ behavior.

```
@sortoptions
def heapsort(l):
    """Perform heapsort of values."""
    h = Heap(l)
    for i in range(len(l)):
        l[i] = h.remove()
```

At any time, of course, the amount of space used is exactly n list entries.

The feature of a heap that makes the sort so efficient is the heap's short height. The values are always stored in as full a tree as possible and, as a result, we may place a logarithmic upper bound on the time it takes to insert and remove values. In Section 11.4.3 we investigate the use of unrestricted heaps to implement priority queues. These structures have *amortized* cost that is equivalent to heaps built atop lists.

11.4.3 Skew Heaps

The performance of `list`-based heaps is directly dependent on the fact that these heaps are complete. Since complete heaps are a minority of all heaps, it is reasonable to ask if efficient priority queues might be constructed from unrestricted heaps. The answer is yes, if we relax the way we measure performance.

We consider, here, the implementation of heaps using dynamically structured binary trees. A direct cost of this decision is the increase in space. Whereas a `list` stores a single reference, the binary tree node keeps an additional three references. These three references allow us to implement noncomplete heaps, called *skew heaps*, in a space-efficient manner (see Problem **??**). Here are the protected data and the initializer for this structure:

SkewHeap

```
@checkdoc
class SkewHeap(PriorityQueue):

    __slots__ = ["_root", "_count", "_dirty", "_frozen", "_hash",
                 "_keyfun", "_reverse", "_cmp"]

    def __init__(self,data=None,frozen=False, key=None, reverse=False):
        """Construct a SkewHeap from an iterable source."""
        self._root = BinTree.Empty
        self._count = 0
        self._frozen = False
        self._cmp = Comparator(key=key,reverse=reverse)
        if data:
            for x in data:
                self.add(x)
        self._frozen = frozen
        self._hash = None
```

Notice that we keep track of the size of the heap locally, rather than asking the `BinaryTree` for its size. This is simply a matter of efficiency, but it requires us to maintain the value within the `add` and `remove` procedures. Once we commit to implementing heaps in this manner, we need to consider the implementation of each of the major operators.

The implementation of `first` simply references the value stored at the root. Its implementation is relatively straightforward:

```
@property
def first(self):
    """The first value in the priority queue."""
    return self._root.value
```

As with all good things, this will eventually seem necessary.

Before we consider the implementation of the `add` and `remove` methods, we consider a (seemingly unnecessary) operation, `_merge`. This method takes two heaps and merges them together. This is a *destructive* operation: the elements of the participating heaps are consumed in the construction of the result. Our approach will be to make `_merge` a recursive method that considers several cases. First, if either of the two heaps participating in the merge is empty, then the result of the merge is the other heap. Otherwise, both heaps contain at least a value—assume that the minimum root is found in the left heap (if not, we can swap them). We know, then, that the result of the merge will be a reference to the root node of the left heap. To see how the right heap is merged into the left we consider two cases:

1. If the left heap has no left child, make the right heap the left child of the left heap (see Figure 11.6b).

2. Otherwise, exchange the left and right children of the left heap. Then merge (the newly made) left subheap of the left heap with the right heap (see Figure 11.6d).

Notice that if the left heap has one subheap, the right heap becomes the left subheap and the merging is finished. Here is the code for the `merge` method:

```
def _merge(self, left, right):
    """Merge two BinTree heaps into one."""
    if left.empty: return right
    if right.empty: return left
    leftVal = left.value
    rightVal = right.value
    if self._cmp(rightVal, leftVal) < 0:
        result = self._merge(right,left)
    else:
        result = left
        if result.left.empty:
            result.left = right
        else:
            temp = result.right
            result.right = result.left
            result.left = self._merge(temp,right)
```

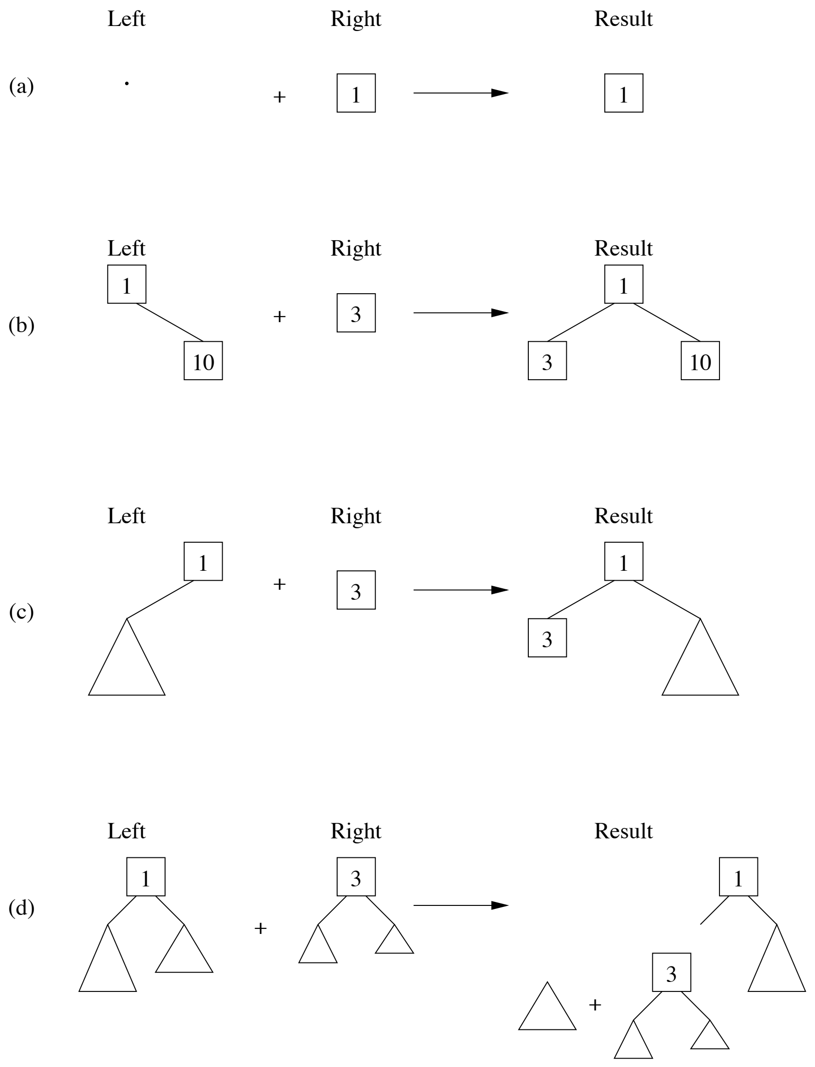


Figure 11.6 Different cases of the `merge` method for `SkewHeaps`. In (a) one of the heaps is empty. In (b) and (c) the right heap becomes the left child of the left heap. In (d) the right heap is merged into what was the right subheap.

```
        return result
```

Exercise 11.2 *Rewrite the merge method to be iterative.*

Once the `merge` method has been defined, we find that the process of adding a value or removing the minimum is relatively straightforward. To add a value, we construct a new `BinaryTree` containing the single value that is to be added. This is, in essence, a one-element heap. We then merge this heap with the existing heap, and the result is a new heap with the value added:

```
    @mutatormethod
    def add(self, value):
        """Add a comparable value to the SkewHeap."""
        self._dirty = True
        smallTree = BinTree(value)
        self._root = self._merge(smallTree,self._root)
        self._count += 1
```

To remove the minimum value from the heap we must extract and return the value at the root. To construct the smaller resulting heap we detach both subtrees from the root and merge them together. The result is a heap with all the values of the left and right subtrees, but not the root. This is precisely the result we require. Here is the code:

```
    @mutatormethod
    def remove(self):
        """Remove the minimum element from the SkewHeap."""
        self._dirty = True
        result = self._root.value
        self._root = self._merge(self._root.left,self._root.right)
        self._count -= 1
        return result
```

The remaining priority queue methods for skew heaps are implemented in a relatively straightforward manner.

Because a skew heap has unconstrained topology (see Problem **??**), it is possible to construct examples of skew heaps with degenerate behavior. For example, adding a new maximum value can take $O(n)$ time. For this reason we cannot put very impressive bounds on the performance of any individual operation. The skew heap, however, is an example of a self-organizing structure: inefficient operations spend some of their excess time making later operations run more quickly. If we are careful, time "charged against" early operations can be *amortized* or redistributed to later operations, which we hope will run very efficiently. This type of analysis can be used, for example, to demonstrate that $m > n$ skew heap operations applied to a heap of size n take no more than $O(m \log n)$ time. On average, then, each operation takes $O(\log n)$ time. For applications where it is expected that a significant number of requests of a heap will be made, this performance is appealing.

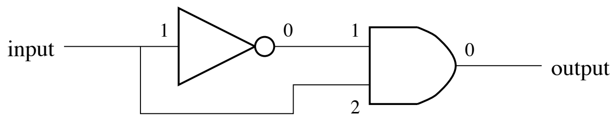


Figure 11.7 A circuit for detecting a rising logic level.

11.5 Example: Circuit Simulation

Consider the electronic digital circuit depicted in Figure 11.7. The two devices shown are *logic gates*. The wires between the gates propagate electrical signals. Typically a zero voltage is called *false* or *low*, while a potential of 3 volts or more is *true* or *high*.

The triangular gate, on the left, is an *inverter*. On its output (pin 0) it "inverts" the logic level found on the input (pin 1): false becomes true and true becomes false. The gate on the right is an *and-gate*. It generates a true on pin 0 exactly when both of its inputs (pins 1 and 2) are true.

The action of these gates is the result of a physical process, so the effect of the inputs on the output is delayed by a period of time called a *gate delay*. Gate delays depend on the complexity of the gate, the manufacturing process, and environmental factors. For our purposes, we'll assume the gate delay of the inverter is 0.2 nanosecond (ns) and the delay of the and-gate is 0.8 ns.

The question is, what output is generated when we toggle the input from low to high and back to low several times? To determine the answer we can build the circuit, or simulate it in software. For reasons that will become clear in a moment, simulation will be more useful.

The setup for our simulation will consist of a number of small classes. First, there are a number of components, including an `Inverter`; an `And`; an input, or `Source`; and a voltage sensor, or `Probe`. When constructed, gates are provided gate delay values, and `Sources` and `Probes` are given names. Each of these components has one or more pins to which wires can be connected. (As with real circuits, the outputs should connect only to inputs of other components!) Finally, the voltage level of a particular pin can be set to a particular level. As an example of the interface, we list the `public` methods for the `And` gate:

```
class And(Component):
    def __init__(self,delay):
        super().__init__(delay,3)

    def set(self,t,pinNum,level):
```

`Circuit`

Notice that there is a time associated with the `set` method. This helps us document when different events happen in the component. These events are simulated by a comparable Event class. This class describes a change in logic level on an input pin for some component. As the simulation progresses, Events are created and scheduled for simulation in the future. The ordering of Events is based on an event time. Here are the details:

```
@total_ordering
class Event(object):
    __slots__ = ["_time", "_level", "_c" ]

    def __init__(self, conn, t, l):
        self._c = conn
        self._time = t
        self._level = l

    def go(self):
        self._c.component.set(self._time,self._c.pin,self._level)

    def __lt__(self,other):
        return self._time < other._time
```

The Connection mentioned here is simply a component's input pin.

Finally, to orchestrate the simulation, we use a priority queue to correctly simulate the order of events. The following method simulates a circuit by removing events from the priority queue and setting the logic level on the appropriate pins of the components. The method returns the time of the last event to help monitor the progress of the simulation.

```
EventQueue = None
time = 0

def simulate():
    global EvenQueue,time
    # post: run simulation until event queue is empty
    #       returns final clock time
    low = 0.0   # voltage of low logic
    high = 3.0  # voltage of high logic
    clock = 0.0
    while not EventQueue.empty:
        e = EventQueue.remove()
        # keep track of time
        clock = e._time
        # simulate the event
        e.go()
    print("-- circuit stable after {} ns --".format(clock))
    return clock
```

As events are processed, the logic level on a component's pins are updated. If the inputs to a component change, new Events are scheduled one gate delay later for each component connected to the output pin. For Sources and Probes, we write a message to the output indicating the change in logic level. Clearly, when there are no more events in the priority queue, the simulated circuit is stable. If the user is interested, he or she can change the logic level of a Source and resume the simulation by running the simulate method again.

We are now equipped to simulate the circuit of Figure 11.7. The first portion of the following code sets up the circuit, while the second half simulates the effect of toggling the input several times:

```
global time, EventQueue
low = 0     # voltage of low logic
high = 3    # voltage of high logic
EventQueue = Heap()

# set up circuit
inv = Inverter(0.2)
and_gate = And(0.8)
output = Probe("output")
input = Source("input",inv.pin(1))

input.connectTo(and_gate.pin(2))
inv.connectTo(and_gate.pin(1))
and_gate.connectTo(output.pin(1))

# simulate circuit
time = 0
time = simulate()
input.set(time+1.0,0,high) # first: set input high
time = simulate()
input.set(time+1.0,0,low)  # later: set input low
time = simulate()
input.set(time+1.0,0,high) # later: set input high
time = simulate()
input.set(time+1.0,0,low)  # later: set input low
simulate()
```

When run, the following output is generated:

```
1.0 ns: output now 0 volts
-- circuit stable after 1.0 ns --
2.0 ns: input set to 3 volts
2.8 ns: output now 3 volts
3.0 ns: output now 0 volts
-- circuit stable after 3.0 ns --
4.0 ns: input set to 0 volts
-- circuit stable after 5.0 ns --
6.0 ns: input set to 3 volts
6.8 ns: output now 3 volts
7.0 ns: output now 0 volts
```

```
-- circuit stable after 7.0 ns --
8.0 ns: input set to 0 volts
-- circuit stable after 9.0 ns --
```

When the input is moved from low to high, a short spike is generated on the output. Moving the input to low again has no impact. The spike is generated by the *rising edge* of a signal, and its width is determined by the gate delay of the inverter. Because the spike is so short, it would have been difficult to detect it using real hardware.[2] Devices similar to this edge detector are important tools for detecting changing states in the circuits they monitor.

11.6 Conclusions

We have seen three implementations of priority queues: one based on a `list` that keeps its entries in order and two others based on heap implementations. The `list` implementation demonstrates how any ordered structure may be adapted to support the operations of a priority queue.

Heaps form successful implementations of priority queues because they relax the conditions on "keeping data in priority order." Instead of maintaining data in sorted order, heaps maintain a competition between values that becomes progressively more intense as the values reach the front of the queue. The cost of inserting and removing values from a heap can be made to be as low as $O(\log n)$.

If the constraint of keeping values in a `list` is too much (it may be impossible, for example, to allocate a single large chunk of memory), or if one wants to avoid the uneven cost of extending a `list`, a dynamic mechanism is useful. The `SkewHeap` is such a mechanism, keeping data in general heap form. Over a number of operations the skew heap performs as well as the traditional `list`-based implementation.

[2] This is a very short period of time. During the time the output is high, light travels just over 2 inches!

Chapter 12

Search Trees

STRUCTURES ARE OFTEN THE SUBJECT OF A SEARCH. We have seen, for example, that binary search is a natural and efficient algorithm for finding values within ordered, randomly accessible structures. Recall that at each point the algorithm compares the value sought with the value in the middle of the structure. If they are not equal, the algorithm performs a similar, possibly recursive search on one side or the other. The pivotal feature of the algorithm, of course, was that the underlying structure was in order. The result was that a value could be efficiently *found* in approximately logarithmic time. Unfortunately, the modifying operations—`add` and `remove`—had complexities that were determined by the linear nature of the vector.

Heaps have shown us that by relaxing our notions of order we can improve on the linear complexities of adding and removing values. These logarithmic operations, however, do not preserve the order of elements in any obviously useful manner. Still, if we were somehow able to totally order the elements of a binary tree, then an algorithm like binary search might naturally be imposed on this branching structure.

12.1 Binary Search Trees

The *binary search tree* is a binary tree whose elements are kept in order. This is easily stated as a recursive definition.

Definition 12.1 *A binary tree is a* binary search tree *if it is trivial, or if every node is simultaneously greater than or equal to each value in its left subtree, and less than or equal to each value in its right subtree.*

To see that this is a significant restriction on the structure of a binary tree, one need only note that if a maximum of n distinct values is found at the root, all other values must be found in the left subtree. Figure 12.1 demonstrates the many trees that can contain even three distinct values. Thus, if one is not too picky about the value one wants to have at the root of the tree, there are still a significant number of trees from which to choose. This is important if we want to have our modifying operations run quickly: ideally we should be as nonrestrictive about the outcome as possible, in order to reduce the friction of the operation.

One important thing to watch for is that even though our definition allows duplicate values of the root node to fall on either side, our code will prefer to

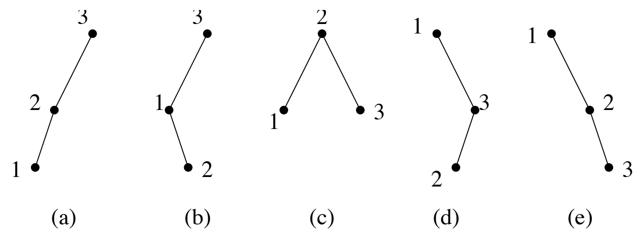


Figure 12.1 Binary search trees with three nodes.

have them on the left. This preference is arbitrary. If we assume that values equal to the root will be found in the left subtree, but in actuality some are located in the right, then we might expect inconsistent behavior from methods that search for these values. In fact, this is not the case.

To guide access to the elements of a binary search tree, we will consider it an implementation of an `Sorted`, supporting the following methods:

`SearchTree`

```
@checkdoc
class SearchTree(Sorted,Freezable):
    Tree = BinTree
    __slots__ = [ "_root", "_count", "_dirty", "_frozen", "_hash",
                  "_cmp" ]

    def __init__(self,data=None,frozen=False,key=None,reverse=False):
        """Construct a SearchTree from an iterable source."""
    @property
    def empty(self):
        """SearchTree contains no items."""
    @mutatormethod
    def add(self,value):
        """Add value to SearchTree."""
    def _insert(self,value):
        """Insert value into appropriate location in SearchTree in O(log n) time."""
    def __contains__(self,value):
        """value is in SearchTree."""
    @mutatormethod
    def remove(self,value):
        """Remove value from SearchTree."""
```

Unlike the `BinTree`, the `SearchTree` provides only one `iterator` method. This method provides for an in-order traversal of the tree, which, with some thought, allows access to each of the elements in order.

Maybe even with no thought!

12.2 Example: Tree Sort

Because the SearchTree is an Sorted it provides the natural basis for sorting. The algorithm of Section **??** will work equally well here, provided the allocation of the Sorted is modified to construct a SearchTree. The binary search structure, however, potentially provides significant improvements in performance. If the tree can be kept reasonably short, the cost of inserting each element is $O(\log n)$. Since n elements are ultimately added to the structure, the total cost is $O(n \log n)$.[1] As we have seen in Chapter **??**, all the elements of the underlying binary tree can be visited in linear time. The resulting algorithm has a potential for $O(n \log n)$ time complexity, which rivals the performance of sorting techniques using heaps. The advantage of binary search trees is that the elements need not be removed to determine their order. To attain this performance, though, we must keep the tree as short as possible. This will require considerable attention.

12.3 Example: Associative Structures

Associative structures play an important role in making algorithms efficient. In these data structures, *values* are associated with *keys*. Typically (though not necessarily), the keys are unique and aid in the retrieval of more complete information—the value. In a list, for example, we use integers as indices to find values. The *SymbolTable* associated with the PostScript lab (Section **??**) is, essentially, an associative structure.

Associative structures are an important feature of many symbol-based systems. Here, for example, is a first approach to the construction of a general-purpose symbol table, with potentially logarithmic performance:

```
class SymTab(object):
    __slots__ = ["_table"]

    def __init__(self):
        self._table = SearchTree()

    def __contains__(self,symbol):
        return KV(symbol) in self._table

    def __setitem__(self,symbol,value):
        pair = KV(symbol,value)
        if pair in self._table:
            self._table.remove(pair)
        self._table.add(pair)

    def __getitem__(self,symbol):
```

[1] This needs to be proved! See Problem **??**.

```
        a = KV(symbol, None)
        if a in self._table:
            a = self._table.get(a)
            return a.value
        else:
            return None

    def __delitem__(self,symbol):
        a = KV(symbol,None)
        if a in self._table:
            self._table.remove(a)

    def remove(self,symbol):
        a = KV(symbol,None)
        if a in self._table:
            a = self._table.remove(a)
            return a.value
        else:
            return None
```

Based on such a table, we might have a program that reads in a number of alias-name pairs terminated by the word END. After that point, the program prints out the fully translated aliases:

```
table = SymTab()
reading = True
for line in stdin:
    words = line.split()
    if len(words) == 0:
        continue
    if reading:
        if words[0] == 'END':
            reading = False
        else:
            table[words[0]] = words[1]
    else:
        name = words[0]
        while name in table:
            name = table[name]
        print(name)
```

Given the input:

```
three 3
one unity
unity 1
pi three
END
one
two
```

```
three
pi
```

the program generates the following output:

```
1
two
3
3
```

We will consider general associative structures in Chapter **??**, when we discuss dictionaries. We now consider the details of actually supporting the `SearchTree` structure.

12.4 Implementation

In considering the implementation of a `SearchTree`, it is important to remember that we are implementing an `Sorted`. The methods of the `Sorted` accept and return values that are to be compared with one another. By default, we assume that the data are ordered and that the natural order is sufficient. If alternative orders are necessary, or an ordering is to be enforced on elements that do not directly implement a `__lt__` method, alternative key functions may be used. Essentially the only methods that we depend upon are the compatibility of the key function and the elements of the tree.

We begin by noticing that a `SearchTree` is little more than a binary tree with an imposed order. We maintain a reference to a `BinTree` and explicitly keep track of its size. The constructor need only initialize these two fields and suggest an ordering of the elements to implement a state consistent with an empty binary search tree:

```
@checkdoc
class SearchTree(Sorted,Freezable):
    Tree = BinTree
    __slots__ = [ "_root", "_count", "_dirty", "_frozen", "_hash",
                  "_cmp" ]

    def __init__(self,data=None,frozen=False,key=None,reverse=False):
        """Construct a SearchTree from an iterable source."""
        self._root = self.Tree.Empty
        self._count = 0
        self._frozen = False
        self._cmp = Comparator(key=key,reverse=reverse)
        if data:
            for x in data:
                self.add(x)
        self._frozen = frozen
        self._hash = None
```

As with most implementations of `Sorted`s, we develop a method to find the correct location to insert the value and then use that method as the basis for implementing the public methods—`add`, `contains`, and `remove`. Our approach to the method `_locate` is to have it return a reference to the location that identifies the correct point of insertion for the new value. This method, of course, makes heavy use of the ordering. Here is the Python code for the method:

```
def _locate(self,value):
    """A private method to return either
    (1) a node that contains the value sought, or
    (2) a node that would be the parent of a node created to hold value, or
    (3) an empty tree if the tree is empty."""
    result = node = self._root
    while not node.empty:
        nodeValue = node.value
        result = node
        relation = self._cmp(value,nodeValue)
        if relation == 0: return result
        node = node.left if relation < 0 else node.right
    return result
```

The approach of the `_locate` method parallels binary search. Comparisons are made with the `_root`, which serves as a median value. If the value does not match, then the search is refocused on either the left side of the tree (among smaller values) or the right side of the tree (among larger values). In either case, if the search is about to step off the tree, the current node is returned: if the value were added, it would be a child of the current node.

Once the `_locate` method is written, the `__contains__` method must check to see if the node returned by `_locate` actually equals the desired value:[2]

```
def __contains__(self,value):
    """value is in SearchTree."""
    if self._root.empty: return False
    possibleLocation = self._locate(value)
    return 0 == self._cmp(value,possibleLocation.value)
```

It now becomes a fairly straightforward task to add a value. We simply locate the value in the tree using the `_locate` function. If the value was not found, `_locate` returned a node off of which a leaf with the desired value may be added. If, however, `_locate` has found an equivalent value, we must insert the new value as the right child of the predecessor of the node returned by `_locate`.[3]

```
@mutatormethod
```

[2] We reemphasize at this point the importance of making sure that the `__eq__` method for an object is consistent with the ordering suggested by the `_lt` method of the particular `Comparator`.

[3] With a little thought, it is clear to see that this is a correct location. If there are two copies of a value in a tree, the second value added is a descendant and predecessor (in an in-order traversal) of the located value. It is also easy to see that a predecessor has no right child, and that if one is added, *it* becomes the predecessor.

```
def add(self,value):
    """Add value to SearchTree."""
    self._insert(value)

def _insert(self,value):
    """Insert value into appropriate location in SearchTree in O(log n) time."""
    self._dirty = True
    newNode = self.Tree(value)
    if self._root.empty:
        self._root = newNode
    else:
        insertLocation = self._locate(value)
        nodeValue = insertLocation.value
        if self._cmp(nodeValue,value) < 0:
            insertLocation.right = newNode
        else:
            if not insertLocation.left.empty:
                self._pred(insertLocation).right = newNode
            else:
                insertLocation.left = newNode
    self._count += 1
    return newNode
```

Our add code makes use of the protected "helper" function, `_pred`, which, after calling `_rightmost`, returns a pointer to the node that immediately precedes the indicated root:

```
def _pred(self,root):
    """Node immediately preceding root."""
    return self._rightmost(root.left)

def _rightmost(self,root):
    """Rightmost descendant of root."""
    result = root
    while not result.right.empty:
        result = result.right
    return result
```

A similar routine can be written for successor, and would be used if we preferred to store duplicate values in the right subtree.

We now approach the problem of removing a value from a binary search tree. Observe that if it is found, it might be an internal node. The worst case occurs when the root of a tree is involved, so let us consider that problem.

There are several cases. First (Figure 12.2a), if the root of a tree has no left child, the right subtree can be used as the resulting tree. Likewise (Figure 12.2b), if there is no right child, we simply return the left. A third case (Figure 12.2c) occurs when the left subtree has no right child. Then, the right

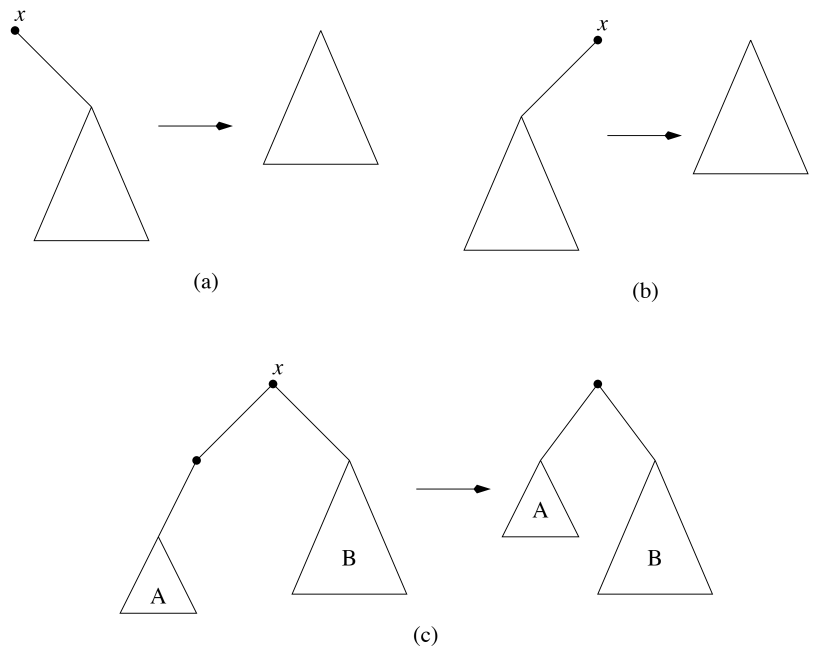


Figure 12.2 The three simple cases of removing a root value from a tree.

subtree—a tree with values no smaller than the left root—is made the right subtree of the left. The left root is returned as the result. The opposite circumstance could also be true.

We are, then, left to consider trees with a left subtree that, in turn, contains a right subtree (Figure 12.3). Our approach to solving this case is to seek out the predecessor of the root and make it the new root. Note that even though the predecessor does not have a right subtree, it may have a left. This subtree can take the place of the predecessor as the right subtree of a nonroot node. (Note that this is the result that we would expect if we had recursively performed our node-removing process on the subtree rooted at the predecessor.)

Finally, here is the Python code that removes the top `BinTree` of a tree and returns the root of the resulting tree:

```
def _remove_node(self,top):
    """Remove the top of the binary tree pointed to by top.
    The result is the root of the tree to be used as replacement."""
    left,right = top.left,top.right
    top.left = top.right = self.Tree.Empty
    # There are three general cases:
    #  - the root is empty (avoid): return same
    #  - the root has fewer than two children: the one (or empty) is new top
```

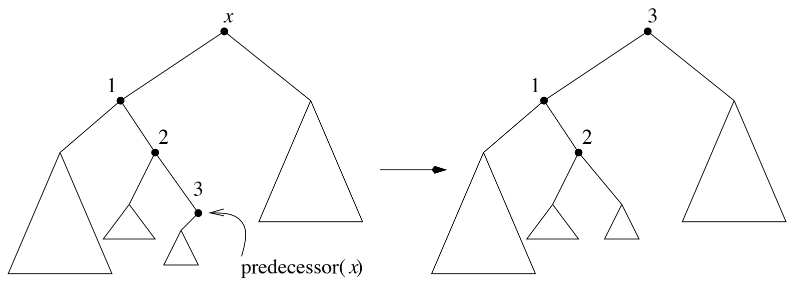


Figure 12.3 Removing the root of a tree with a rightmost left descendant.

```
        #  - the root has two children:
        #    the predecessor of the old root use used as parent of left and right
        if left.empty:
            return right
        elif right.empty:
            return left
        newTop = self._rightmost(left)
        # At this point, newTop will be the new root of this tree.
        # It may be the left node: make right the right child of left
        if newTop is left:
            newTop.right = right
        # It has a parent, but no right child: reparent left child under parent
        # then make left and right children of newtop
        else:
            p = newTop.parent
            newTop.clip()
            p.right = newTop.left
            newTop.left,newTop.right = left,right
        return newTop
```

With the combined efforts of the `_remove_node` and `_locate` methods, we can now simply locate a value in the search tree and, if found, remove it from the tree. We must be careful to update the appropriate references to rehook the modified subtree back into the overall structure.

Notice that inserting and removing elements in this manner ensures that the in-order traversal of the underlying tree delivers the values stored in the nodes in a manner that respects the necessary ordering. We use this, then, as our preferred iteration method.

```
    def __iter__(self):
        """Iterator over the SearchTree."""
        return self._root.valueOrder()
```

The remaining methods (`size`, etc.) are implemented in a now-familiar manner.

Exercise 12.1 *One possible approach to keeping duplicate values in a binary search tree is to keep a list of the values in a single node. In such an implementation, each element of the list must appear externally as a separate node. Modify the* `SearchTree` *implementation to make use of these lists of duplicate values.*

Each of the time-consuming operations of a `SearchTree` has a worst-case time complexity that is proportional to the height of the tree. It is easy to see that checking for or adding a leaf, or removing a root, involves some of the most time-consuming operations. Thus, for logarithmic behavior, we must be sure that the tree remains as short as possible.

Unfortunately, we have no such assurance. In particular, one may observe what happens when values are inserted in descending order: the tree is heavily skewed to the left. If the same values are inserted in ascending order, the tree can be skewed to the right. If these values are distinct, the tree becomes, essentially, a singly linked list. Because of this behavior, we are usually better off if we shuffle the values beforehand. This causes the tree to become, on average, shorter and more balanced, and causes the expected insertion time to become $O(\log n)$.

Considering that the tree is responsible for maintaining an order among data values, it seems unreasonable to spend time shuffling values before ordering them. In Section 12.5 we find out that the process of adding and removing a node can be modified to maintain the tree in a relatively balanced state, with only a little overhead.

12.5 Splay Trees

Because the process of adding a new value to a binary search tree is *deterministic*—it produces the same result tree each time—and because inspection of the tree does not modify its structure, one is stuck with the performance of any degenerate tree constructed. What might work better would be to allow the tree to reconfigure itself when operations appear to be inefficient.

Splay: to spread outward.

The *splay tree* quickly overcomes poor performance by rearranging the tree's nodes on the fly using a simple operation called a *splay*. Instead of performing careful analysis and optimally modifying the structure whenever a node is added or removed, the splay tree simply moves the referenced node to the top of the tree. The operation has the interesting characteristic that the average depth of the ancestors of the node to be splayed is approximately halved. As with skew heaps, the performance of a splay tree's operators, when amortized over many operations, is logarithmic.

The basis for the splay operation is a pair of operations called *rotations* (see Figure 12.4). Each of these rotations replaces the root of a subtree with one of

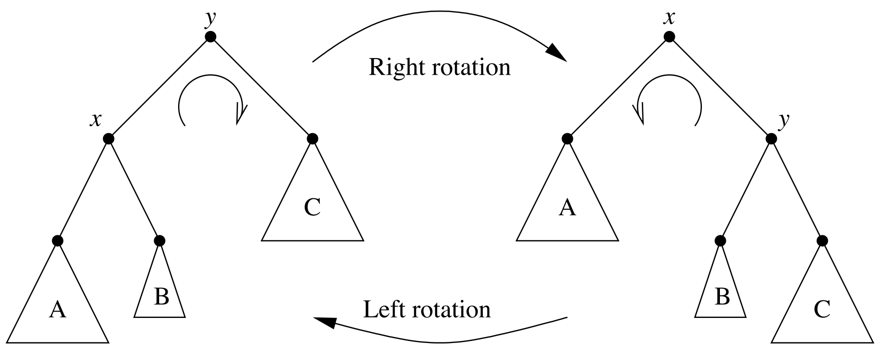


Figure 12.4 The relation between rotated subtrees.

its children. A right rotation takes a left child, x, of a node y and reverses their relationship. This induces certain obvious changes in connectivity of subtrees, but in all other ways, the tree remains the same. In particular, there is no structural effect on the tree above the original location of node y. A left rotation is precisely the opposite of a right rotation; these operations are inverses of each other.

The code for rotating a binary tree about a node is a method of the `BinTree` class. We show, here, `rotateRight`; a similar method performs a left rotation.

Finally, a right handed method!

```
def rotateRight(self):
    """Rotate this node (a left subtree) so it is the root."""
    parent = self.parent
    newRoot = self.left
    wasChild = parent != None
    wasLeft = self.isLeftChild
    # hook in new root (sets newRoot's parent, as well)
    self.left = newRoot.right
    newRoot.right = self
    if wasChild:
        if wasLeft:
            parent.left = newRoot
        else:
            parent.right = newRoot
```

For each rotation accomplished, the nonroot node moves upward by one level. Making use of this fact, we can now develop an operation to splay a tree at a particular node. It works as follows:

- If x is the root, we are done.
- If x is a left (or right) child of the root, rotate the tree to the right (or left)

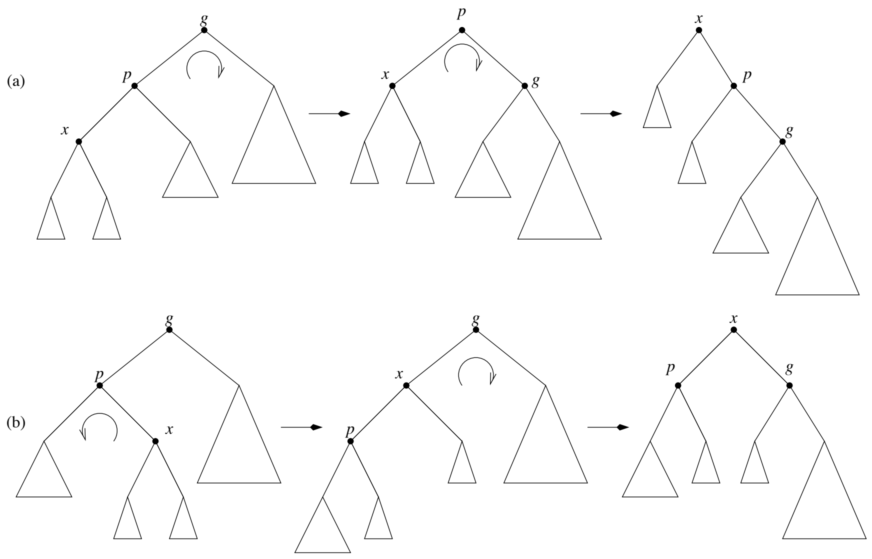


Figure 12.5 Two of the rotation pairs used in the splaying operation. The other cases are mirror images of those shown here.

about the root. x becomes the root and we are done.

- If x is the left child of its parent p, which is, in turn, the left child of its grandparent g, rotate right about g, followed by a right rotation about p (Figure 12.5a). A symmetric pair of rotations is possible if x is a left child of a left child. After double rotation, continue splay of tree at x with this new tree.
- If x is the right child of p, which is the left child of g, we rotate left about p, then right about g (Figure 12.5b). The method is similar if x is the left child of a right child. Again, continue the splay at x in the new tree.

After the splay has been completed, the node x is located at the root of the tree. If node x were to be immediately accessed again (a strong possibility), the tree is clearly optimized to handle this situation. It is *not* the case that the tree becomes more balanced (see Figure 12.5a). Clearly, if the tree is splayed at an extremal value, the tree is likely to be extremely unbalanced. An interesting feature, however, is that the depth of the nodes on the original path from x to the root of the tree is, on average, halved. Since the average depth of these nodes is halved, they clearly occupy locations closer to the top of the tree where they may be more efficiently accessed.

To guarantee that the splay has an effect on all operations, we simply perform each of the binary search tree operations as before, but we splay the tree at the node accessed or modified during the operation. In the case of `remove`, we splay the tree at the parent of the value removed.

12.6 Splay Tree Implementation

SplayTree

Because the splay tree supports the binary search tree interface, we extend the `SearchTree` data structure. Methods written for the `SplayTree` hide or *override* existing code inherited from the `SearchTree`. Here, for example, is the initializer:

```
@checkdoc
class SplayTree(SearchTree):
    def __init__(self,data=None,frozen=False):
        """Construct a SplayTree from an iterable source."""
        super().__init__(data,frozen)
```

As an example of how the splay operation is incorporated into the existing binary tree code, we look at the `__contains__` method. Here, the root is reset to the value of the node to be splayed, and the splay operation is performed on the tree. The postcondition of the splay operation guarantees that the splayed node will become the root of the tree, so the entire operation leaves the tree in the correct state.

```
def __contains__(self,item):
    """item is in SplayTree."""
    if self._root.empty:
        return False
    possibleLocation = self._locate(item)
    self._root = self._splay(possibleLocation)
    return item == possibleLocation.value
```

One difficulty with the splay operation is that it potentially modifies the structure of the tree. For example, the `__contains__` method—a method normally considered nondestructive—potentially changes the underlying topology of the tree. This makes it difficult to construct iterators that traverse the `SplayTree` since the user may use the value found from the iterator in a read-only operation that inadvertently modifies the structure of the splay tree. This *can* have disastrous effects on the state of the iterator. A way around this difficulty is to have the iterator keep only that state information that is necessary to help reconstruct—with help from the structure of the tree—the complete state of our traditional nonsplay iterator. In the case of the `__iter__`, we keep track of two references: a reference to an "example" node of the tree and a reference to the current node inspected by the iterator. The example node helps recompute the root whenever the iterator is reset. To determine what nodes would have been stored in the stack in the traditional iterator—the stack of unvisited ancestors of the current node—we consider each node on the (unique) path from the root to the current node. Any node whose left child is also on the path is an element of

It can also wreck your day.

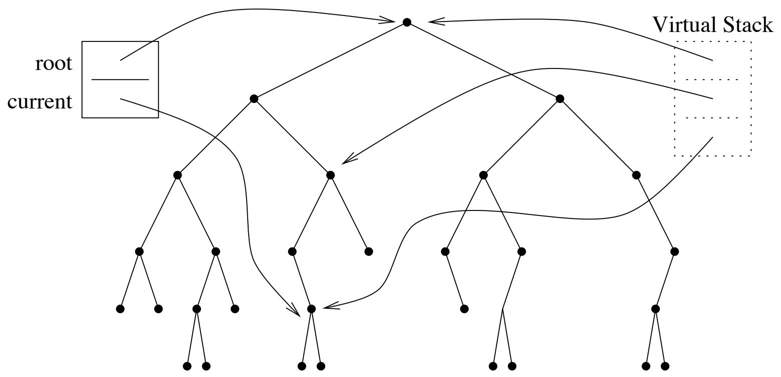


Figure 12.6 A splay tree iterator, the tree it references, and the contents of the virtual stack driving the iterator.

our “virtual stack.” In addition, the top of the stack maintains the current node (see Figure **??**).

REWRITE The constructor sets the appropriate underlying references and resets the iterator into its initial state. Because the `SplayTree` is dynamically restructuring, the root value passed to the constructor may not always be the root of the tree. Still, one can easily find the root of the current tree, given a node: follow parent pointers until one is `None`. Since the first value visited in an inorder traversal is the leftmost descendant, the reset method travels down the leftmost branch (logically pushing values on the stack) until it finds a node with no left child.

The current node points to, by definition, an unvisited node that is, logically, on the top of the outstanding node stack. Therefore, the `hasNext` and `get` methods may access the current value immediately.

All that remains is to move the iterator from one state to the next. The `next` method first checks to see if the current (just visited) element has a right child. If so, `current` is set to the leftmost descendant of the right child, effectively popping off the current node and pushing on all the nodes physically linking the current node and its successor. When no right descendant exists, the subtree rooted at the current node has been completely visited. The next node to be visited is the node under the top element of the virtual stack—the closest ancestor whose left child is also an ancestor of the current node. Here is how we accomplish this in Python:
The iterator is now able to maintain its position through splay operations.

Again, the behavior of the splay tree is logarithmic when amortized over a number of operations. Any particular operation may take more time to execute, but the time is usefully spent rearranging nodes in a way that tends to make the tree shorter.

From a practical standpoint, the overhead of splaying the tree on every operation may be hard to justify if the operations performed on the tree are relatively random. On the other hand, if the access patterns tend to generate degenerate binary search trees, the splay tree can improve performance.

12.7 An Alternative: Red-Black Trees

A potential issue with both traditional binary search trees and splay trees is the fact that they potentially have bad performance if values are inserted or accessed in a particular order. Splay trees, of course, work hard to make sure that repeated accesses (which seem likely) will be efficient. Still, there is no absolute performance guarantee.

One could, of course, make sure that the values in a tree are stored in as perfectly balanced a manner as possible. In general, however, such techniques are both difficult to implement and costly in terms of per-operation execution time.

Exercise 12.2 *Describe a strategy for keeping a binary search tree as short as possible. One example might be to unload all of the values and to reinsert them in a particular order. How long does your approach take to* `add` *a value?*

Because we consider the performance of structures using big-O notation, we implicitly suggest we might be happy with performance that is within a constant of optimal. For example, we might be happy if we could keep a tree balanced within a factor of 2. One approach is to develop a structure called a *red-black tree*.

For accounting purposes only, the nodes of a red-black tree are imagined to be colored red or black. Along with these colors are several simple rules that are constantly enforced:

1. Every red node has two black children.
2. Every leaf has two black (`BinTree.Empty` is considered black) children.
3. Every path from a node to a descendent leaf contains the same number of black nodes.

The result of constructing trees with these rules is that the height of the tree measured along two different paths cannot differ by more than a factor of 2: two red nodes may not appear contiguously, and every path must have the same number of black nodes. This would imply that the height of the tree is $O(\log_2 n)$.

Exercise 12.3 *Prove that the height of the tree with n nodes is no worse than $O(\log_2 n)$.*

Of course, the purpose of data abstraction is to be able to maintain the consistency of the structure—in this case, the red-black tree rules—as the structure is probed and modified. The methods `add` and `remove` are careful to maintain the red-black structure through at most $O(\log n)$ rotations and re-colorings of nodes. For example, if a node that is colored black is removed from the tree, it is necessary to perform rotations that either convert a red node on the path to the root to black, or reduce the *black height* (the number of black nodes from root to leaf) of the entire tree. Similar problems can occur when we attempt to add a new node that must be colored black.

The code for red-black trees can be found online as `RedBlackSearchTree`. While the code is too tedious to present here, it is quite elegant and leads to binary search trees with very good performance characteristics.

No longer true. Rewrite.

The implementation of the `RedBlackSearchTree` structure in the `structure` package demonstrates another approach to packaging a binary search tree that *is* important to discuss. Like the `BinTree` structure, the `RedBlackSearchTree` is defined as a recursive structure represented by a single node. The `RedBlackSearchTree` also contains a dummy-node representation of the empty tree. This is useful in reducing the complexity of the tests within the code, and it supports the notion that leaves have children with color, but most importantly, it allows the user to call methods that are defined even for red-black trees with no nodes. This approach—coding inherently recursive structures as recursive classes—leads to *side-effect free* code. Each method has an effect on the tree at hand but does not modify any global structures. This means that the user must be very careful to record any side effects that might occur. In particular, it is important that methods that cause modifications to the structure return the "new" value of the tree. If, for example, the root of the tree was the object of a `remove`, that reference is no longer useful in maintaining contact with the tree.

To compare the approaches of the `SearchTree` wrapper and the recursive `RedBlackSearchTree`, we present here the implementation of the `SymTab` structure we investigated at the beginning of the chapter, but cast in terms of `RedBlackSearchTrees`. Comparison of the approaches is instructive (important differences are highlighted with uppercase comments).

```
class RBSymTab(object):
    __slots__ = ["_table"]

    def __init__(self):
        self._table = RedBlackSearchTree()

    def __contains__(self,symbol):
        return KV(symbol,None) in self._table

    def __setitem__(self,symbol,value):
        a = KV(symbol,value)
        if a in self._table:
            self._table.remove(a)
        self._table.add(a)

    def __getitem__(self,symbol):
        a = KV(symbol,None)
        if a in self._table:
            a = self._table.get(a)
            return a.value
        else:
            return None

    def remove(self,symbol):
        a = KV(symbol, None)
        if a in self._table:
            a = self._table.get(a)
            self._table.remove(a)
            return a.value
        else:
            return None
```

The entire definition of `RedBlackSearchTrees` is available in the `structure` package, when $O(\log n)$ performance is desired. For more details about the structure, please see the documentation within the code.

12.8 Conclusions

A binary search tree is the product of imposing an order on the nodes of a binary tree. Each node encountered in the search for a value represents a point where a decision can be accurately made to go left or right. If the tree is short and fairly balanced, these decisions have the effect of eliminating a large portion of the remaining candidate values.

The binary search tree is, however, a product of the history of the insertion of values. Since every new value is placed at a leaf, the internal nodes are left untouched and make the structure of the tree fairly static. The result is that poor distributions of data can cause degenerate tree structures that adversely impact the performance of the various search tree methods.

To combat the problem of unbalanced trees, various rotation-based optimizations are possible. In splay trees, rotations are used to force a recently accessed value and its ancestors closer to the root of the tree. The effect is often to shorten degenerate trees, resulting in an amortized logarithmic behavior. A remarkable feature of this implementation is that there is no space penalty: no accounting information needs to be maintained in the nodes.

Chapter 13

Sets

It is often useful to keep track of an unordered collection of *unique* values. For this purpose, a *set* is the best choice. Python provides several built-in types that have set-like semantics, and we will investiate those in this chapter and the next. First, however, we think about the identity of objects.

13.1 The Set Abstract Base Class

The notion of a *set* in Python is a container that contains unique values that may be traversed. As an aid to programmers, the `Set` class found in the `collections` package's abstract base classes (`collections.abc.Set`) provides rich comparison operations as well as the simplest set operations. What is left abstract are three methods: `__contains__`, `__iter__`, and `__len__`. Sets that may be changed, objects that inherit from `MutableSet`, implement the three methods, along with basic methods for adding (`add`) and discarding (`discard`) values from a set allow the programmer to produce a rudimentary set.

Let's investigate the simplest of implementations: keeping a list of unique elements. Since we cannot make assumptions about the ordering of the elements, themselves, we keep an unordered list. Here is the sketch of a simple mutable class, `ListSet`:

```
from collections.abc import Set

class ListSet(Set):
    __slots__ = ["_data"]

    def __init__(self,data=None):
        """Create ListSet, possibly populated by an iterable."""
        self._data = []
        if data:
            for x in data:
                self.add(x)

    def add(self,value):
        """Add a value, if not already present."""
        if value not in self:
            self._data.append(value)
    def discard(self,value):
        """Remove a value, if present."""
        if value in self:
            self._data.remove(value)

    def clear(self):
        """Empty the set."""
        if self._data:
            self._data = []

    def __contains__(self,value):
        """Check for value in set."""
        return value in self._data

    def __iter__(self):
        """Return a traversal of the set elements."""
        return iter(self._data)

    def __len__(self):
        """Return number of elements in set."""
        return len(self._data)

    def __repr__(self):
        return "ListSet("+repr(self._data)+")"
```

To populate the set we incorporate values using `add` or, in the initializer, the element-by-element addition through iteration. Removal of values can be accomplished by `discard`, which removes the matching value if present and is silent otherwise,[1] and `clear`, which empties the `ListSet`. While `clear` is a generic method of the `MutableSet` class, it is often more efficient when overridden for each class.

Here are some interactions of sets that become possible, even with this simple implementation:

```
>>> a = ListSet(range(10))
>>> b = ListSet(range(5,7))
>>> a < b
False
>>> b < a
True
>>> a == b
False
>>> a == set(range(9,-1,-1))
True
>>> a == range(9,-1,-1)
```

[1] This is compared with the operation `remove`, which typically raises a `ValueError` if the value is not found.

```
False
>>> a - b
ListSet([0, 1, 2, 3, 4, 7, 8, 9])
>>> a | b
ListSet([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a.remove(5)
>>> a & b
ListSet([6])
>>> a.isdisjoint(b)
False
>>> b.clear()
>>> len(b)
0
```

Notice, by the way, that the set-combining operations generate new objects of the type found on the left side of the operation. Thus, the result of taking a `ListSet` and removing values found in a builtin `set` generate a new `ListSet`.

Because the `__contains__` method takes $O(n)$ time, many operations are linear. For example, `add` must first check for the presence of a value before performing a constant-time append of the value to the list. This means, for example, that initialization from an iterable source of n values takes $O(n^2)$ time. The reader is encouraged to consider ways that the performance of basic operations can be improved, though it will be necessary to override the default methods from the abstract base class to get the best performance.

The reason that `__contains__` method is slow is that the value to be located may be found in any position within the list. If, however, we could predict where the value would be found if it was there, the `contains` method might be a constant time operation. This observation is the basis for our next example of a set, a `BitSet`.

13.2 Example: Sets of Integers

Integers are represented in the computer as a binary number stored as a fixed length string of binary digits or *bits*. Typically, the *width* of an integer, or the number of bits used to encode a value, is 32 or 64. A *byte* has a width of 8 bits, allowing the storage of values between 0 and 255. Python supports a mutable list-like structure that contains only byte-sized integers. A number of Python operations we have not explored to this point are *bitwise operations*, like *bitwise and* (written `&`), *bitwise or* (written `|`), *bitwise complement* (written `~`), *bitwise shift left* (written `<<`), and *bitwise shift right* (written `>>`). These operations work directly on the binary representation of integers. We will make use of several simple observations about the manipulation of bits.

Observation 13.1 *The value 2^n can be computed as* `1<<n`.

This is a direct result of the fact that bit n (always counting from the right, starting at zero), has weight 2^n in the binary number system.

The bitwise-or operation simply computes a value whose individual bits are set if the bit is set in either (or both) of the composed values.

Observation 13.2 *To set bit* `n` *of integer* `i`*, we perform* `i = i | (1«n)`.

Often, the shorthand `|=` is used to update the integer value.

The bitwise-and operation produces a one in the result if there is a one in *both* of the composed values. This is a handy mechanism for *masking* values: wherever there is a 1 in the mask, the bits of the target of the mask are kept, otherwise they are cleared. The bit-complement operation (˜) computes the complement of each bit. These two operations give us the power to turn off bits:

Observation 13.3 *To clear bit* `n` *of integer* `i`*, we perform* `i = i & ˜(1«n)`.

Of course, if a bit is already cleared, this operation has little impact.

We now have the basis for storing a set of compactly storing a mutable set of integers. The presence of an integer in the set corresponds to a bit in a fixed position being set to 1. If that bit is not present, or if it is cleared (0), we take that to mean that the integer is not in the set. Because Python `byte` values can store 8 bits, we may store the presence of values between 0 and 7 in the setting of bits in byte 0 of the `bytearray`. The presence of integers 8 through 15 is represented by settings in byte 1, and so forth. It is useful to convert between integers and the *index* and bit *offset*. The private utility functions `_index` and `_offset` accomplish this task. In addition, it is helpful to be able to construct a value with a single bit set, which is accomplished with the `_bit` method.

```
def _index(n):
    """Compute array index of bit n."""
    return n//BitSet.WIDTH

def _offset(n):
    """Compute bit index of bit n in array element."""
    return n%BitSet.WIDTH

def _bit(n):
    """Generate a 1 in bit n, 0 otherwise."""
    return 1<<n
```

The value of `BitSet.WIDTH` would be 8 for this implementation, since we're using arrays of bytes to store the membership information. The conversion of index and bit offset back to an integer value is relatively simple: we simply multiply compute `index*BitSet.WIDTH+offset`.

The `BitSet` keeps track of two items: the underlying `bytearray`, `_bytes`, and the number of elements in the set, `_count`, which is otherwise expensive to determine (you might think about how this might be accomplished). The initialization of this structure makes use of the `clear` method, which is responsible for resetting the `BitSet` into an empty state.

```
class BitSet(MutableSet,Freezable):
    WIDTH = 8
    __slots__ = ["_bytes", "_count"]

    def __init__(self, data=None, frozen=False):
        """Construct a BitSet with elements optionally derived from data."""
        self.clear()
        if data:
            for x in data:
                self.add(x)

    def clear(self):
        """Discard all elements of set."""
        self._count = 0
        self._bytes = bytearray()
```

At any time the *extent* of the BitSet contains only the bytes that have been necessary to encode the presence of values from 0 to the maximum value encountered during the life of the BitSet. For this reason, it is always necessary to check to see if an operation is valid within the current extent and, on occasion, to expand the extend of the set to allow relatively large, new values to be added. The hidden method _probe checks the extent while _ensure expands the range of values that may be stored.

```
def _probe(self,n):
    """Verify bit n is in the extent of the set."""
    index = _index(n)
    return index < len(self._bytes)

def _ensure(self,n):
    """Ensure that bit n is within the capacity of the set."""
    index = _index(n)
    increment = index-(len(self._bytes)-1)
    if increment > 0:
        self._bytes.extend(increment*[0])
```

The action of the _ensure method is much the same as the extension of a list: new bytes are only added as they are needed. Because the bytearray extend operation might involve copying the entire array of bytes, an operation that is linear in the size of the set, it might be preferred to have the extent expanded through exponential growth. The reader is encouraged to revisit that discussion on page **??**.

The __contains__ operation is arguably the most important operation. It probes, possibly being able to eliminate the possibility if the _probe fails. If the _probe is successful the value may or may not be in the set, so it is important that we check that the particular bit is set to 1.

```
def __contains__(self,n):
    """n is in set.  Equivalent to __get_item__(n)."""
    if not self._probe(n): return False
```

```
        index = _index(n)
        bitnum = _offset(n)
        return 0 != (self._bytes[index] & _bit(bitnum))
```

While Python typically considers zero values and empty strings to be treated as False, that interpretation only occurs when a boolean value is needed. In this case, however, the result of the and (&) is an integer formed from the bits that result from the operation. The value returned, then, is an integer, so to convert it to a boolean value, the comparison with zero is necessary.

Two methods add and discard must set and clear bits. To add a value we first ensuring that the appropriate is available (if not already present) to be set.

```
    def add(self,n):
        """Add n to set."""
        self._ensure(n)
        index = _index(n)
        bitnum = _offset(n)
        if not (self._bytes[index] & _bit(bitnum)):
            self._bytes[index] |= _bit(bitnum)
            self._count += 1
```

Because bits are being turn on, the important operation, here, is or. Of course, we only add one to the _count when we actually expand the size of the BitSet.

The discard method removes values if present. It does not shrink the bytearray, though it could. Here, we use _probe to exit quickly if n could not possibly be in the BitSet. If the correct bit is present, it is set to 0 only if it is not currently 1.

```
    def discard(self,n):
        """Remove n from self, if present; remain silent if not there."""
        if not self._probe(n): return
        index = _index(n)
        bitnum = _offset(n)
        if self._bytes[index] & _bit(bitnum):
            self._bytes[index] &= ~(_bit(bitnum))
            self._count -= 1
```

Here, because we're trying to turn off the bit, we use an and operation with a mask that is missing the bit to be cleared.

As we have seen, the bitwise operations | and & are used to force bits to be 1 or 0. If, we know the current setting, it is possible to *toggle* or *complement* an individual bit with the *bitwise-exlusive or* operator, ^. Since, in both the add and discard operations, we know the current state and are simply interested in changing the state, both operations could use the exclusive-or operation in a symmetric manner.

The __iter__ function for BitSets is particularly interesting. Unlike other structures to be traversed we must examine, possibly, several locations before we yield a single value. We use the enumerate iteration to produce individual bytes and their respective index values. Only if a byte is not zero do we check the individual bits. If one of these bits is set, we reconstruct the corresponding number, based on its index (`i`) and bit offset (`b`). The expression `i*BitSet.WIDTH+b`, we note, is the combining operation that is the inverse of the `_index` and `_offset` hidden methods.

```
    def __iter__(self):
        """(lazy) iterator of all elements in set."""
        for i,v in enumerate(self._bytes):
            if v != 0:
                for b in range(BitSet.WIDTH):
                    if v & _bit(b):
                        yield (i*BitSet.WIDTH)+b
```

For very sparsely populated sets that cover a wide range of values, the `__iter__` method may loop many times between production of values.

The `BitSet` structure is an efficient implementation of a `Set` because members are stored in locations that are found at easily computed and dedicated locations. These characteristics allow us to add and discard values from sets in constant time. There are downsides, of course: sometimes we want to store values that are not integers, and sparse sets may take up considerable space. We now consider techniques that allow us to construct sets of objects other than integers with some bounds on the space utilization.

13.3 Hash tables

We will now spend some time developing a `MutableSet` implementation that stores values sparsely in a list. Each value is stored in a computable, predetermined location. However, because there may be an arbitrarily wide range of values stored in a fixed length structure, it becomes clear that locations may be targeted by more than one value. Our discussion, here, investigates how the target destination is computed and how we can manage the *collisions* that will naturally occur when set elements compete for storage locations.

13.3.1 Fingerprinting Objects: Hash Codes

We have seen many data structures that depend on being able to compare values. For example, in the `list` class, the `__contains__` method traverses the structure looking for an equivalent object. Because the comparison of values is dependent, heavily, on the structure of the values, the performance of operations like `__contains__` can be significantly impacted if the equality test is slow. In structures, like sets, where considerable time is spent comparing values it would be useful to think about ways that we might manage the cost of this important operation.

Since every data item in Python is an object, if it is known that every instance of a class—every object—is different than every other, we can quickly perform equality testing by simply comparing the *reference* to the object. Unfortunately, we rarely have the luxury of knowing that objects are unique (though, see problem **??** which discusses *interning* of strings and other objects).

Another approach is developing a manageable *fingerprint* of the object. When the this condensed representation of the object is a numeric value, it is called a *hash code*. Hash codes are not usually in one-to-one correspondence with object values, but a good method for determining hash codes will cause them to be shared by relatively few objects. While it is not absolutely necessary, the computation of a hash code for a value (the *hashing* of a value) is typically not more complex than the process of comparing values. Once a good hash code generated, equality testing can be reduced to the comparison of hash values. When hash values differ, it is proof that the fingerprinted objects must differ as well. When hash values are the same, a full equality test can be performed to eliminate the remote possibility that the two values are different, but hash to the same code.

If the computation of hash codes is expensive (as it might be to incorporate the features of a large object), the value can be cached within the object: a private attribute, initially `None`, is set to the hash code when it is computed the first time, and the resulting value is simply returned from that point onward.

Most data structures, however, are allowed to change over time. This means that two objects that were initially hashed and compared and found to be different can, over time, change to become equal. To avoid this situation, it is important that either we force the hash code be recomputed as the structure changes or we *freeze* the structure making it *immutable*. Because hash codes are stable for immutable structures in Python it is common to equate the notions of being *immutable* and *hashable*. Strictly speaking, mutable structures can be hashable as long as that part of the structure that determines equality is, in fact, immutable. For example, two medical records are the same as long as the patient identifiers are the same. Updating a medical record with new tests—features that do not effect record identification—does not impact equality testing or relevant hash codes. Indeed, a natural hash code for many records is simply the patient id; one may even change the *name* of the patient without impacting record location.

The Hashable Interface

The `Hashable` abstract base class, found in the `container.abc` package, promises the definition of the special method, `__hash__`. This method is called when the builtin function `hash` is called with an object as its sole parameter. Many builtin primitive types, including numeric types, booleans, and strings all define this method. Immutable container types, including `tuple` types cannot be directly modified and, thus, are hashable. Because the `list` is a dynamic container type, a hash code is not available. In those situations where one wishes to have a hashable list-like container, a `tuple` is usually first constructed from the desired `list`.

Because we think of the comparison of hash values as a first approximation to a full equality test of immutable objects, it is vital that the hash codes of equal values be equal.

Principle 11 *Equivalent objects (the same under* `__eq__`*) should return equivalent hash codes (as computed by* `__hash__`*).*

As was hinted at earlier, the development of appropriate hash methods can be tricky.[2] Developers look for methods that are fast, reproducible, and good at reducing collisions among data one might expect to see. For example, for `int` types, the hash code is simply the value of the integer, itself.[3] For boolean values, `hash(False)` is zero, while `hash(True)` is one. Hash of `float` types is related to the internal representation of the value, though float types that correspond to `int` values use the hash of the `int` value.

Example: Hashing Strings

Strings—objects of type `str`—are immutable, hashable objects of variable length. String comparison, of course, can be expensive, so an appropriate method of hashing strings is important to think about. Indeed, for many of the container classes we construct in this book the potential benefits of fast hash code generation parallel those of the `str` class.

Most of the approaches for hashing strings involve manipulations of the characters that make up the string. Fortunately, when a character is cast as an integer, the internal representation (often the ASCII encoding) is returned, usually an integer between 0 and 255.[4] Our first approach, then, might be to use the first character of the string. This has rather obvious disadvantages: the first letters of strings are not uniformly distributed, and there isn't any way of generating hash codes greater than 255. If we were to use the hash code as an index into a list or array, we would never expect to find values beyond location 255.

Our next approach would be to sum all the letters of the string. This is a simple method that generates large-magnitude hash codes if the strings are long. Even when large hash codes are used to index a small list or array, this is not a problem: we simply compute the location we would expect to find it if we counted by going around the array locations many times. This value is the *modulus* of dividing the integer by the size of the list. The main disadvantage of the summing technique is that if letters are transposed, then the strings generate the same hash values. For example, the string `"dab"` has $100+97+98=295$ as its sum of ASCII values, as does the string `"bad"`. The string `"bad"` and `"bbc"` are also equivalent under this hashing scheme. Figure 13.1 is a histogram of the number of words that might be found at or *hash to*, using this method, to each

[2] While it may be *tricky* to build good hash methods, bad hashing techniques do not generally cause failure, just bad performance.

[3] Actually, in CPython (a common version of Python on most platforms), the `int -1` hashes to `-2`. This is due to a technical constraint of the implementation of Python in C that is relatively unimportant to this discussion. Still, do not be surprised if, when you expect a hash value of `-1`, Python computes a value of `-2`.

[4] Python also transparently supports Unicode strings. The discussion, here, applied equally well there, but the range of character values may be considerably larger.

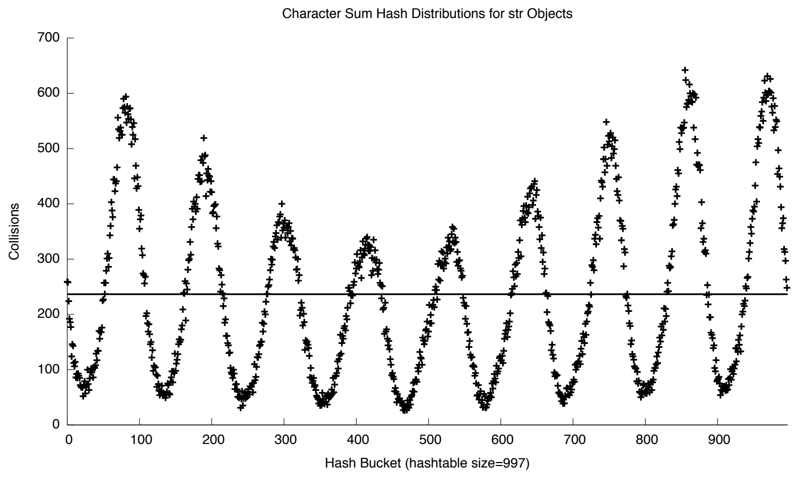


Figure 13.1 Numbers of words from the UNIX spelling dictionary hashing to each of the 997 buckets of a default hash table, if sum of characters is used to generate hash code.

slot of a 997 element list.[5] The periodic peaks demonstrate the fact that some slots of the table are heavily preferred over others. The performance of looking up and modifying values in the hash table will vary considerably, depending on the slot that is targeted by the hash function. Clearly, it would be useful to continue our search for a good mechanism.

Another approach might be to weight each character of the string by its position. To ensure that even very short strings have the potential to generate large hash values, we can provide exponential weights: the hash code for an l character string, s, is

$$\sum_{i=0}^{l-1} s[i]c^i$$

where c is usually a small integer value. When c is 2, each character is weighted by a power of 2, and we get a distribution similar to that of Figure 13.2. While this is closer to being uniform, it is clear that even with exponential behavior, the value of $c = 2$ is too small: not many words hash to table elements with large indices. When $c = 256$, the hash code represents the first few characters of the string exactly (see Figure 13.3).

[5] It is desirable to have the size of the list of target locations be a prime number because it spreads out the keys that might collide because of poor hash definition. For example, if every hash code were a multiple of 8, every cell in a 997 element array is a potential target. If the table had size 1024, any location that is not a multiple of 8 would not be targeted.

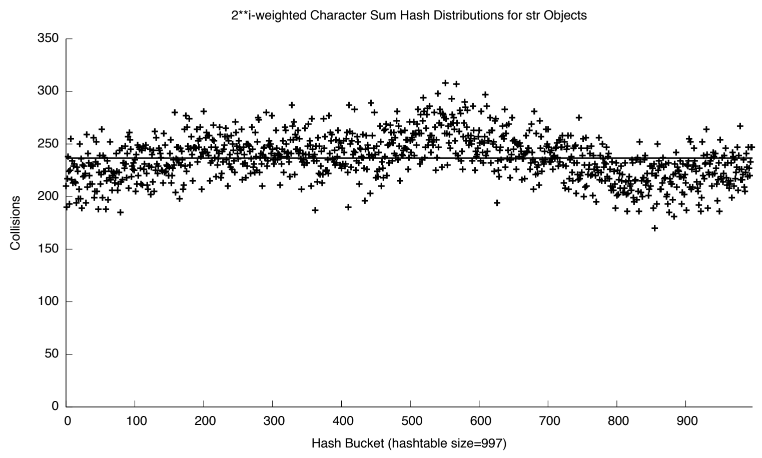


Figure 13.2 Frequency of dictionary words hashing to each of 997 buckets if characters are weighted by powers of 2 to generate hash code. Some periodicity is observable.

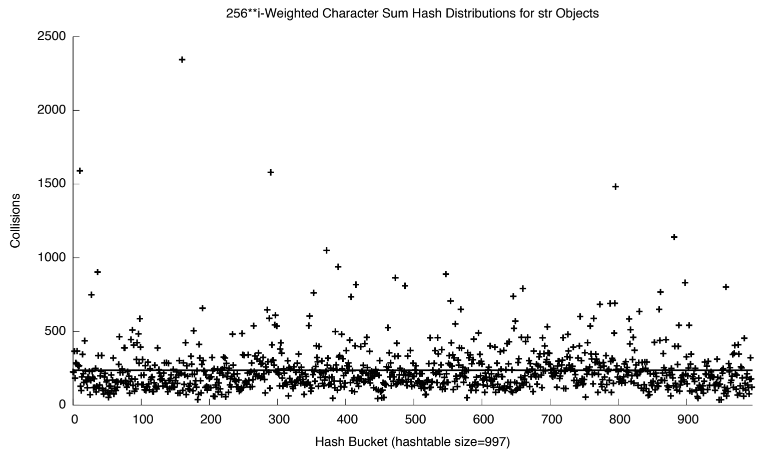


Figure 13.3 Frequency of words from dictionary hashing to each of 997 buckets if hash code is generated by weighting characters by powers of 256. Notice that bucket 160 has 2344 words, well above the expected average of 237.

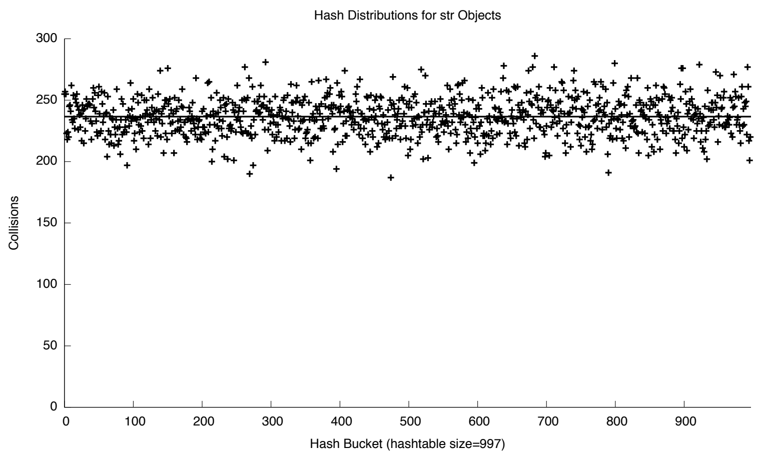


Figure 13.4 Frequency of words from dictionary hashing to each of 997 buckets, using the Python `str` hash code generation. The deviation is small and the distribution is seeded with a random constant determined each Python session.

The hashing mechanism currently used by Python approximately computes

$$r_0 * p^l + (\sum_{i=0}^{l-1} s[i]p^{l-i-1}) + l + r_1$$

where p is a prime[6] and r_0 and r_1 are random values selected when Python starts up (see Figure 13.4). Because p is prime, the effect of multiplying by powers of p is similar to shifting by a fractional number of bits; not only do the bits move to the left, but they change value, as well. The values r_0 and r_1 introduce an element of randomness that works against collisions of particular strings. It also means that the hash codes for individual runs of a program will differ.[7]

Example: Hash Values for Tuples

As we mentioned earlier, the computation of hash values for immutable container classes are computed in a manner that is very similar to that of `str` objects. The values combined in the hashing of strings, recall, were the internal representation of the string's individual characters. For tuples, the values combined are the *hash codes* associated with the elements of the tuple. In

[6] 1,000,003 in Python 3.3.
[7] This randomness can be turned off by setting the seed of the random number generator to zero.

Python 3.3, the hash computation for tuples computes a modified sum of hash codes whose weights are related to the *square* of the distance the element is from the end of the string.

Little is important to know about the technique other than to realize that the hash codes of each element contributes to the hash code for the tuple. Because of this, tuples that contain mutable values (like lists) do not have hash codes, because lists, themselves, are not hashable. In addition, in CPython (the standard for most platforms), hash codes can never be -1; if a hash of -1 is produced, it is mapped, automatically, to -2. The reason for this is an unimportant implementation decision, but it means that all tuples whose elements are each either -1 or -2 hash to exactly the same value!

13.4 Sets of Hashable Values

We now develop two different implementations of sets that allow one to store elements that are `Hashable`. The first approach provides a structure with constant time access. The second approach is more space efficient, but, as is usual with data structure tradeoffs, takes slightly more time.

13.4.1 HashSets

We now implement a `HashSet` whose values are stored sparsely in a list. All elements in the table are stored in a fixed-length list whose length is, ideally, prime. Initialization ensures that each slot within the array is set to `None`. Eventually, slots will contain hashable values. We use list for speed, but any container with constant time access to elements would be a logical alternative.

```
@checkdoc
class HashSet(MutableSet,Freezable):
    RESERVED = "Rent This Space"
    SIZES = [3, 7, 17, 37, 67, 127, 257, 509, 997, 1999, 4001, 8011,
             16001, 32003, 64007, 125003, 250007, 500009, 1000003]
    __slots__ = ["_data", "_capacity", "_maxload", "_dirty", "_counter", "_frozen", "_hash"]

    def __init__(self, data=None, capacity=17, maxload=0.6, frozen=False):
        """Construct a HashSet containing values in an iterable source."""
        # create empty main list
        self._capacity = _goodCapacity(capacity)
        self._maxload = maxload
        self._frozen = False
        self.clear()
        # store data, if given
        if data:
            for value in data:
                self.add(value)
        self._dirty = False
```

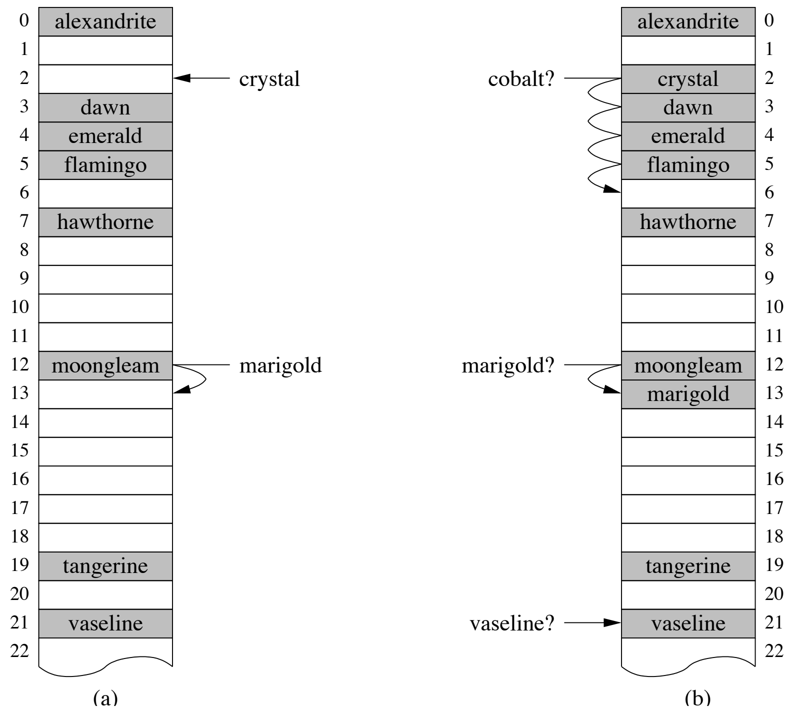


Figure 13.5 Hashing color names of antique glass. (a) Values are hashed into the first available slot, possibly after rehashing. (b) The lookup process uses a similar approach to possibly find values.

```
        self._frozen = frozen
        self._hash = None
```

The value management methods depend on a function, `_locate`, that finds a good location for a value in the structure. First, we use an index-producing function that "hashes" a value to a slot or bucket (see Figure 13.5). Since all `Hashable` objects have a `__hash__` function defined, we can use `hash(value)` for precisely this purpose. For simplicity, let's assume the hash code is the alphabet code ($a = 0$, $b = 1$, etc.) of the first letter of the word. The hash code for a particular key (2 for the word "crystal") is used as an index to the first slot to be considered for storing or locating the value in the table. If the slot is empty, the value can be stored there. If the slot is full, it is possible that another value already occupies that space (consider the insertion of "marigold" in Figure 13.5). When the keys of the two objects do not match, we have a *collision*. A *perfect hash function* guarantees that (given prior knowledge of the set of potential keys) no collisions will occur. When collisions do occur, they can

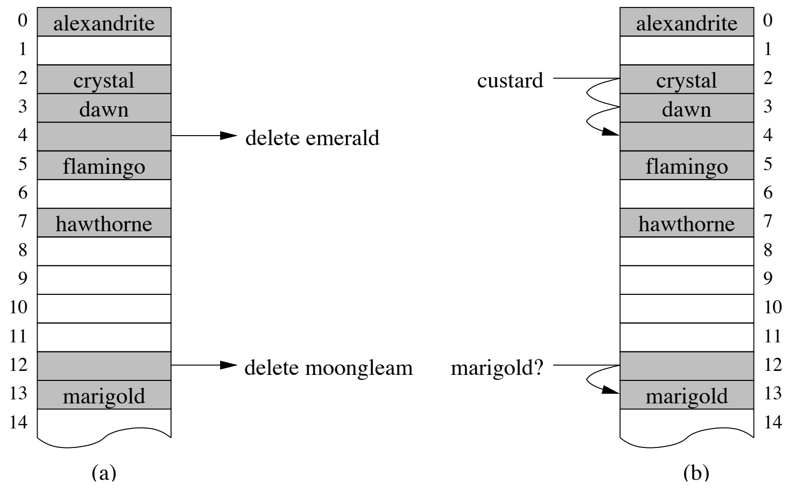


Figure 13.6 (a) Deletion of a value leaves a shaded reserved cell as a place holder. (b) A reserved cell is considered empty during insertion and full during lookup.

---

be circumvented in several ways. With *open addressing*, a collision is resolved by generating a new hash value, or *rehashing*, and reattempting the operation at a new location.

Slots in the hash table logically have two states—empty (`None`) or full (a reference to an object)—but there is also a third possibility. When values are removed, we replace the value with a *reserved* value that indicates that the location potentially impacts the lookup process for other cells during insertions. That association is represented by the empty shaded cell in Figure 13.6a. Each time we come across the reserved value in the search for a particular value in the array (see Figure 13.6b), we continue the search as though there had been a collision. We keep the first reserved location in mind as a possible location for an insertion, if necessary. In the figure, this slot is used by the inserted value "custard."

When large numbers of different-valued keys hash or rehash to the same locations, the effect is called *clustering* (see Figure 13.7). *Primary clustering* is when several keys hash to the same initial location and rehash to slots with potential collisions with the same set of keys. *Secondary clustering* occurs when keys that initially hash to different locations eventually rehash to the same sequence of slots.

In this simple implementation we use *linear probing* (demonstrated in Figures 13.5 to 13.7). Any rehashing of values occurs a constant distance from the last hash location. The linear-probing approach causes us to wrap around the array and find the next available slot. It does not solve either primary or secondary clustering, but it is easy to implement and quick to compute. To

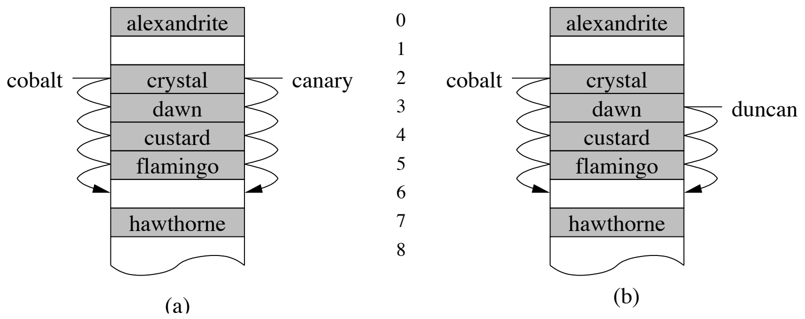


Figure 13.7 (a) *Primary clustering* occurs when two values that hash to the same slot continue to compete during rehashing. (b) Rehashing causes keys that initially hash to different slots to compete.

avoid secondary clustering we use a related technique, called *double hashing*, that uses a second hash function to determine the magnitude of the constant offset (see Figure 13.8). This is not easily accomplished on arbitrary keys since we are provided only one `__hash__` function. In addition, multiples and factors of the hash table size (including 0) must also be avoided to keep the `locate` function from going into an infinite loop. Still, when implemented correctly, the performance of double hashing can provide significant improvements over linear-probing.

We now discuss our implementation of hash tables. First, we consider the `_locate` function. Its performance is important to the efficiency of each of the public methods.

```
def _locate(self, value):
    """Return the list contained within self._data where value would be stored."""
    index = hash(value) % self._capacity
    reserved = None
    while True:
        if self._data[index] is HashSet.RESERVED:
            reserved = index
        else:
            if self._data[index] is None:
                return index if reserved is None else reserved
            if self._data[index] == value:
                return index
        index = self._rehash(value,index)

def _rehash(self,value,index):
    """Find another slot based on value and old hash index."""
    return (index+1)%self.capacity
```

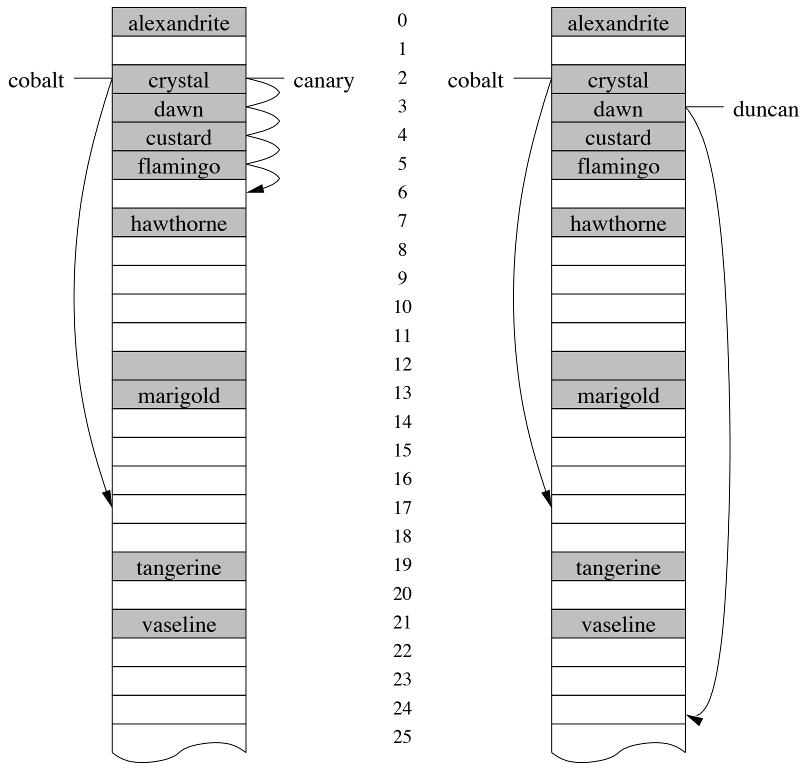


Figure 13.8 The keys of Figure 13.7 are rehashed by an offset determined by the alphabet code ($a = 1$, $b = 2$, etc.) of the *second* letter. No clustering occurs, but strings must have two letters!

To measure the difficulty of finding an empty slot by hashing, we use the *load factor*, α, computed as the ratio of the number of values stored within the table to the number of slots used. For open addressing, the load factor cannot exceed 1. As we shall see, to maintain good performance we should keep the load factor small as possible. Our maximum allowable load factor is a constant `_maxload`. Exceeding this value causes the list to be reallocated and copied over (using the method `_checkLoad`).

When a value is added, we simply locate the appropriate slot and insert the new value. If the ideal slot already has a value (it must have an equal key), we simply replace the value.

```
@mutatormethod
def add(self, value):
    """Add value to HashSet, regardless of if it is already present."""
```

```
        self._checkLoad(self._counter+1)
        index = self._locate(value)
        if _free(self._data[index]):
            self._counter += 1
        self._data[index] = value
        self._dirty = True
```

The `get` function works similarly—we simply return the value that matches in the table, or `None`, if no equivalent value could be found.

```
    def get(self, value):
        """Return object stored in HashSet that is equal to value."""
        if value not in self:
            raise KeyError("Value {} not found in HashSet.".format(value))
        index  = self._locate(value)
        return self._data[index]
```

The `__contains__` method is similar.

```
    def __contains__(self, value):
        """value is in this HashSet."""
        index = self._locate(value)
        return not _free(self._data[index])
```

To discard a value from the `HashSet`, we locate the correct slot for the value and, if found, we leave a reserved mark (`HashSet.RESERVED`) to maintain consistency in `_locate`.

```
    @mutatormethod
    def discard(self, value):
        """Remove value from HashSet, if present, or silently do nothing"""
        index = self._locate(value)
        if _free(self._data[index]):
            return
        self._data[index] = HashSet.RESERVED
        self._dirty = True
        self._counter = self._counter - 1
```

Hash tables are efficiently traversed. Our approach is based on the `list` iterator, yielding all values that are not "free"; values of `None` and `HashSet.RESERVED` are logically empty slots in the table. When the iterator is queried for a value, the underlying list is searched from the current point forward to find the next meaningful reference. The iterator must eventually inspect every element of the structure, even if very few of the elements are currently used.[8]

[8] The performance of this method could be improved by linking the contained values together. This would, however, incur an overhead on the `add` and `discard` methods that may not be desirable.

13.4.2 ChainedSets

Open addressing is a satisfactory method for handling hashing of data, if one can be assured that the hash table will not get too full. When open addressing is used on nearly full tables, it becomes increasingly difficult to find an empty slot to store a new value.

One approach to avoiding the complexities of open addressing—reserved values and table extension—is to handle collisions in a fundamentally different manner. *External chaining* solves the collision problem by inserting all elements that hash to the same bucket into a single collection of values. Typically, this collection is an unordered list. The success of the hash table depends heavily on the fact that the average length of the linked lists (the *load factor* of the table) is small and the inserted objects are uniformly distributed. When the objects are uniformly distributed, the *deviation* in list size is kept small and no list is much longer than any other.

Check this term

The process of locating the correct slot in an externally chained table involves simply computing the initial `hash` for the key and "modding" by the table size. Once the appropriate bucket is located, we verify that the collection is constructed and the value in the collection is updated. Because Python's `list` classes do not allow element retrieval by value, we may have to remove and reinsert the appropriate value.

```
@mutatormethod
def add(self, value):
    """Add value to ChainSet, regardless of if it is already present."""
    index = self._locate(value)
    bucket = self._data[index]
    if bucket is None:
        self._data[index] = []
        bucket = self._data[index]
    if value in bucket:
        bucket.remove(value)
    else:
        self._counter += 1
    bucket.append(value)
    self._dirty = True
```

Most of the other methods are implemented in a similar manner: they locate the appropriate bucket to get a `list`, then they search for the equivalent value within the `list`.

One method, `__iter__`, essentially requires the iteration over two dimensions of the hash table. One loop searches for buckets in the hash table—buckets that contain values—and an internal loop that explicitly iterates across the `list`. This is part of the price we must pay for being able to store arbitrarily large numbers of values in each bucket of the hash table.

```
def __iter__(self):
    for l in self._data:
        if l is not None:
            for v in l:
                yield v

def __len__(self):
    """The number of objects stored in ChainSet."""
    return self._counter

def __str__(self):
    """String representation of ChainSet."""
    return str(list(self))

def __repr__(self):
    """Evaluable representation of this ChainSet."""
    return "ChainSet({!r})".format(list(self))
```

At times the implementations appear unnecessarily burdened by the interfaces of the underlying data structure. For example, once we have found an appropriate value to manipulate, it is difficult to modify: instead it must be completely replaced. This is reasonable, though, since the immutability of the value is what helped us locate the bucket containing the association. If the value could be modified it might become inconsistent with its bucket's location.

Another subtle issue is the selection of the collection class associated with the bucket. Since linked lists have poor linear behavior for most operations, it might seem reasonable to use more efficient collection classes—for example, tree-based structures—for storing data with common hash codes. The graph of Figure 13.9 demonstrates the performance of various ordered structures when asked to construct collections of various sizes. It is clear that while `SplayTrees` provide better ultimate performance, the simple linear structures are more efficient when the structure size is in the range of expected use in chained hash tables (see Figure 13.10). When the average collection size gets much larger than this, it is better to increase the size of the hash table and re-insert each of the elements (this is accomplished with the `Hashtable` method, `extend`).

13.5 Freezing Structures

Sets constructed using hashing techniques are quite efficient, but they require that the elements of the sets be hashable. Unfortunately, if the elements are allowed to change, their hash values are likely to change. If this happens while a value is stored in a set based on hash value, the value becomes lost and the logic of the structure becomes inconsistent. What can be done if we wish to build sets of immutable types?

First, as we had mentioned previously, it is only important that the part of the structure encoded by the hashing mechanism must be immutable. For example, in our `KV` class, which stores key-value associations, the key portion of

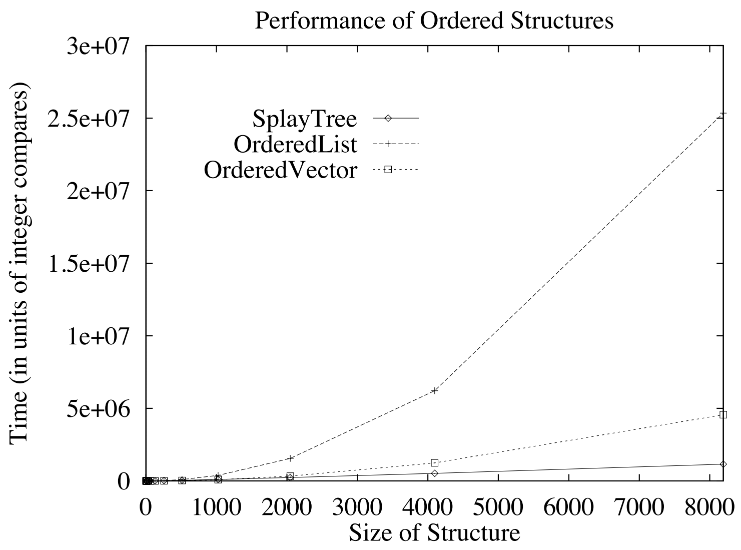


Figure 13.9 The time required to construct large ordered structures from random values.

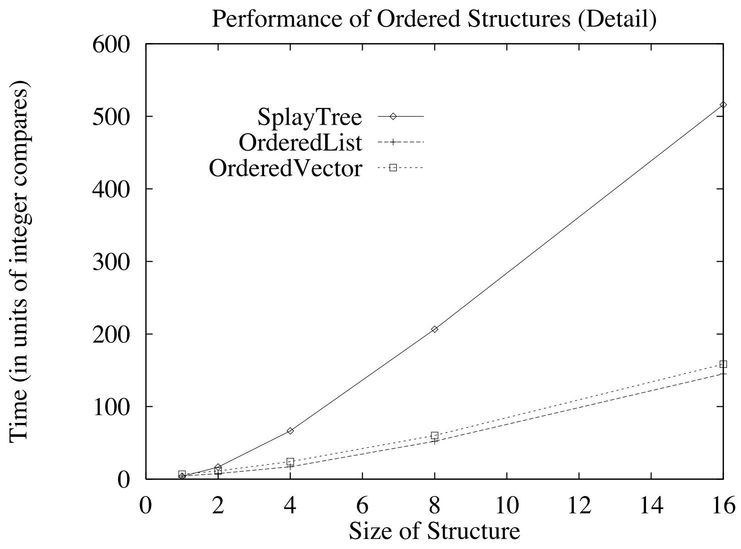


Figure 13.10 The time required to construct small ordered structures from random values.

the structure is immutable, but the value is not. Since KV types are considered equal if their keys are equal, the simplest effective hashing mechanism (and the one actually used) for a KV pair is to simply use the hash code of the key. For this reason, we can construct HashSets and ChainSets of KV types.

For other types that are mutable, it is sometimes useful to construct *frozen* versions that are immutable. For example, Python provides both sets and frozensets. The reason for this (and the implementors of sets were undoubtedly sensitive to this) is that we often want to construct *power sets* or sets of sets. For these purposes, (mutable) set objects are made immutable by constructing immutable frozenset copies as they are added to the main set structure. Unfortunately, the construction of frozen copies of structures is cumbersome enough that few of Python's classes have frozen alternates.

Another approach, and one we use in the structure package, is to provide each structure with the ability to *freeze* or become immutable. This process cannot be directly reversed other than by constructing a fresh, mutable copy.

As an example of the technique, we will now look more closely at the implementation of the LinkedList class. This class, recall, has all the semantics of Python's builtin list class: values can be added, removed, found and modified. To freeze a such a structure is to disable all methods that have the potential of modifying the state of the hashable part of the structure.

First, let's look at the __hash__ method, to see the scope of the problem:

```
@hashmethod
def __hash__(self):
    """Hash value for LinkedList."""
    if self._hash:
        return self._hash
    index = 73
    total = 91
    for v in self:
        try:
            x = hash(v)
        except TypeError as y:
            raise y
        total = total + (x * index)
        index = index + 1
    self._hash = total
    return total
```

Notice that the hash code for the structure is affected by the hash code for each of the elements contained within the list. If we are to freeze the list, we will have to freeze each of the elements that are contained. Freezing LinkedList structures is *deep* or *recursive*. If we fail to freeze an element of the structure, the element can be changed, thus affecting the hash code of the entire structure.

To indicate that a structure may be frozen, we have it extend the Freezable abstract base class. This requires two methods, freeze and frozen:

```
class Freezable(Hashable):
    """
    An abstract base class for objects that support structured freezing.
    """
    @abc.abstractmethod
    def freeze(self):
        ...

    @abc.abstractproperty
    def frozen(self):
        """Data structure is frozen."""
        ...
```

One approach to providing this interface is to supply with a hidden `_frozen` attribute. This boolean is `True` when the structure is immutable. When the structure is mutable, `_frozen` is `False`. During initialization, we carefully initialize these variables to allow the ability to populate the structure with values from an iterable source. Typically, we provide the ability to specify the value for a boolean keyword, `frozen`, to indicate that the structure is initially frozen.

```
def __init__(self, data=None, frozen=False):
    """Initialize a singly linked list."""
    self._hash = None
    self._head = None
    self._len = 0
    self._frozen = False
    if data is not None:
        self.extend(data)
    if frozen:
        self.freeze()
```

The two required methods for any container structure are typically written as they are for `LinkedLists`:

```
def freeze(self):
    for v in self:
        if isinstance(v,Freezable):
            v.freeze()
    self._frozen = True
    self._hash = hash(self)

@property
def frozen(self):
    return self._frozen
```

Notice how the structure of `freeze` parallels that of `__hash__`. Also, because freezing the structure will indirectly freeze the hash code, we compute and cache this value, though it's not logically necessary. Calling the `__hash__` function before the structure is frozen causes an exception to be raised. This behavior is indicated by the `@hashmethod` decorator.

```
@hashmethod
def __hash__(self):
    """Hash value for LinkedList."""
    if self._hash:
        return self._hash
    index = 73
    total = 91
    for v in self:
        try:
            x = hash(v)
        except TypeError as y:
            raise y
        total = total + (x * index)
        index = index + 1
    self._hash = total
    return total
```

During the life of a structure, its state is changed by methods that change or *mutate* the state. These methods, of course, should not be callable if the object is frozen. The structure package provides a method decorator, `@mutatormethod`, that indicates which methods change the structure's state.

```
    @mutatormethod
    def append(self, value):
        """Append a value at the end of the list (iterative version).

        Args:
           value: the value to be added to the end of the list.
        """
        ...
```

The effect is to wrap the function with a test to ensure that `self.frozen` is `False`. If, on the other hand, the structure is frozen, an exception is raised.

Once frozen, a structure cannot be unfrozen. It may, however, be used as a iterable source for initializing mutable structures. More efficient techniques are possible with the `copy` package and Python's serialization methods, called *pickling*.

Chapter 14

Maps

One of the most important primitive data structures of Python is the dictionary, `dict`. The dictionary is, essentially, a set of key-value pairs where the keys are unique. In abstract terms, a dictionary is an implementation of a *function*, where the domain of the function is the collection of keys, and the range is the collection of values.

In this chapter we look at a simple reimplementation of the builtin `dict` class, as well as a number of extensions of the notion of a map. First, we revisit the symbol table application to make use of a `dict`.

14.1 Mappings

A *mapping* is a data structure that captures the correspondence between values found in a *domain* of *keys* to members of a *range values*. For most implementations, the size of the range is finite, but it is not required. In Python, the abstract base class for immutable mappings is `collection.abc.Mapping`, which requires the implementation of a number of methods, the most important of which is `__getitem__`. Recall that this method provides support for "indexing" the structure. Mutable mappings require the implementation of methods that can modify the mapping including `__setitem__` and `__delitem__`.

The builtin class, `dict`, implements the methods required by `MutableMapping`. As an example of the use of these mappings, we revisit the symbol table example from the previous chapter.

14.1.1 Example: The Symbol Table, Revisited

Previously, we saw that we could use sets to maintain a list of key-value pairs. Here, we revisit that example, but we make use of the `dict`, a structure that manages the key-value association directly.

```
table = dict()
reading = True
for line in stdin:
    words = line.split()
    if len(words) == 0:
        continue
    if reading:
        if words[0] == 'END':
            reading = False
            print("Table contains aliases: {}".format(table.keys()))
        else:
            table[words[0]] = words[1]
    else:
        name = words[0]
        while name in table:
            name = table[name]
        print(name)
```

Here, we see that Python's native `dict` object can be used directly to maintain the symbol map structure. The semantics of indexing are extended to us to access value by an arbitrary hashable key. The `Mapping` interface demands, as well, the construction of three different types of *views*, or collections of data: `keys` returns a view of the valid keys in the dictionary, `items` returns a view of the valid `(key,value)` 2-tuples, and `values` returns a view of a collection of the values that correspond to keys. The values in this last view are not necessarily unique or hashable. The `keys` view is used in our symbol table application to identify those words that would be rewritten.

14.1.2 Unordered Mappings

The most direct implementation of dictionaries is based on an unordered set of key-value pairs. Recall that the keys of `KV` types are immutable. When they are hashable then the `KV` objects are hashable, as well. A dictionary, then, can be implemented as a `HashSet` of `KV` pairs. Let's walk through the details of this implementation which is most similar to Pythons builtin `dict` class.

In our implementation, a `Dictionary`, the we construct an empty set of key-value mappings. As with many of the structures we have seen, we can initialize the structure from another iterable object. Here, we have two different approaches to interpreting the data source. If it is an implementation of a `Mapping`, we can ask it for an item view: each element of this view is a 2-tuple that holds a key and a value. Otherwise, we consider the iterable to be an explicit source of 2-tuples interpreted in the same way. Each results in the setting of a key-value correspondence, the details of which we see later.

```
@checkdoc
class Dictionary(MutableMapping,Freezable):

    __slots__ = ["_kvset", "_frozen", "_dirty"]

    def __init__(self,data=None,capacity=10,frozen=False):
        """Construct a Dictionary from a mapping or an iterable source of key-value pairs."
        self._frozen = False
        self._dirty = False
        self._kvset = HashSet(capacity=capacity)
        if data:
            if isinstance(data,Mapping):
                for (key,value) in data.items():
                    self[key] = value
            else:
                try:
                    for (key,value) in data:
                        self[key] = value
                except:
                    raise TypeError("If data is an iterable, it must be able to generate key-value pai
        if frozen:
            self.freeze()
        self.dirty = False
```

Here, again, we handle freezing the data structure in the standard way.

Methods like `__len__` are directly inherited from the underlying `_set` variable.

The workhorse for adding new key-value pairs to the dictionary is the `__setitem__` mutator method. Recall this method is called whenever and expression like `d[key]=value` is encountered; the parameters to the method call are the key and value. The action of this method is to search for the unique `KV` object with a matching key and, if one is found, update its value. Otherwise a the key-value mapping is placed in a `KV` object.

```
@mutatormethod
def __setitem__(self,key,value):
    """Set the value associated with key to value."""
    kv = KV(key,value)
    if kv in self._kvset:
        self._kvset.get(kv).value = value
    else:
        self._kvset.add(kv)
```

The previous value associated with the key is no longer referenced by the dictionary and may be collected as garbage if it is not otherwise referenced. Note that the method is marked with the `@mutatormethod` decorator to cause the assignment to raise an exception if the `Dictionary` is frozen. The `setdefault` method works acts similarly, but the value provided is a value to be added to the dictionary only if the key does not already have a value. The final value associated with the key—either the existing value or the new "default value"—is returned.

To get the value associated with a key, the `__getitem__` is called with the search key. If a mapping from the key is found, the appropriate value is returned, otherwise nothing (`None`) is returned.

```
def __getitem__(self,key):
    """The value associated with key."""
    kv = KV(key,None)
    entry = self._kvset.get(kv)
    return entry.value if entry is not None else None
```

Notice that the appropriate `KV` pair is found by constructing a `KV` pair with a value of `None`. Since equality of `KV` items is determined by keys only, the value is unimportant (and cannot be predicted; that's what we're looking for!). The result is the actual mapping we need. The `get` method performs a similar operation, but has an optional second parameter which is the value to return is there was no matching key-value pair in the set.

Deletion of the an association is determined by the appropriate key. The `__delitem__` method removes the pair from the underlying set.

```
@mutatormethod
def __delitem__(self,key):
    """Remove key-value pair associated with key from Dictionary."""
    self._kvset.remove(KV(key,None))
```

Notice that we use `remove` that, unlike set's `discard` method, will raise an exception if the item is not found. This is the behavior we desire in the `MutableMapping` method. The `pop` method works similarly, but returns the value associated with the key removed.

A number of other straightforward methods are required to meet the specification of a `MutableMapping`. The `clear` method removes all the key-value associations. The easiest approach is to `clear` the underlying set.

```
def clear(self):
    """Remove all contents of Dictionary."""
    self._kvset.clear()
```

As with the set and list implementations, a `popitem` method is provided that allows us to pick an arbitrary mapping element to remove and return. In most implementations, the pair returned by `popitem` is simply the first one that would be encountered in a traversal of the mapping.

```
@mutatormethod
def popitem(self):
    """Remove and return some key-value pair as a tuple.
    If Dictionary is empty, raise a KeyError."""
    if len(self) == 0:
        raise KeyError("popitem(): dictionary is empty")
    item = next(iter(self.items()))
    (key,value) = item
    self._kvset.remove(KV(key,None))
    return item
```

Because `MutableMapping`s store key-value pairs, the value returned is an arbitrary pair formed into a `(key,value)` 2-tuple.

Views are objects that allow one to iterate (often multiple times) across a subordinate *target* structure. When changes are made to the target, they are immediately reflected in the view. Every view supports `__len__` (it's `Sized`), `__contains__` (it's a `Container`), and `__iter__` (it's `Iterable`). All `Mapping` types provide three methods that construct *views* of the underlying structure: `items`, which returns a view of key-value tuples, `keys`, which returns a view of the keys in the structure, and `values`, which returns a view of all the values that appear in the structure. In the symbol map example, we saw how the `keys` method could be used to generate a list of valid keys to the map. The `Mapping` class provides generic implementations of each of the methods `items`, `keys`, and `values` similar to the following:

```
def items(self):
    """View over all tuples of key-value pairs in Dictionary."""
    return ItemsView(self)

def keys(self):
    """View over all keys in Dictionary."""
    return KeysView(self)

def values(self):
    """View over all values in Dictionary."""
    return ValuesView(self)
```

Each type of view has a corresponding abstract base class that provides generic implementation of the appropriate functionality. These classes—`ItemsView`, `KeysView`, and `ValuesView`—are found in the `collections` package. Views don't typically have internal state (other than a reference to the target structure), but the right circumstance (for example, when the target structure is frozen) they provide an opportunity for efficiently caching read-only views of structures when on-the-fly traversals with iterators are computationally costly.

An important benefit of the use of the `HashSet` class is the ability of this set implementation to gracefully grow as the number of entries in the mapping increases. Since most uses of mapping will actually use a surprisingly small number of key-value pairs, we initially set the capacity (the number of buckets in the `HashSet`) to be relatively small. As the number of mapping entries increases, this capacity increases as well. It is, then, relatively unimportant that we estimate the capacity of the `Mapping` very accurately.

However, if we were to depend on a `ChainSet`, whose capacity determines the fixed number of buckets to be used during the life of the structure, the performance will degrade quickly if the capacity was underestimated. This is because the performance of the structures is largely determined by the structure that manages the multiple entries found in each bucket.

Add figure with degrading chained set performance.

14.2 Tables: Sorted Mappings

One of the costs of having near-constant time access to key-value pairs using hashing is the loss of order. If the keys can be ordered, it is natural to want to have the key-value pairs stored in sorted order. In the structure package we call sorted mappings, *tables*. The implementation of table structures is not logically much different than the implementation of dictionaries, we simply keep our `KV` pairs in a `Sorted` structure. Because of its (expected) speed, the `SplayTree` is a natural target for such an implementation, thought any `Sorted` structure would work.

Internally, the `Table` collect the `KV` pairs in the `_sorted` attribute. Since the table is a container, it is necessary to implement the `__container__` method, which simply passes on a request for a KV with an unspecified value:

```
    def __contains__(self,key):
        """key is in table."""
        kv = KV(key,None)
        return kv in self._sorted
```

This method has expected logarithmic behavior since the SplayTree has that characteristic.

Most of the other methods, like __getitem__ parallel their Dictionary counterparts. The downside, of course, is that the performance of Table methods is driven, primarily, by the performance of SplayTree equivalents. For example, to check to see if a key is in a table, we must perform a logarithmic (expected) search. To update a value in the table, a search is first performed, taking logarithmic time. Whether or not this is successful, the update takes constant time: the splay tree has reoririented itself to be particularly fast for operations related to the last search. So, while there are many operations that are logarithmic, the price is usually paid once, no matter the complexity of the type of access performed.

14.3 Combined Mappings

We often think of mappings as static implementations of deterministic functions. Seen in this light, there are a number of useful ways we can combine mappings.

First is an idea that naturally appears in the support of scoping for variables in languages like Python: ChainMaps in the collections package. A ChainMap is a container and a mapping whose elements are, themselves, mappings. When you look up a key, each subordinate mapping is searched, in turn, until an associated value is found and returned. When new associations are established, they are always written to the first mapping in the container. This structure is motivated by the need to construct a combined dictionary of visible variables from within Python.

Details needed. Only reasonable for immutable mappings?

The next idea is *function composition*.

14.4 Conclusions

Chapter 15

Graphs

RELATIONS ARE OFTEN AS USEFUL AS DATA. The process of building and accessing a data structure can be thought of as a means of effectively focusing the computation. Linear structures record the history of their accesses, sorted structures perform incremental ordering, and binary trees encode decisions about the partitioning of collections of data.

The most general mechanism for encoding relations between data is the *graph*. Simple structures, like arrays, provide implicit connections, such as adjacency, between stored values. Graphs are more demanding to construct but, as a result, they can encode more detailed information. Indeed, the versatility of graphs allows them to represent many of the most difficult theoretical problems of computer science.

This chapter investigates two traditional implementations of graphs, as well as several standard algorithms for analyzing their structure. We first agree on some basic terminology.

15.1 Terminology

A *graph* G consists of a collection of *vertices* $v \in V_G$ and relations or *edges* $(u, v) \in E_G$ between them (see Figure 15.1). An edge is *incident to* (or *mentions*) each of its two component vertices. A graph is *undirected* if each of its edges is considered a set of two unordered vertices, and *directed* if the mentioned vertices are ordered (e.g., referred to as the *source* and *destination*). A graph S is a *subgraph* of G if and only if $V_S \subseteq V_G$ and $E_S \subseteq E_G$. Simple examples of graphs include the *list* and the *tree*.

In an undirected graph, the number of edges (u, v) incident to a vertex u is its *degree*. In a directed graph, the outgoing edges determine its *out-degree* (or just *degree*) and incoming edges its *in-degree*. A *source* is a vertex with no incoming edges, while a *sink* is a vertex with no outgoing edges.

Two edges (u, v) and (v, w) are said to be *adjacent*. A *path* is a sequence of n distinct, adjacent edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$. In a *simple path* the vertices are distinct, except for, perhaps, the *end points* v_0 and v_n. When $v_0 = v_n$, the simple path is a *cycle*.

Two vertices u and v are *connected* (written $u \leadsto v$) if and only if a simple path of the graph mentions u and v as its end points. A subgraph S is a *connected component* (or, often, just a *component*) if and only if S is a largest subgraph of G such that for every pair of vertices $u, v \in V_S$ either $u \leadsto v$ or $v \leadsto u$. A

Components are always connected.

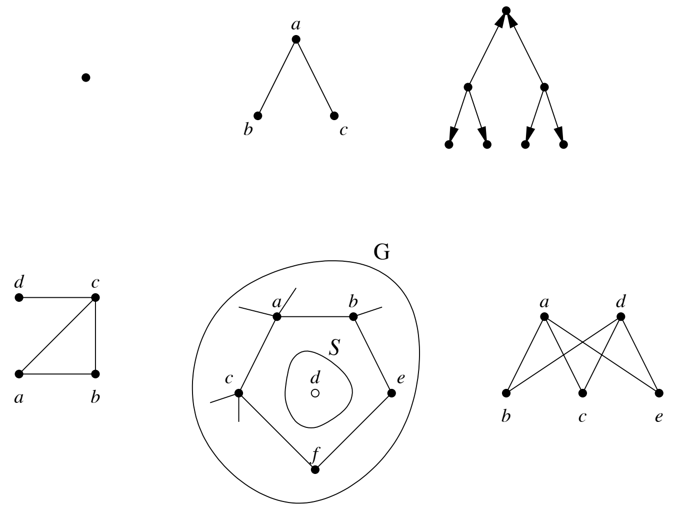


Figure 15.1 Some graphs. Each node a is adjacent to node b, but never to d. Graph G has two components, one of which is S. The directed tree-shaped graph is a directed, acyclic graph. Only the top left graph is complete.

connected component of a directed graph G is *strongly connected* if $u \leadsto v$ and $v \leadsto u$ for all pairs of vertices $u, v \in V_S$.

A graph containing no cycles is *acyclic*. A directed, acyclic graph (*DAG*) plays an important role in solving many problems. A *complete graph* G contains an edge (u, v) for all vertices $u, v \in V_G$.

15.2 The Graph Interface

Vertices of a graph are usually labeled with application-specific information. As a result, our implementations of a graph structure depend on the user specifying unique labels for vertices. In addition, edges may be labeled, but not necessarily uniquely. It is common, for example, to specify weights or lengths for edges. All the graph implementations allow addition and removal of vertices and edges:

```
@checkdoc
class Graph(Sized,Freezable,Iterable):
    slots = ["_frozen", "_hash", "_directed"]

    def __init__(self, directed):
        """Create unfrozen graph."""

    @property
    def frozen(self):
        """Graph is frozen."""

    def freeze(self):
        """Freeze graph to prevent adding and removing vertices and edges and
        to allow hashing."""

    @abc.abstractmethod
    def add(self, label):
        """Add vertex with label label to graph."""

    @abc.abstractmethod
    def add_edge(self, here, there, edge_label):
        """Add edge from here to there with its own label edge_label to graph."""
    @abc.abstractmethod
    def remove(self, label):
        """Remove vertex with label label from graph."""

    @abc.abstractmethod
    def remove_edge(self, here, there):
        """Remove edge from here to there."""
    @abc.abstractmethod
    def get(self, label):
        """Return actual label of indicated vertex.
        Return None if no vertex with label label is in graph."""

    @abc.abstractmethod
    def get_edge(self, here, there):
        """Return actual edge from here to there."""

    @abc.abstractmethod
    def __contains__(self, label):
        """Graph contains vertex with label label."""

    @abc.abstractmethod
    def contains_edge(self, here, there):
        """Graph contains edge from vertex with label here to vertex with
        label there."""

    @abc.abstractmethod
    def visit(self, label):
        """Set visited flag on vertex with label label and return previous
        value."""

    @abc.abstractmethod
    def visit_edge(self, edge):
        """Set visited flag on edge and return previous value."""

    @abc.abstractmethod
    def visited(self, label):
        """Vertex with label label has been visited."""

    @abc.abstractmethod
    def visited_edge(self, edge):
        """edge has been visited."""

    @abc.abstractmethod
    def reset(self):
        """Set all visited flags to false."""

    @abc.abstractmethod
    def __len__(self):
        """The number of vertices in graph."""

    @abc.abstractmethod
    def degree(self, label):
        """The number of vertices adjacent to vertex with label label."""

    @abc.abstractproperty
    def edge_count(self):
        """The number of edges in graph."""

    @abc.abstractmethod
    def __iter__(self):
        """Iterator across all vertices of graph."""

    @abc.abstractmethod
    def vertices(self):
        """View of all vertices in graph."""

    @abc.abstractmethod
    def neighbors(self, label):
        """View of all vertices adjacent to vertex with label label.
        Must be edge from label to other vertex."""

    @abc.abstractmethod
    def edges(self):
        """View across all edges of graph."""
    @abc.abstractmethod
    def clear(self):
        """Remove all vertices from graph."""
    @property
    def empty(self):
        """Graph contains no vertices."""
    @property
    def directed(self):
        """Graph is directed."""

    @abc.abstractmethod
    def __hash__(self):
        """Hash value for graph."""
    def __eq__(self, other):
        """self has same number of vertices and edges as other and all
        vertices and edges of self are equivalent to those of other."""
```

Because edges can be fully identified by their constituent vertices, edge operations sometimes require pairs of vertex labels. Since it is useful to implement both directed and undirected graphs, we can determine the type of a specific graph using the `directed` method. In undirected graphs, the addition of an edge effectively adds a directed edge in both directions. Many algorithms keep track of their progress by visiting vertices and edges. This is so common that it seems useful to provide direct support for adding (`visit`), checking (`visited`), and removing (`reset`) marks on vertices and edges.

Two iterators—generated by `__iter__` and `edges`—traverse the vertices and edges of a graph, respectively. A special iterator—generated by `neighbors`—traverses the vertices adjacent to a given vertex. From this information, outbound edges can be determined.

Before we discuss particular implementations of graphs, we consider the abstraction of vertices and edges. From the user's point of view a vertex is a label. Abstractly, an edge is an association of two vertices and an edge label. In addition, we must keep track of objects that have been visited. These features of vertices and edges are independent of the implementation of graphs; thus we commit to an interface for these objects early. Let's consider the `_Vertex` class.

```
@checkdoc
class _Vertex(object):

    """An abstract base class for _MatrixVertex and _ListVertex."""

    __slots__ = ["_label", "_visited", "_hash"]

    def __init__(self, label):
        """Create unvisited _Vertex."""
        self._label = label
        self._visited = False
        self._hash = None

    @property
    def label(self):
        """The label of this _Vertex."""
        return self._label

    def visit(self):
        """Set visited flag true and return its previous value."""
        old_visited = self.visited
        self._visited = True
        return old_visited

    def reset(self):
        """Set visited flag false."""
        self._visited = False

    @property
    def visited(self):
        """This _Vertex has been visited."""
        return self._visited

    def __eq__(self, other):
        """Label is equivalent to label of other."""
        if not isinstance(other, _Vertex):
            return False
        return self.label == other.label

    def __str__(self):
        """String representation of the _Vertex."""
        return repr(self)

    def __hash__(self):
        """Hash value for _Vertex."""
        if self._hash is None:
            try:
                self._hash = hash(self.label)
            except TypeError as y:
                raise y
        return self._hash
```

This class is similar to a `KV` pair: the label portion of the `_Vertex` cannot be modified, but the visited flag can be freely set and reset. Two `_Vertex` objects are considered equal if their labels are equal. It is a bare-bones interface. It should also be noted that the `_Vertex` is a nonpublic class (thus the leading underscore). Since a `_Vertex` is not visible through the `Graph` interface, there is no reason for the user to have access to the `_Vertex` class.

Because the `Edge` class is a visible feature of a `Graph` interface (you might ask why—see Problem **??**), the `Edge` class is a visible declaration:

```
@checkdoc
class Edge(object):
```

```
"""An edge used by Graph classes to join two vertices."""

slots = ["_here", "_there", "_label", "_directed", "_visited", "_frozen", "_hash"]

def __init__(self, here, there, label=None, directed=False):
    """Create unvisited, unfrozen Edge."""
    self._here = here
    self._there = there
    self._label = label
    self._directed = directed
    self._visited = False
    self._frozen = False
    self._hash = None

@property
def frozen(self):
    """Edge is frozen."""
    return self._frozen

def freeze(self):
    """Freeze Edge."""
    self._frozen = True

@property
def here(self):
    """Starting vertex."""
    return self._here

@property
def there(self):
    """Ending vertex."""
    return self._there

@property
def label(self):
    """The label of this Edge."""
    return self._label

@label.setter
@mutatormethod
def label(self, value):
    """Set label to value."""
    self._label = value

def visit(self):
    """Set visited flag true and return its previous value."""
    old_visited = self.visited
    self._visited = True
    return old_visited

@property
def visited(self):
    """This Edge has been visited."""
    return self._visited

@property
def directed(self):
    """This Edge is directed."""
    return self._directed

def reset(self):
    """Set visited flag false."""
    self._visited = False

def __eq__(self, other):
    """Other is an Edge and self starts and finishes at the same two
    vertices as other (in the same order, if self is directed)."""
    if not isinstance(other, Edge):
        return False
    # for any edge, if here and there match, edges are equal
    if (self.here == other.here) and (self.there == other.there):
        return True
    # undirected edges could have here and there in opposite order
    if not self.directed:
        if (self.here == other.there) and (self.there == other.here):
            return True
    # no other way for edges to be equal
    return False

@hashmethod
def __hash__(self):
    """Hash value for Edge."""
    if self._hash is None:
        try:
            self._hash = hash(self.here) + hash(self.there) + hash(self.label)
        except TypeError as y:
            return y
    return self._hash

def __repr__(self):
    """Parsable string representation for Edge."""
    string = "Edge(" + repr(self.here)
    string = string + "," + repr(self.there)
    if self.label:
        string = string + "," + repr(self.label)
    if self.directed:
        string = string + ",directed=True"
    string = string + ")"
    return string
```

As with the _Vertex class, the Edge can be constructed, visited, and reset. Unlike its _Vertex counterparts, an Edge's label may be changed. The methods here and there provide access to labels of the vertices mentioned by the edge. These method names are sufficiently ambiguous to be easily used with undirected edges and convey a slight impression of direction for directed edges. Naming of these methods is important because they are used by those who wish to get vertex information while traversing a (potentially directed) graph.

15.3 Implementations

Now that we have a good feeling for the graph interface, we consider traditional implementations. Nearly every implementation of a graph has characteristics of one of these two approaches. Our approach to specifying these implementations, however, will be dramatically impacted by the availability of object-oriented features. We first remind ourselves of the importance of abstract base classes in Python and as a feature of our design philosophy.

As "traditional" as this science gets, anyway!

15.3.1 Abstract Classes Reemphasized

Normally, when a class is declared, code for each of the methods must be provided. Then, when an instance of the class is constructed, each of the methods can be applied to the resulting object. As is common with our design approach, however, it is useful to partially implement a class and later finish the implementation by *extending* the class in a particular direction. The partial base class is *abstract*; it cannot be directly constructed because some of the methods are not completely defined. The extension to the class *inherits* the methods that have been defined and specifies any incomplete code to make the class *concrete*.

Again, we use an abstract base class to enforce a common interface in our various graph implementations. Our approach will be to provide all the code that can be written without considering the particular graph implementation. For example, whether or not a graph is directed is a property that is native to all implementations, and is directly supported by concrete code:

```
slots = ["_frozen", "_hash", "_directed"]

def __init__(self, directed):
    """Create unfrozen graph."""
    self._hash = None
```

```
        self._frozen = False
        self._directed = directed
    @property
    def directed(self):
        """Graph is directed."""
        return self._directed
```

When we must write code that is dependent on particular design decisions, we delay it by decorating an abstract header for the particular method. For example, we will need to add edges to our graph, but the implementation depends on whether or not the graph is directed. Looking ahead, here is what the declaration for `add_edge` looks like in the abstract `Graph` class:

```
    @abc.abstractmethod
    def add_edge(self, here, there, edge_label):
        """Add edge from here to there with its own label edge_label to graph."""
        ...
```

That's it! It is simply a *promise* that code will eventually be written.

Once the abstract class is described as fully as possible, we implement (and possibly extend) it, in various ways, committing in each case, to a particular approach of storing vertices and edges. Because we declare the class `GraphMatrix` to be an extension of the `Graph` class, all the code written for the abstract `Graph` class is inherited; it is as though it had been written for the `GraphMatrix` class. By providing the missing pieces of code (tailored for our matrix implementation graphs), the extension class becomes concrete. We can actually construct instances of the `GraphMatrix` class.

A related concept, *subtyping*, allows us to use any extension of a class wherever the extended class could be used. We call the class that was extended the *base type* or *superclass*, and the extension the *subtype* or *subclass*. Use of subtyping allows us to write code thinking of an object as a `Graph` and not necessarily a `GraphList`. Because `GraphMatrix` is an extension of `Graph`, it *is* a `Graph`. Even though we cannot construct a `Graph`, we can correctly manipulate concrete subtypes using the methods described in the abstract class. In particular, a call to the method `add_edge` calls the method of `GraphMatrix` because each object carries information that will allow it to call the correct implementation.

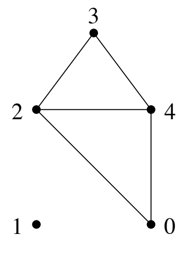
In parallel, we develop an abstract base class for private `_Vertex` implementations which, in each graph implementation, is made concrete by a dedicated extension to a hidden, specific vertex class.

We now return to our normally scheduled implementations!

15.3.2 Adjacency Matrices

An $n \times n$ matrix of booleans is sufficient to represent an arbitrary graph of relations among n vertices. We simply store `True` in the boolean at matrix location $[u][v]$ to represent the fact that there is an edge between u and v (see Figure 15.2), and `False` otherwise. Since entries $[u][v]$ and $[v][u]$ are independent, the representation is sufficient to describe directed graphs as well. Our convention is that the first index (the row) specifies the source and the second

Beware: Edges on the diagonal appear exactly once.

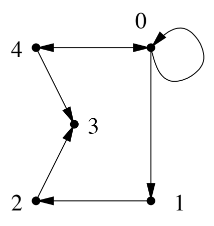


(a)

	0	1	2	3	4
0	F	F	T	F	T
1	F	F	F	F	F
2	T	F	F	T	T
3	F	F	T	F	T
4	T	F	T	T	F

(b)

Figure 15.2 (a) An undirected graph and (b) its adjacency matrix representation. Each nontrivial edge is represented twice across the diagonal—once in the gray and once in the white—making the matrix symmetric.



(a)

Destination

Source	0	1	2	3	4
0	T	T	F	F	T
1	F	F	T	F	F
2	F	F	F	T	F
3	F	F	F	F	F
4	T	F	F	T	F

(b)

Figure 15.3 (a) A directed graph and (b) its adjacency matrix representation. Each edge appears exactly once in the matrix.

index (the column) indicates the destination. To represent undirected graphs, we simply duplicate the entry $[u][v]$ at entry $[v][u]$.[1] This is called an *adjacency matrix* representation of a graph. The undirected and directed graphs of Figures 15.2a and 15.3a, respectively, are represented by the matrices of Figures 15.2b and 15.3b.

One difficult feature of our implementation is the arbitrary labeling of vertices and edges. To facilitate this, we maintain a mapping that translates a vertex label to a `_Vertex` object. To help each vertex keep track of its associated index we extend the `_Vertex` class to a `_List`Vertex that includes methods that

[1] If the vertices are totally ordered—say we can determine that $u < v$—then we can reduce the cost of representing edges by a factor of two by storing information about equivalent edges (u, v) and (v, u) with a boolean at $[u][v]$.

manipulate an `_index` field. Each index is a small integer that identifies the dedicated matrix row *and* column that maintains adjacency information about each vertex. To help allocate the indices, we keep a free list (a set) of available indices.

One feature of our implementation has the potential to catch the unwary programmer by surprise. Because we use a mapping to organize vertex labels, it is important that the vertex label class implement the `__hash__` function in such a way as to guarantee that if two labels are equal (using the `__eq__` method), they have the same hash code.

We can now consider the protected data and initializer for the `GraphMatrix` class:

GraphMatrix

```
@checkdoc
class GraphMatrix(Graph):

    __slots__ = ["_size", "_capacity", "_edge_count", "_matrix",
                 "_vertex_dict", "_free"]

    def __init__(self, vertices=None, edges=None, directed=False):
        """Construct a GraphMatrix from an iterable source.  Specify if the graph is direct
        super().__init__(directed)
        # current number of vertices in graph
        self._size = 0
        # maximum number of vertices ever in graph
        self._capacity = 0
        # current number of edges in graph
        self._edge_count = 0
        # _matrix is list of lists indicating which vertices are adjacent to which others
        self._matrix = []
        # _vertex_dict maps vertex label:vertex object
        self._vertex_dict = {}
        # _free lists free indices
        self._free = set()
        if vertices:
            for vertex in vertices:
                self.add(vertex)
        if edges:
            for edge in edges:
                if len(edge) == 2:
                    self.add_edge(edge[0], edge[1])
                elif len(edge) == 3:
                    self.add_edge(edge[0], edge[1], edge[2])
                else:
                    raise KeyError("Incorrect parameters for initializing edge.")
```

The initializer first calls the initializer from the `Graph` class to initialize that part of the `Graph` class that is concrete (for example, the `_directed` property). This is important; failing to perform this initialization at the start will leave the graph in an inconsistent state.

By default, these Graph instances are undirected. By specifying the directed keyword, one can change this behavior for the life of the graph. The graph can also be seeded with an initial set of vertices and/or edges.

The add method adds a vertex. If the vertex already exists, the operation does nothing. If it is new to the graph, an index is allocated from the free list, a new _MatrixVertex object is constructed, and the label-vertex association is recorded in the _matrix map. The newly added vertex mentions no edges, initially.

GraphMatrix

```
@mutatormethod
def add(self, label):
    """Add vertex with label label to graph."""
    if label not in self:
        # index is last available space in _free
        # if _free is empty, index is len(self)
        if len(self._free) != 0:
            index = self._free.pop()
        else:
            index = len(self)
            # grow matrix: add row, column
            self._capacity = self._capacity + 1
            for there in self._matrix:
                there.append(None)
            self._matrix.append([None] * self._capacity)
        # increasing current size
        self._size = self._size + 1
        # create _Vertex object
        vertex = _MatrixVertex(label, index)
        # add vertex to _vertex_dict
        self._vertex_dict[label] = vertex
```

An important part of making the matrix space efficient is to keep a list of matrix indices (_free) that are not currently in use. When all indices are in use and a new vertex is added, the matrix is extended by adding a new row and column. The new index immediately used to support the vertex entry.

Removing a vertex reverses the add process. We must, however, be sure to set each element of the vertex's matrix row and column to None, removing any mentioned edges (we may wish to add a new, isolated vertex with this index in the future). When we remove the vertex from the _matrix map and we "recycle" its index by adding it to the list of free indices. As with all of our remove methods, we return the previous value of the label. (Even though the labels match using __eq__, it is likely that they are not precisely the same. Once returned, the user can extract any unknown information from the previous label before the value is collected as garbage.)

```
@mutatormethod
def remove(self, label):
    """Remove vertex with label label from graph."""
```

```
        if label in self:
            # get the vertex object and ask for its index
            index = self._vertex_dict[label].index
            # set all entries in row index of _matrix to None
            # can’t travel from here to any other vertex
            # count number of edges being removed and adjust current_edges accordingly
            count = 0
            for i in range(len(self._matrix[index])):
                if self._matrix[index][i]:
                    count = count + 1
                    self._matrix[index][i] = None
            self._edge_count = self._edge_count - count
            # set all entries in column index of each _matrix row to None
            # can’t travel from any other vertex to here
            count = 0
            for there in self._matrix:
                if there[index]:
                    count = count + 1
                    there[index] = None
            # in directed graph, these edges were counted separately
            # so need to subtract them from current_edges
            if self.directed:
                self._edge_count = self._edge_count - count
            # current size is decreasing
            self._size = self._size - 1
            # add index to _free
            self._free.add(index)
            # remove vertex from _vertex_dict
            del self._vertex_dict[label]
```

Within the graph we store references to Edge objects. Each Edge records all of the information necessary to position it within the graph, including whether it is directed or not. This allows the equals method to work on undirected edges, even if the vertices were provided in the opposite order (see Problem **??**). To add an edge to the graph, we require two vertex labels and an edge label. The vertex labels uniquely identify the vertices within the graph, and the edge label (which may be None). This Edge reference (which is, effectively, True) is the value inserted within the matrix at the appropriate row and column. To add the edge, we construct a new Edge with the appropriate information. This object is written to appropriate matrix entries: undirected graphs update one or two locations; directed graphs update just one. Here is the add_edge method:

```
    @mutatormethod
    def add_edge(self, here, there, edge_label=None):
        """Add edge from here to there with its own label edge_label to graph."""
        if not here in self:
            self.add(here)
        if not there in self:
            self.add(there)
        # get vertices from _vertex_dict to have access to their exact labels
        here_vertex = self._vertex_dict[here]
        there_vertex = self._vertex_dict[there]
        # create edge
        new = Edge(here_vertex.label, there_vertex.label, edge_label, directed=self.directed)
        # increment edge count if this is a new edge
        if not self.contains_edge(here, there):
            self._edge_count = self._edge_count + 1
        # put new into _matrix[here_vertex.index][there_vertex.index]
        self._matrix[here_vertex.index][there_vertex.index] = new
        # if undirected, also put new into _matrix[there_vertex.index][here_vertex.index]
        if not self._directed:
            self._matrix[there_vertex.index][here_vertex.index] = new
```

One might wonder if the performance of the implementation might be improved by extending a single `GraphMatrix` class in two directions: one for directed graphs and another for undirected graphs. The differences are quite minor. While the two different subtypes allow us to write specialized code without performing explicit run-time tests, these tests are replaced by less obvious decreases in performance due to subtyping. Still, the logical complexity of code sometimes can be dramatically reduced through class extension.

The `remove_edge` method removes and returns the label associated with the `Edge` found between two vertices.

```
    @mutatormethod
    def remove_edge(self, here, there):
        """Remove edge from here to there."""
        if self.contains_edge(here, there):
            here_vertex = self._vertex_dict[here]
            there_vertex = self._vertex_dict[there]
            old_edge = self._matrix[here_vertex.index][there_vertex.index]
            self._matrix[here_vertex.index][there_vertex.index] = None
            if not self._directed:
                self._matrix[there_vertex.index][here_vertex.index] = None
            self._edge_count = self._edge_count - 1
        return old_edge.label
```

The `get`, `get_edge`, `__contains__`, and `contains_edge` methods return information about the graph in an obvious way. Modifying the objects returned by these methods can be dangerous: they have the potential of invalidating the state of the underlying graph implementation.

```
    def get(self, label):
        """Return actual label of indicated vertex.
        Return None if no vertex with label label is in graph."""
        if label in self:
            return self._vertex_dict[label]
        return None

    def get_edge(self, here, there):
        """Return actual edge from here to there."""
        if self.contains_edge(here, there):
            here_vertex = self._vertex_dict[here]
            there_vertex = self._vertex_dict[there]
            return self._matrix[here_vertex.index][there_vertex.index]

    def __contains__(self, label):
        """Graph contains vertex with label label."""
        return label in self._vertex_dict

    def contains_edge(self, here, there):
        """Graph contains edge from vertex with label here to vertex with label there."""
        if here in self and there in self:
            here_vertex = self._vertex_dict[here]
            there_vertex = self._vertex_dict[there]
            return self._matrix[here_vertex.index][there_vertex.index] is not None
        # if here and there aren’t valid vertices, no edge between them
        return False
```

Each of the visit-type methods passes on requests to the underlying object. For example, the visit method simply refers the request to the associated Vertex:

GraphMatrix

```
    def visit(self, label):
        """Set visited flag on vertex with label label and return previous value."""
        if label in self:
            vertex = self._vertex_dict[label]
            return vertex.visit()
```

The process of resetting the visitation marks on a graph traverses each of the vertices and edges, resetting them along the way.

But I reiterate myself.

Let’s now consider the implementation of each of the three traversals. Each of these, as with the mapping classes, returns a *view* on the graph. The first, generated by vertices, allows one to traverse the vertex labels. To do that, we implement VerticesView, a view that traverses the labels returned by the graph iterator. For the GraphMatrix, this __iter__ simply returns values encountered in the underlying vertex dictionary:

```
    def __iter__(self):
        """Iterator across all vertices of graph."""
        for vertex in self._vertex_dict.keys():
            yield vertex
```

The neighbors method returns a NeighborsViewMatrix, which traverses the edges adjacent to a single vertex, considers only the outgoing edges. We look up the index associated with the specified vertex label and scan across the matrix row yielding all vertices that appear opposite the queried vertex:

```
class NeighborsViewMatrix(Iterable):

    """A view of the neighbors of a specific vertex.
    Returned by the neighbors method of a GraphMatrix."""

    __slots__ = ["_target", "_label"]

    def __init__(self, target, label):
        self._target = target
        self._label = label

    def __iter__(self):
        g = self._target
        if self._label in g:
            # the _Vertex object with label label
            vertex = g._vertex_dict[self._label]
            row = g._matrix[vertex.index]
            for edge in [e for e in row if e is not None]:
                if edge.here != self._label:
                    yield edge.here
                else:
                    yield edge.there

    def __len__(self):
        return self._target.degree(self._label)
```

All that remains is to construct an iterator over the edges of the graph. In this code we see a good example of the use of views: a single view is constructed of all the values found in the vertex dictionary. This view is traversed by two iterators in a nested loop, so it is important that these iterators act independently. This approach ensures that we only consider those parts of the matrix that are actively used to support the connectivity of the graph. From these entries, we return those that are not `None`; these are edges of the graph. For directed graphs, we return every `Edge` encountered. For undirected graphs, we return only those edges that are found in the upper triangle of the matrix (where the column index is at least as large as the row index).

```
@checkdoc
class EdgesViewMatrix(Iterable):
    """A view of all the edges of a GraphMatrix.
    Returned by the edges method of a GraphMatrix."""

    __slots__ = ["_target"]

    def __init__(self, target):
        self._target = target
    def __iter__(self):
        verts = self._target._vertex_dict.values()
        for here in verts:
            row = self._target._matrix[here.index]
            for there in verts:
                if self._target.directed or (there.index >= here.index):
                    this_edge = row[there.index]
                    if this_edge:
                        yield this_edge

    def __len__(self):
        return self._target.edge_count
```

The great advantage of the adjacency matrix representation is its simplicity. The access to a particular edge in a graph of size n can be accomplished in constant time. Other operations, like `remove`, appear to be more complex, taking $O(n)$ time. The disadvantage is that the implementation may vastly overestimate the storage required for edges. While we have room for storing $O(n^2)$ directed edges, some graphs may only need to make use of $O(n)$ edges. Graphs with superlinear numbers of edges are called *dense;* all other graphs are *sparse*. When graphs are sparse, most of the elements of the adjacency matrix are not used, leading to a significant waste of space (and, for some traversals, time). Our next implementation is particularly suited for representing sparse graphs.

15.3.3 Adjacency Lists

Recalling the many positive features of a linked list over a fixed-size array, we now consider the use of an *adjacency list*. As with the adjacency matrix representation, we maintain a `Map` for identifying the relationship between a vertex label and the associated `Vertex` object. Within the vertex, however, we store a collection (usually a linked list) of edges that mention this vertex. Figures 15.4 and 15.5 demonstrate the adjacency list representations of undirected and directed graphs. The great advantage of using a collection is that it stores only edges that appear as part of the graph.

As with the adjacency matrix implementation, we construct a privately used extension to the `_Vertex` class, `_ListVertex`. In this extension we reference a collection of edges that are incident to this vertex. In directed graphs, we collect edges that mention the associated vertex as the source. In undirected graphs any edge incident to the vertex is collected. Because the edges are stored within the vertices, most of the actual implementation of graphs appears within the implementation of the extended vertex class. We see most of the implementation here:

```
@checkdoc
class _ListVertex(_Vertex):
    """Private vertex class used in GraphList."""
```

GraphListVertex

```
    __slots__ = ["_adjacencies"]

    def __init__(self, label):
        """Create a vertex with no adjacencies."""
        super().__init__(label)
        # store edges adjacent to this vertex
        # in directed graphs, store edges with this vertex as here (edges from this vertex)
        self._adjacencies = []

    def add_edge(self, edge):
        """Add edge object to adjacencies list."""
        if not self.contains_edge(edge):
            self._adjacencies.append(edge)

    def remove_edge(self, edge):
        """Remove and return edge from adjacencies equal to edge."""
        # graph knows if it is directed or undirected and passes information to edge
        # then edge __eq__() method handles equals accordingly
        return_val = None
        for adj_edge in self._adjacencies:
            if adj_edge == edge:
                return_val = adj_edge
                break
        if return_val:
            self._adjacencies.remove(return_val)
        return return_val

    def contains_edge(self, edge):
        """edge is adjacent to this vertex."""
        return edge in self._adjacencies

    def get_edge(self, edge):
        """Return edge from adjacencies equal to edge."""
        for adj_edge in self._adjacencies:
            if adj_edge == edge:
                return adj_edge

    @property
    def degree(self):
        """Number of edges adjacent to this vertex."""
        return len(self._adjacencies)

    def adjacent_vertices(self):
        """Iterator over adjacent vertices."""
        # each edge in adjacencies either has this vertex as here or there
        # and a unique adjacent vertex as the other
```

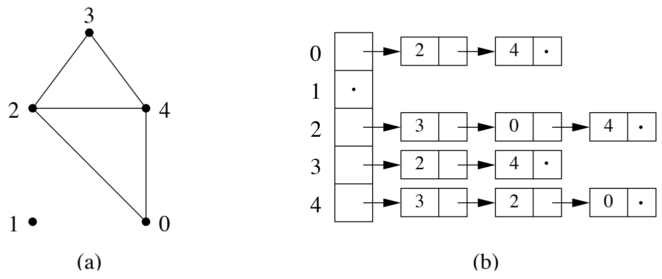


Figure 15.4 (a) An undirected graph and (b) its adjacency list representation. Each edge is represented twice in the structure. (Compare with Figure 15.2.)

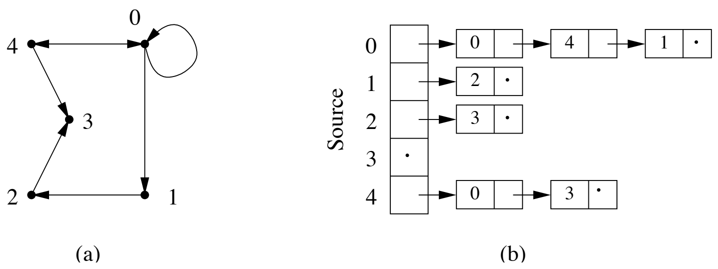


Figure 15.5 (a) A directed graph and (b) its adjacency list representation. Each edge appears once in the source list. (Compare with Figure 15.3.)

```
for edge in self._adjacencies:
    if edge.here != self.label:
        yield edge.here
    else:
        yield edge.there
```

The constructor initializes the concrete attributes of `_Vertex`, and then constructs an empty adjacency list. Elements of this list will be `Edge` objects. Most of the other methods have obvious behavior.

The only difficult method is `get_edge`. This method returns an edge from the adjacency list that is logically equivalent (i.e., is determined to be equal through a call to `Edge`'s `__eq__` method) the edge provided. In an undirected graph the order of the vertex labels may not correspond to the order found in edges in the edge list. As a result, `get_edge` returns a *canonical edge* that represents the edge specified as the parameter. This ensures that there are not multiple instances of edges that keep track of shared information.

We are now ready to implement most of the methods required by the `Graph` interface. First, we consider the protected `GraphList` initializer:

```
@checkdoc
class GraphList(Graph):

    __slots__ = ["_vertex_dict"]

    def __init__(self, vertices=None, edges=None, directed=False):
        """Construct a GraphList from an iterable source.  Specify if the graph is directed."""
        super().__init__(directed)
        self._vertex_dict = {}
        if vertices:
            for vertex in vertices:
                self.add(vertex)
        if edges:
            for edge in edges:
                if len(edge) == 2:
                    self.add_edge(edge[0], edge[1])
                elif len(edge) == 3:
                    self.add_edge(edge[0], edge[1], edge[2])
                else:
                    raise KeyError("Incorrect parameters for initializing edge.")
```

Our approach to extending the abstract `GraphList` type to support directed and undirected graphs is similar to that described in the adjacency matrix implementation.

Adding a vertex (if it is not already there) involves simply adding it to the underlying mapping.

```
@mutatormethod
def add(self, label):
    """Add vertex with label label to graph."""
    if label not in self:
        vertex = _ListVertex(label)
        self._vertex_dict[label] = vertex
```

To remove a vertex, we have to work a little harder. Because the node may be mentioned by edges, it is important that we identify and remove those edges before the vertex is removed from the vertex dictionary. This is a two step process. First, we know that all edges that appear in the adjacency list associated with the node, itself, must be removed. This is easily accomplished by simply clearing the list. The second step involves a traversal of all of the other adjacency lists to find edge where either the source (`here`) or destination (`there`) is the label of the node we are removing. A subtle aspect of this process is the capturing of the adjacency list before edges are removed; remember that modifying a structure while it is being traversed can lead to unpredicatable results.

To add an edge to the graph we insert a reference to the `Edge` object in the appropriate adjacency lists. For a directed graph, we insert the edge in the list associated with the source vertex. For an undirected graph, a reference to the edge must be inserted into *both* lists. It is important, of course, that a *reference* to a single edge be inserted in both lists so that changes to the edge are maintained consistently.

```
@mutatormethod
def add_edge(self, here, there, edge_label=None):
    """Add edge from here to there with its own label edge_label to graph."""
    self.add(here)
    self.add(there)
    # get vertices from _vertex_dict
    here_vertex = self._vertex_dict[here]
    there_vertex = self._vertex_dict[there]
    # create edge
    new = Edge(here_vertex.label, there_vertex.label, edge_label, directed=self.directed)
    # tell here vertex that this edge is adjacent
    here_vertex.add_edge(new)
    # in undirected graph, this edge is also adjacent to there
    if not self.directed:
        there_vertex.add_edge(new)
```

Removing an edge is simply a matter of removing it from the source and (if the graph is undirected) the destination vertex adjacency lists.

```
@mutatormethod
def remove_edge(self, here, there):
    """Remove edge from here to there."""
    if self.contains_edge(here, there):
        here_vertex = self._vertex_dict[here]
        there_vertex = self._vertex_dict[there]
        # create edge object that will be equal to the edge we want to remove from each ver
        edge = Edge(here_vertex.label, there_vertex.label, directed=self.directed)
        old_edge = here_vertex.remove_edge(edge)
        # in directed graph, this step will do nothing:
        if not self.directed:
            there_vertex.remove_edge(edge)
        return old_edge.label
    else:
        return None
```

Notice that to remove an edge a "pattern" edge must be constructed to identify (through `__eq__`) the target of the remove.

Many of the remaining edge and vertex methods have been greatly simplified by our having extended the `Vertex` class. Here, for example, is the `degree` method:

```
def degree(self, label):
    """The number of vertices adjacent to vertex with label label."""
    if label in self:
            return self._vertex_dict[label].degree
```

This code calls the `_ListVertex` `degree` method. That, in turn, returns the length of the underlying adjacency list. Most of the remaining methods are simply implemented.

At this point, it is useful to discuss the implementation of iterators for the adjacency list representation. Like the adjacency matrix implementation, the `__iter__` method simply returns the iterator for the underlying mapping (recall that an iterator across a mapping returns the list of valid keys). Each of the values returned by the iterator is a vertex label, which is exactly what we desire.

The `neighbors` iterator should return a view of the collection of neighbors of the provided vertex as seen by outgoing edges. This is, simply, the set of edges that appear in the vertex's adjacency list (that is, by definition, the meaning of the adjacency list). Our implementation is a bit paranoid: it generates an empty traversal if the vertex is not in currently in the graph. The reason for this is that because this is a view, the underlying graph may change in a way that may no longer contain the vertex.

The view constructed by the `edges` is made complex by the fact that a single edge will be seen twice in an undirected graph. To account for this we only report edges in the list of adjacencies for `vertex` where `vertex` is the source of the edge.

```
@checkdoc
class EdgesViewList(Iterable):
    """A view of all the edges of a GraphList.
    Returned by the edges method of a GraphList."""
    __slots__ = ["_target"]

    def __init__(self, target):
        self._target = target

    def __iter__(self):
        for vertex in self._target._vertex_dict.values():
            for edge in vertex._adjacencies:
                if self._target.directed or (edge.here is vertex.label):
                    yield edge

    def __len__(self):
        return self._target.edge_count
```

With two implementations of graphs in mind, we now focus on a number of examples of their use.

15.4 Examples: Common Graph Algorithms

Because the graph structure is so flexible there are many good examples of graph applications. In this section, we investigate a number of beautiful algorithms involving graphs. These algorithms provide a cursory overview of the problems that may be cast as graph problems, as well as techniques that are commonly used to solve them.

15.4.1 Reachability

Once data are stored within a graph, it is often desirable to identify vertices that are reachable from a common source (see Figure 15.6). One approach is to treat the graph as you would a maze and, using search techniques, find the reachable vertices. For example, we may use *depth-first search*: each time we visit an unvisited vertex we seek to further deepen the traversal. The following code demonstrates how we might use recursion to search for unvisited vertices:

Reachability

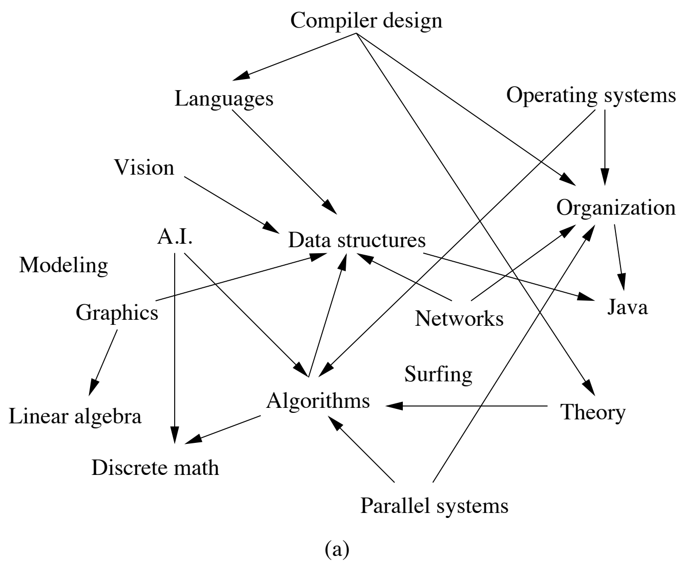
```
def reachable_from(graph, vertex, clear=True):
    """Return a list of all vertices in graph reachable from vertex."""
    if clear:
        graph.reset()
    graph.visit(vertex)
    for n in graph.neighbors(vertex):
        if not graph.visited(n):
            reachable_from(graph, n, clear=False)
```

When `clear` is `True`, we clear each `Vertex`'s visited flag with a call to `reset`, and then call `reachable_from` with the graph and the source vertex for the reachability test. Before the call to `reachable_from`, `vertex` has not been visited. After the call, every vertex reachable from the vertex has been visited. Some vertices may be left unvisited and are not reachable from the source. So, to determine whether you may reach one vertex from another, the following code can be used:

```
reachable_from(g,u)
if g.visited(v):
    ...
```

In Section **??** we discussed the use of a `Linear` structure to maintain the state of a search of a maze. The use of a `Stack` led to a depth-first search. This implementation might be more resilient, especially with large graphs, if a `Stack` had been used.

How long does it take to execute the procedure? Suppose that, ultimately, we visit the reachable vertices V_r. Let E_r be the edges of the graph found among the vertices of V_r. Clearly, each vertex of V_r is visited, so there is one call to `reachable_from` from each vertex $v \in V_r$. For each call, we ask each destination vertex if it has been visited or not. There is one such test for every edge within E_r. Thus, the total time is $O(|V_r|+|E_r|)$. Since $|E_r| \geq |V_r - 1|$ (every



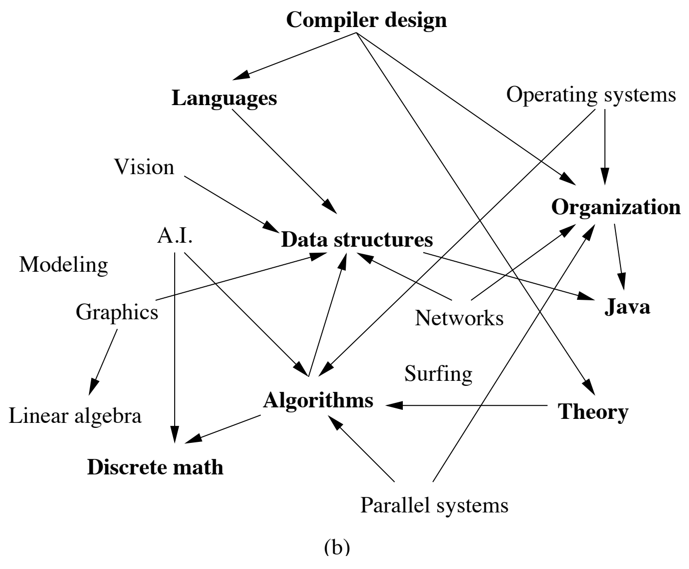


Figure 15.6 Courses you might be expected to have taken if you're in a compiler design class. (a) A typical prerequisite graph (classes point to prerequisites). Note the central nature of data structures! (b) Bold courses can be reached as requisite courses for compiler design.

new vertex is visited by traversing a new edge), the algorithm is dominated by the number of edges actually investigated. Of course, if the graph is dense, this is bounded above by the square of the number of vertices.

In an undirected graph the reachable vertices form a component of the graph. To count the components of a graph (the undirected version of the graph of Figure 15.6 has three components), we iterate across the vertices of the graph, calling the `reachable_from` procedure on any vertex that has not yet been visited. Since each unvisited vertex is not reachable from those that have been encountered before, the number of searches determines the number of components.

15.4.2 Topological Sorting

Occasionally it is useful to list the vertices of a graph in such a way as to make the edges point in one direction, for example, toward the front of the list. Such graphs have to be directed and acyclic (see Problem **??**). A listing of vertices with this property is called a *topological sort*.

One technique for developing a topological sort involves keeping track of a counter or virtual timer. The timer is incremented every time it is read. We now visit each of the nodes using a depth-first search, labeling each node with two *time stamps*. These time stamps determine the span of time that the algorithm spends processing the descendants of a node. When a node is first encountered during the search, we record the *start time*. When the recursive depth-first search returns from processing a node, the timer is again read and the *finish time* is recorded. Figure 15.7 depicts the intervals associated with each vertex of the graph of Figure 15.6. (As arbitrary convention, we assume that a vertex iterator would encounter nodes in the diagram in "reading" order.)

One need only observe that the finish time of a node is greater than the finish time of any node it can reach. (This depth-first search may have to be started at several nodes if there are several independent components, or if the graph is not strongly connected.) The algorithm, then, simply lists the vertices in the order in which they are finished. For our course graph we generate one of many course schedules that allow students to take courses without violating course requirements:

Vertices Ordered by Finish Time					
1-2.	Discrete Math	12-15.	Theory	23-26.	Graphics
4-5.	Python	7-16.	Compiler Design	27-28.	Modeling
3-6.	Organization	17-18.	Vision	29-30.	Surfing
9-10.	Data Structures	19-20.	Networks	31-32.	A.I.
8-11.	Languages	21-22.	Parallel systems	33-34.	Operating Systems
13-14.	Algorithms	24-25.	Linear algebra		

Actually, the time stamps are useful only for purposes of illustration. In fact, we can simply append vertices to the end of a list at the time that they would

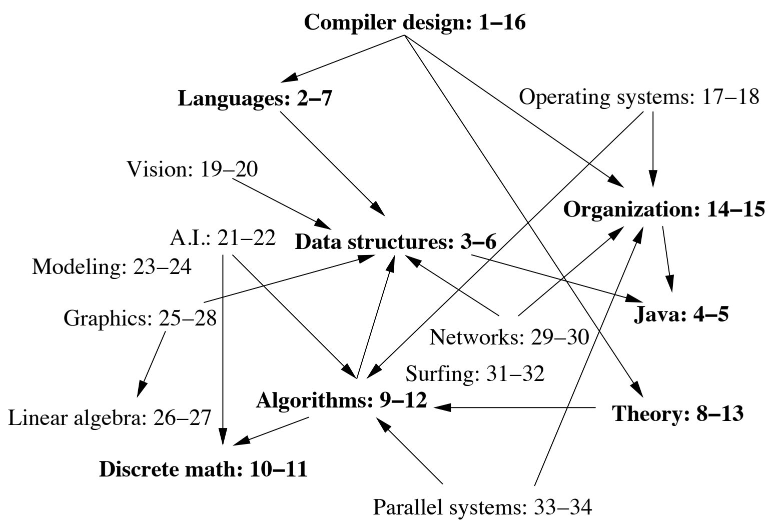


Figure 15.7 The progress of a topological sort of the course graph. The time interval following a node label indicates the time interval spent processing that node or its descendants. Dark nodes reachable from compiler design are all processed during the interval [1–16]—the interval associated with compiler design.

normally be finished. Here is a sample code:

```
def topo_sort(graph):
    results = []
    graph.reset()
    for v in graph:
        if not graph.visited(v):
            DFS(graph, v, results)
    return results

def DFS(graph, vertex, visited_list):
    graph.visit(vertex)
    for n in graph.neighbors(vertex):
        if not graph.visited(n):
            DFS(graph, n, visited_list)
    visited_list.append(vertex)
```

These functions are declared as global procedures of a program that might make use of a topological sort. Alternatively, they could be written as methods of a graph, reducing the complexity of method calls.

15.4.3 Transitive Closure

Previously we discussed a reachability algorithm that determines if it is possible to reach any particular vertex from a particular source. It is also useful to compute the *transitive closure* of a graph: for *each pair* of vertices $u, v \in V$, *is v reachable from u?* These questions can be answered by $O(|V|)$ calls to the depth-first search algorithm (leading to an algorithm that is $O(|V|(|V| + |E|))$), or we can look for a more direct algorithm that has similar behavior.

One algorithm, *Warshall's algorithm*, computes reachability for each pair of vertices by modifying the graph. When the algorithm is applied to a graph, edges are added until there is an edge for every pair of connected vertices (u, v). The concept behind Warshall's algorithm is relatively simple. Two connected vertices u and v are either directly connected, or the path from u to v passes through an intermediate node w. The algorithm simply considers each node and connects all pairs of nodes u and v that can be shown to use w as an intermediate node. Here is a Java implementation:

Warshall

```
def warshall(graph):
    for w in graph:
        for u in graph:
            for v in graph:
                if graph.contains_edge(u, w) and graph.contains_edge(w, v):
                    graph.add_edge(u, v)
    return graph
```

This algorithm is clearly $O(|V|^3)$: each iterator visits $|V|$ vertices and (for adjacency matrices) the check for existence of an edge can be performed in constant time.

To see how the algorithm works, we number the vertices in the order they are encountered by the vertex iterator. After k iterations of the outer loop, all "reachability edges" of the subgraph containing just the first k vertices are completely determined. The next iteration extends this result to a subgraph of $k + 1$ vertices. An inductive approach to proving this algorithm correct (which we avoid) certainly has merit.

15.4.4 All Pairs Minimum Distance

A slight modification of Warshall's algorithm gives us a method for computing the minimum distance between all pairs of points. The method is due to Floyd. Again, we use three loops to compute the new edges representing reachability, but these edges are now labeled, or *weighted*, with integer distances that indicate the current minimum distance between each pair of nodes. As we consider intermediate nodes, we merge minimum distance approximations by computing and updating the distance if the sum of path lengths through an intermediate node w is less than our previous approximation. Object orientation makes this code somewhat cumbersome:

Floyd

```
def floyd(graph):
    for w in graph:
        for u in graph:
            for v in graph:
                if graph.contains_edge(u, w) and graph.contains_edge(w, v):
                    leg1 = graph.get_edge(u,w)
                    leg2 = graph.get_edge(w, v)
                    new_dist = leg1.label + leg2.label
                    if graph.contains_edge(u, v):
                        across = graph.get_edge(u, v)
                        across_dist = across.label
                        if new_dist < across_dist:
                            across.label = new_dist
                    else:
                        graph.add_edge(u, v, new_dist)
    return graph
```

Clearly, edge labels could contain more information than just the path length. For example, the path itself could be constructed, stored, and produced on request, if necessary. Again, the complexity of the algorithm is $O(|V|^3)$. This is satisfactory for dense graphs, especially if they're stored in adjacency matrices, but for sparse graphs the checking of all possible edges seems excessive. Indeed, other approaches can improve these bounds. We leave some of these for your next course in algorithms!

15.4.5 Greedy Algorithms

We now consider two examples of *greedy* algorithms—algorithms that compute optimal solutions to problems by acting in the optimal or "most greedy" manner at each stage in the algorithm. Because both algorithms seek to find the best choice for the next step in the solution process, both make use of a priority queue.

Minimum Spanning Tree

The solution to many network problems involves identifying a *minimum spanning tree* of a graph. A minimum spanning tree of an edge-weighted graph is a tree that connects every vertex of a component whose edges have minimum total edge weight. Such a tree might represent the most inexpensive way to connect several cities with telephone trunk lines. For this reason, we will interpret the weights as edge lengths. For the purposes of our discussion, we will assume that the graph under consideration is composed of a single component. (If the graph contains multiple components, we can compute a minimum spanning forest with multiple applications of the minimum spanning tree algorithm.)

It is useful to note that if the vertices of a connected graph are partitioned into any two sets, any minimum spanning tree contains a shortest edge that connects nodes between the two sets. We will make use of this fact by segregating visited nodes from unvisited nodes. The tree (which spans the visited nodes and does so minimally) is grown by iteratively incorporating a shortest edge that incorporates an unvisited node to the tree. The process stops when $|V|-1$ edges have been added to the tree.

```
def mcst(graph):
    q = SkewHeap(key=label_order)
    graph.reset()
    if len(graph) == 0:
        return
    vi = iter(graph)
    v = next(vi)
    searching = False
    while not searching:
        graph.visit(v)
        # add all outgoing edges to priority queue
        for v2 in graph.neighbors(v):
            e = graph.get_edge(v,v2)
            q.add(e)
        searching = True
        while searching and not q.empty:
            e = q.remove()
            v = e.here
            if graph.visited(v):
                v = e.there
```

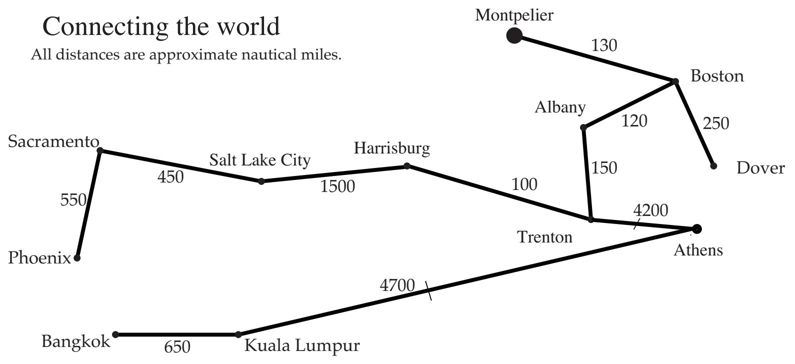


Figure 15.8 The progress of a minimum spanning tree computation. Bold vertices and edges are part of the tree. Harrisburg and Trenton, made adjacent by the graph's shortest edge, are visited first. At each stage, a shortest external edge adjacent to the tree is incorporated.

```
if not graph.visited(v):
    searching = False
    graph.visit_edge(e)
```

First, we use a priority queue to rank edges based on length. As we remove the edges from the queue, the smallest edges are considered first (see Figure 15.8). When an edge is considered that includes an unvisited vertex, we visit it, logically adding it to the minimum spanning tree. We then add any edges that are outward-bound from the newly visited node. At any time, the priority queue contains only edges that mention at least one node of the tree. If, of course, an edge is considered that mentions two previously visited nodes, the edge is superfluous, as the nodes are already connected by a path in the tree (albeit a potentially long one). When the priority queue "runs dry," the tree is fully computed. The result of the algorithm will be visited marks on all nodes and edges that participate in the tree. (If the graph has multiple components, some vertices will not have been visited.)

Our implementation begins by finding a source vertex (v) to "prime" the greedy algorithm. The main loop of the algorithm then runs until no new vertices can be added to the tree. Each new vertex is marked as visited and its outbound edges[2] are then added to the priority queue (q) of those to be con-

[2] We use a key function that targets the label of the edge here, which would be typical for this

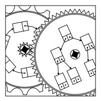
sidered. Short edges are removed from the queue until an unvisited vertex is mentioned by a new tree edge (e), or the queue is emptied.

Over the course of the algorithm, consideration of each edge and vertex results in a priority queue operation. The running time, then is $O((|V| + |E|)\log(|V|))$.

Notice that the first edge added may not be the graph's shortest.

Single-Source Shortest Paths

The minimum spanning tree algorithm is related to a fast, single-source, shortest-path algorithm attributed to Dijkstra. In this algorithm, we desire the minimum-length paths from a single source to all other nodes. We expect the algorithm, of course, to run considerably faster than the all-pairs version. This algorithm also runs in time proportional to $O((|V| + |E|)\log(|V|))$ due to the fact that it uses much the same control as the minimum spanning tree. Here is the code:



Dijkstra

```
def dijsktra(g,start):
    """Given graph g (edges labeled (distance name)) determine shortest path
    to each reachable vertex from start."""

    # priority queue for ranking closest unvisited next city
    q = SkewHeap(key=highwaydistance)
    # table of results: key is city, value is
    # (distance-from-start prior-city new-city highway)
    result = Table()
    v = start
    possible = (0,start,start,"<start>")

    while v:
        if v not in result:
            # a new city is considered; record result
            result[v] = possible
            vdist = possible[0]
            # add all outgoing roads to priority queue
            for w in g.neighbors(v):
                (distance, highway) = g.get_edge(v,w).label
                q.add((distance+vdist,v,w,highway))
        # find next closest city to start
        if len(q) == 0: break
        possible = q.remove()
        (distance, somewhere, v, highwayname) = possible
        # break out of this loop
    return result
```

algorithm. Another possibility would be to extend the Edge class to totally order edges based on their label.

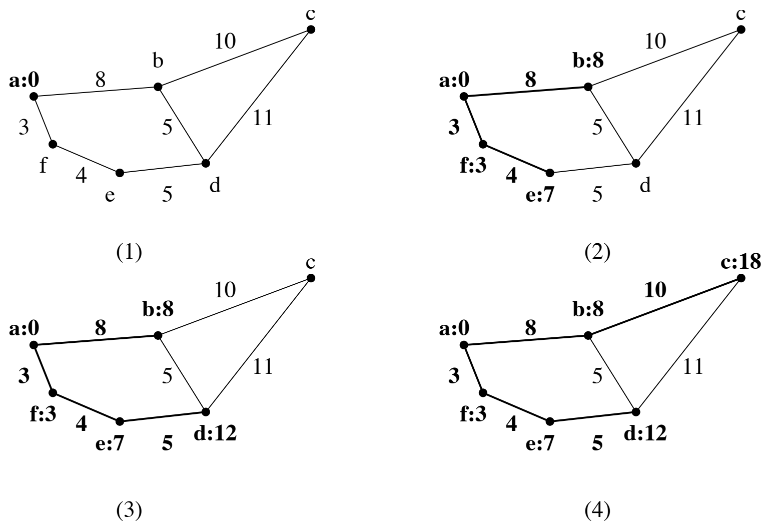


Figure 15.9 The progress of a single-source, shortest-path computation from source **a**. As nodes are incorporated, a minimum distance is associated with the vertex. Compare with Figure 15.8.

Unlike the minimum cost spanning tree algorithm, we return a `Table` of results. Each entry in the `Table` has a vertex label as a key. The value is an association between the total distance from the source to the vertex, and (in the nontrivial case) a reference to the last edge that supports the minimum-length path.

We initially record trivial results for the source vertex (setting its distance to zero) and place every outgoing edge in the priority queue (see Figure 15.9). Unlike the minimum spanning tree algorithm, we rank the edges based on *total* distance from the source. These edges describe how to extend, in a nearest-first, greedy manner, the paths that pass from the source through visited nodes. If, of course, an edge is dequeued that takes us to a vertex with previously recorded results, it may be ignored: some other path from the source to the vertex is shorter. If the vertex has not been visited, it is placed in the `Table` with the distance from the source (as associated with the removed edge). New outbound edges are then enqueued.

The tricky part is to rank the edges by the distance of the destination vertex from the source. We can think of the algorithm as considering edges that fall within a neighborhood of increasing radius from the source vertex. When the boundary of the neighborhood includes a new vertex, its minimum distance from the source has been determined.

Since every vertex is considered once and each edge possibly twice, the worst-case performance is $O(|V| + |E|)$, an improvement over the $O(|V|^3)$ performance for sparse graphs.

15.5 Conclusions

In this chapter we have investigated two traditional implementations of graphs. The adjacency matrix stores information about each edge in a square matrix while the adjacency list implementation keeps track of edges that leave each vertex. The matrix implementation is ideal for dense graphs, where the number of actual edges is high, while the list implementation is best for representing sparse graphs.

Our approach to implementing graph structures is to use partial implementations, called abstract classes, and extend them until they are concrete, or complete. Other methods are commonly used, but this has the merit that common code can be shared among similar classes. Indeed, this inheritance is one of the features commonly found in object-oriented languages.

This last section is, in effect, a stepping stone to an investigation of algorithms. There are many approaches to answering graph-related questions, and because of the dramatic differences in complexities in different implementations, the solutions are often affected by the underlying graph structure.

Finally, we note that many of the seemingly simple graph-related problems cannot be efficiently solved with *any* reasonable representation of graphs. Those problems are, themselves, a suitable topic for many future courses of study.

Appendix A

Answers

This section contains answers to many problems presented to the reader in the text. In the first section, we provide answers to self check problems. In the second section, we provide answers to many of the problems from the text.

A.1 Solutions to Self Check Problems

Chapter 4

Problems begin on page 92.
4.1 Self-check answer.

A.2 Solutions to Select Problems

Chapter 4

Problems begin on page 92.
4.1 Regular answer.