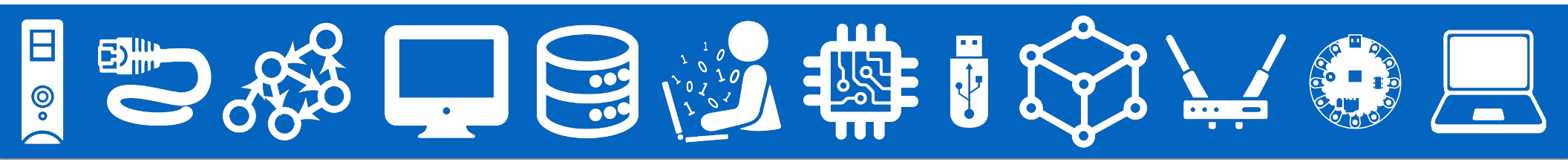
CS134:
Sorting & Dictionaries



Announcements & Logistics

- **No homework** this week!
- **Practice midterm** will be released on Glow under Files
  - Two versions: with and without solutions
  - Midterm from F18 with slight modifications to fit our syllabus
- **Lab 5** will be a short debugging lab
  - Expect most people to finish it during scheduled lab period
- **Midterm**: Thur Oct 20, 6-7:30pm or 8-9:30pm
- **Midterm review**: Tue Oct 18, 8-9:30pm
- **No class** Fri Oct 21 regardless of Mountain Day!

Do You Have Any Questions?

Last Time

- Discussed new ***immutable*** sequences: **tuples**
  - All sequence operations apply to tuples
  - Useful for multi-item assignment (argument *unpacking*)
  - Appropriate when passing collections of data around that should ***not*** be mutated (and you want to avoid aliasing issues)
- Learned about **sorting** and default sorting behavior
- Discussed how we can override the default sorting behavior
  - By using `reverse=True`

Today's Plan

- Continue discussing sorting in Python
  - Explore ways to override default behavior using `key` function
  - Discuss **stable** sorting
- Discuss a new data structure: **dictionary**
  - "**Unordered**" and **mutable** collection
  - Ordered/sequential data structures (like lists, tuples, strings) aren't appropriate for all use cases
  - For many applications, unordered collections are more efficient

Sorting with a `key` function

- Now suppose we have a list of tuples that we want to sort by something *other* than the first item
- Example: We have a list of course tuples, where the first item is the course name, second item is the enrollment capacity, and third item is the term (Fall/Spring).

```
courses = [('CS134',   90, 'Spring'), ('CS136',   60, 'Spring'),
           ('AFR206',  30, 'Spring'), ('ECON233', 30, 'Fall'),
           ('MUS112',  10, 'Fall'),   ('STAT200', 50, 'Spring'),
           ('PSYC201', 50, 'Fall'),   ('MATH110', 90, 'Spring')]
```

- Suppose we want to sort these courses by their **capacity** (second element)
- We can accomplish this by supplying the `sorted()` function with a `key` function that tells it how to compare the tuples to each other

Sorting with a `key` function

- **Defining a key function explicitly:**
  - We can define an explicit `key` function that, when given a tuple, returns the parameter we want to sort the tuples with respect to

```
def capacity(courseTuple):
    '''Takes a sequence and returns item at index 1'''
    return courseTuple[1]
```

  - Once we have defined this function, we can pass it as a `key` when calling `sorted()`

```
# we can tell sorted() to sort by capacity instead
sorted(courses, key=capacity)
```

Sorting with a **key** function

- `sorted(seq, key=function)`
  - Interpret as `for el in seq`: use `function(el)` to sort `seq`
  - For **each element in the sequence**, `sorted()` **calls the key function on the element** to figure out what "feature" of the data should be used for sorting

```
# we can tell sorted() to sort by capacity instead
sorted(courses, key=capacity)
```

  - For each `course` in `courses` (a list of tuples), sort based on value returned by `capacity(course)`

Sorting with a **key** function

```
courses = [('CS134',   90, 'Spring'), ('CS136',   60, 'Spring'),
           ('AFR206',  30, 'Spring'), ('ECON233', 30, 'Fall'),
           ('MUS112',  10, 'Fall'),   ('STAT200', 50, 'Spring'),
           ('PSYC201', 50, 'Fall'),   ('MATH110', 90, 'Spring')]
```

```
def capacity(courseTuple):
    '''Takes a sequence and returns item at index 1'''
    return courseTuple[1]
```

```
# we can tell sorted() to sort by capacity instead
sorted(courses, key=capacity)
```

```
[('MUS112',  10, 'Fall'),
 ('AFR206',  30, 'Spring'),
 ('ECON233', 30, 'Fall'),
 ('STAT200', 50, 'Spring'),
 ('PSYC201', 50, 'Fall'),
 ('CS136',   60, 'Spring'),
 ('CS134',   90, 'Spring'),
 ('MATH110', 90, 'Spring')]
```

Python Sorting is Stable

- Python's sorting functions are **stable**
  - Items that are "equal" according to the sorting **key** have the same relative order as in the original (unsorted) sequence

```
courses = [('CS134',   90, 'Spring'), ('CS136',   60, 'Spring'),
           ('AFR206',  30, 'Spring'), ('ECON233', 30, 'Fall'),
           ('MUS112',  10, 'Fall'),   ('STAT200', 50, 'Spring'),
           ('PSYC201', 50, 'Fall'),   ('MATH110', 90, 'Spring')]
```

```
def term(courseTuple):
    '''Takes a sequence and returns item at index 2'''
    return courseTuple[2]
```

```
# sort courses by term
# notice the impact of stable sorting wrt to ties
sorted(courses, key=term)
```

```
[('ECON233', 30, 'Fall'),
 ('MUS112',  10, 'Fall'),
 ('PSYC201', 50, 'Fall'),
 ('CS134',   90, 'Spring'),
 ('CS136',   60, 'Spring'),
 ('AFR206',  30, 'Spring'),
 ('STAT200', 50, 'Spring'),
 ('MATH110', 90, 'Spring')]
```

Here we are sorting by term. Notice the ordering of courses with Fall term and those with Spring term (same as original list)

Breaking Ties using **key**

- We can override this default behavior and specify how to break ties by supplying a `key` function that returns a **tuple**

```
# if you want to handle ties, can return a tuple in key function
def termAndCap(courseTuple):
    '''Takes a sequence and returns item at index 2'''
    return courseTuple[2], courseTuple[1]
```

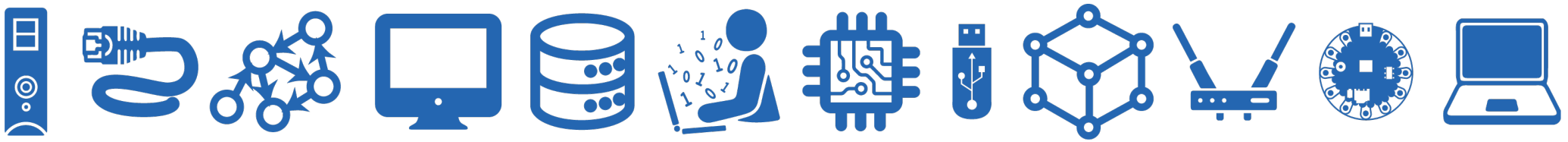
```
sorted(courses, key=termAndCap)
```

```
[('MUS112',  10, 'Fall'),
 ('ECON233', 30, 'Fall'),
 ('PSYC201', 50, 'Fall'),
 ('AFR206',  30, 'Spring'),
 ('STAT200', 50, 'Spring'),
 ('CS136',   60, 'Spring'),
 ('CS134',   90, 'Spring'),
 ('MATH110', 90, 'Spring')]
```

Notice that now the ties are broken in favor of capacity

Examples:

Sorting with a key Function



Other uses for `key`

- What if we want to override the default sorting behavior for integers so that they sort based on **absolute values** (or magnitude)?
- That is,
  - For an input `[-50, 50, -29, 27, 8]`
  - The sorted output should be `[8, 27, -29, -50, 50]`

- Can we also define some sensible sorting behavior on mixed lists e.g., `['a', 42, 'b', 100]`? By default, `sorted()` will throw an error on such lists.

- Ex: Jupyter notebook

Sorting on Magnitude

```
def absoluteValue(num):
  '''Takes a number and returns its absolute value'''
   if num < 0:
     return -1 * num
   else:
     return num
```

```
>>> numbers = [-50, 50, -29, 27, 8]
>>> print("Default sorting behavior", sorted(numbers))
Default sorting behavior [-50, -29, 8, 27, 50]
>>> print("Sorting on magnitude", sorted(numbers, key=absoluteValue))
Sorting on magnitude [8, 27, -29, -50, 50]
```

Sorting Mixed Lists

- We can use the ASCII values of characters to make sensible comparisons of letters to numbers. However, custom sorting behaviors are really only limited by your imagination!

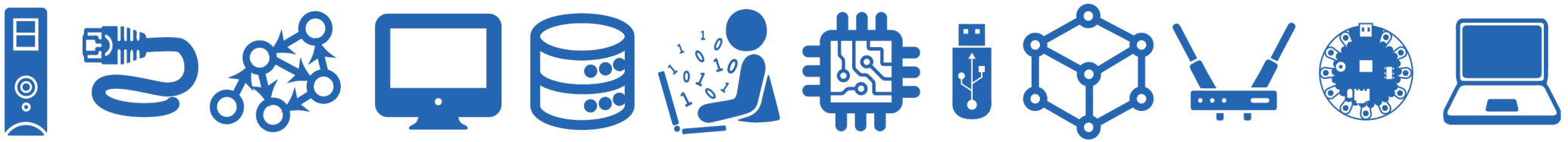
```
def returnOrdValue(element):
   ''' Returns the ASCII value for an element if it is a character,
   otherwise assumes that the given element is a number and returns
   the number itself '''
   if type(element) == str:
     return ord(element)
   else:
     return element
```

```
>>> mixedList = ['a', 'b', 24, 50, 125]
>>> print("Sorting mixed list ", sorted(mixedList, key=returnOrdValue))
Sorting mixed list [24, 50, 'a', 'b', 125]
```

Sorting Takeaways

- `sorted()` function and `.sort()` list method, by default, sort sequences in ascending and lexicographic order
    - `sorted()` function works for any **sequence**, always returns a new sorted **list**
    - `.sort()` method **sorts lists in place**, uses dot notation for invocation (only works on lists!)
- We can override Python's default sorting behavior by supplying optional parameters `key` (function), and `reverse` (Boolean)
- Note: `.sort()` method for lists also supports `key` and `reverse` parameters just like `sorted()`

Dictionaries



Sequences vs Unordered Collections

- **Sequence:** a group of items that come one after the other (there is an implicit ***ordering*** of items)
    - Sequences in Python: strings, lists, tuples, ranges
- **Unordered Collection:** a group of things bundled together for a reason but without a specific ordering
    - Maintaining order between items is not always necessary
    - Ordering items comes at a cost in terms of efficiency!
- For some use cases, it is better to store an unordered collection
- Python has two data structures which are **unordered**:
    - **Dictionaries** and **sets:** both of them are **mutable**
    - We will discuss **dictionaries** today

Dictionaries

- A **dictionary** is a **mutable** collection that maps **keys** to **values**
  - Enclosed with curly brackets, and contains **comma-separated** items
  - Each item in the dictionary is a **colon-separated** ***key, value pair***
  - There is no ordering between the keys of a dictionary!

```
# sample dictionary
zipCodes = {'01267': 'Williamstown', '60606': 'Chicago',
            '48202': 'Detroit', '97210': 'Portland'}
```

key value key value

- **Keys** must be an **immutable** type such as ints, strings, or tuples
  - Keys of a dictionary must also be **unique**: no duplicates allowed!
- **Values** can be any Python type (ints, strings, lists, tuples, etc.)

Accessing Items in a Dictionary

- Dictionaries are **unordered** so we cannot index into them: no notion of first or second item, etc.
- We access a dictionary using its `keys` as the subscript in `[]` notation
  - If the key exists, its corresponding value is returned
  - If the key does not exist, it leads to a `KeyError`

```
>>> # sample dictionary
>>> zipCodes = {"01267": "Williamstown", "60606": "Chicago",
                "48202": Detroit, "97210": "Portland"}

>>> # what US city has this zip code?
>>> zipCodes["60606"]
'Chicago'

>>> # what US city has this zip code?
>>> zipCodes["48202"]
'Detroit'
```

key

value

value associated with key '60606'

value associated with key '48202'

Adding a Key, Value Pair

- Dictionaries are **mutable**, so we can add items or remove items from it
- To add a new **key, value** pair, we can simply assign the key to the value using: `dictName[key] = value`

```
>>> zipCodes["11777"] = "Port Jefferson"
>>> zipCodes
{'01267': 'Williamstown',
 '60606': 'Chicago',
 '48202': 'Detroit',
 '97210': 'Portland',
 '11777': 'Port Jefferson'}
```

Add key, value pair `'11777': 'Port Jefferson'`

- If the key already exists, an assignment operation as above will ***overwrite*** its value and assign it the new value

Operations on Dictionaries

- Just like sequences, we can use the `len()` function on dictionaries to find out the **number of keys** it contains
- To check if a **key** exists or does not exist in a dictionary, we can use the `in` or `not in` operator, respectively

```
>>> zipCodes
{'01267': 'Williamstown',
 '60606': 'Chicago',
 '48202': 'Detroit',
 '97210': 'Portland',
 '11777': 'Port Jefferson'}
>>> len(zipCodes)
5
```

```
>>> "90210" in zipCodes
False
>>> "01267" in zipCodes
True
>>> "Chicago" in zipCodes
False
```

Should always check if a key exists before accessing its value in a dictionary

Creating Dictionaries

- Several ways to create dictionaries:
  - **Direct assignment**: provide key, value pairs delimited with `{ }`
  - Start with empty dict and add key, value pairs
    - Empty dict is `{}` or `dict()`
  - Apply the built-in function `dict()` to a list of tuples

```
# direct assignment
scrabbleScore = {'a':1 , 'b':3, 'c':3, 'd':2, 'e':1,
                 'f':4, 'g':2, 'h':4, 'i':1, 'j':8,
                 'k':5, 'l':1, 'm':3, 'n':1, 'o':1,
                 'p':3, 'q':10, 'r':1, 's':1, 't':1,
                 'u':1, 'v':8, 'w':4, 'x':8, 'y':4, 'z': 10}
```

Note: keys may be listed in any order, since dictionaries are unordered

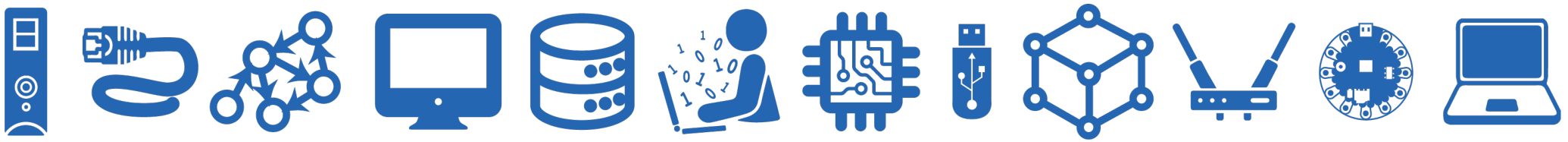
Creating Dictionaries

- Direct assignment: provide key, value pairs delimited with { }
- Start with empty dict and add key, value pairs
  - Empty dict is `{}` or `dict()`
- Apply the built-in function `dict()` to a list of tuples

```
# accumulate in a dictionary
verse = "let it be,let it be,let it be,let it be,there will be an answer,let it be"
counts = {} # empty dictionary
for line in verse.split(','):
    if line not in counts:
        counts[line] = 1 # initialize count
    else:
        counts[line] += 1 # update count
```

```
>>> counts
{'let it be': 5, 'there will be an answer': 1}
>>> # use dict() function
>>> dict([('a', 5), ('b', 7), ('c', 10)])
{'a': 5, 'b': 7, 'c': 10}
```

Example:
Frequency



Example: `frequency`

- Let's write a function `frequency()` that takes as input a list of strings `wordList` and returns a dictionary `freqDict` with the unique strings in `wordList` as keys, and their number of occurrences (ints) in `wordList` as values
- For example if `wordList` is

```
['hello', 'world', 'hello', 'earth', 'hello', 'earth']
```

the function should return a dictionary with the following items

```
{'hello': 3, 'world': 1, 'earth': 2}
```

Example: `frequency`

- Let's write a function `frequency()` that takes as input a list of strings `wordList` and returns a dictionary `freqDict` with the unique strings in `wordList` as keys, and their number of occurrences (ints) in `wordList` as values

```
def frequency(wordList):
    """Given a list of words, returns a dictionary of word frequencies"""
    freqDict = {} # initialize accumulator as empty dict
    for word in wordList:
        if word not in freqDict:
            freqDict[word] = 1 # add key with count 1
        else:
            freqDict[word] += 1 # update count
    return freqDict
```

- More on this next time!