CSCI 134 Fall 2021: Classes and Objects

Nov 03, 2021

Shikha Singh, 9AM
Jeannie Albrecht, 10AM
Announcements & Logistics

- **Lab 7** due Wed/Thurs, 10 pm
  - Remember to **add, commit and push** code and **required images**
  - Any questions?
- **Lab 6** graded feedback: coming soon
- **HW 7** released today, due Mon 10 pm (fewer questions this week)
- **Lab 8 (Ciphers)** is going to be a **partner lab**
  - Fill out partner google form (from Lida) by **tomorrow @ 10 pm**
  - **Both partners have to fill out the form!**
  - Can work by yourself but **strongly encourage** you to find a partner
  - **Must attend one lab session together**
  - Mon lab due on Wed, Tue lab due on Thur

_Do You Have Any Questions?_
Class of 60s Public Talk Tomorrow

- Rare opportunity to hear a public talk from an expert in the field
- Streamed in Wege, get colloquium credit for attending

Computer Science Class of ’60s Speaker – Jon Kleinberg, Cornell

Thu, November 4th, 2021
7:00 pm - 8:30 pm

Designing Algorithms for High-Stakes Decisions

Algorithms are increasingly used to aid decision-making in high-stakes settings including employment, lending, healthcare, and the legal system. These developments have led to an ecosystem of growing complexity in which algorithms and people interact around consequential decisions. We consider some of the societal challenges that have emerged from this process, focusing in particular on the ways in which sources of bias and discrimination can affect algorithmic decision-making, and how we can work to understand and detect these effects. (This talk will include joint work with Jens Ludwig, Sendhil Mullainathan, Manish Raghavan, and Cass Sunstein.)
Last Time

• Recorded lecture on graphical recursion; three examples
  • Spirals (both fruitful and non-fruitful versions)
  • Trees (very helpful for Task 3 Shrub in Lab 7)
  • Sierpinski Triangle (example of three recursive calls)
Today

• Starting on a new topic: **classes** and **objects**
• Python is an **object oriented programming** (OOP) language
• Everything in Python is an **object** and has a **type**
• Learn how to define our own **classes** (**types**) and **methods**
  • Today’s topics: `__init__` special method, self parameter
Familiar Objects in Python

- Python supports many different kinds of data:
  
  1234    3.14159    "Hello"    [1, 5, 7, 11, 13]
  {"CA": "California", "MA": "Massachusetts"}

- Each of these is an **object**, and every object has:
  
  - a **type**
  - an internal **data representation** (primitive or composite)
  - a set of functions for **interaction** with the object

- A specific object is an **instance** of a type:
  
  - 1234 is an instance of an **int**
  - "hello" is an instance of a **string**
The outputs of `type` queries appear differently on Jupyter notebooks and interactive Python.

- Interactive Python shows the word `class`, Jupyter does not.

```python
In [1]: type(1234)
Out[1]: int

In [2]: type("hello")
Out[2]: str

In [3]: type([1, 5, 7, 11, 13])
Out[3]: list

In [4]: type(range(5))
Out[4]: range
```
Objects and Types in Python

EVERYTHING IN PYTHON IS AN OBJECT
(AND HAS A TYPE)

• Even functions are a type!

• Guido designed the language according to the principle “first-class everything”

“One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth." — Guido Van Rossum

(Blog, The History of Python, February 27, 2009)
Stepping Back: Object-Oriented Programming (OOP)

• Python is an “object-oriented” language
  • We have been only hinting at this aspect so far
  • Today we will embrace it!

• OOP is a fundamental programming paradigm

• It has four major principles:
  • Abstraction (data and procedural)
  • Inheritance
  • Encapsulation
  • Polymorphism

• We’ll explore some of these principles in more detail in the coming lectures
What are Objects?

• It’s time to formally define **objects** in Python

• Objects are:
  
  • collections of data (variables and **attributes**) and
  
  • **methods** (functions) that act on those data

• Example of abstraction:
  
  • Methods define behavior but hide implementation and internal representation of data
    
    • Eg., You have been using built-in Python data types (lists, strings, etc) all semester without really knowing how they are implemented
Example: \([1,2,3,4]\) has type list

- Lists are represented internally by a sequence of “cells” connected via pointers (formally called a singly linked list).

\[
\text{L} = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4
\]

- This representation is private and the user does not need to know it to use list objects (you’ve been doing it all semester!)

- How do we manipulate lists? Using methods defined in interface.
  - L[i], L[i: j], +, len(), etc.
  - L.append(), L.extend(), etc.

**Summary:** Internal representation should be private. Objects are manipulated through associated methods.
Creating Our Own Types: Classes

• We can create our own type by defining our own class.
  • Classes are like blueprints for objects.

• Creating a class involves:
  • Defining the class name, attributes, methods.
    • E.g., someone wrote the code to implement a list class.

• Using the class involves:
  • Creating new instances of the class (objects).
    • E.g., L = [1, 2], L = list("abc")
  • Performing operations on the instances through methods.
    • E.g., L.append(3)
Defining Our Own Type: Car class

Class provides a “blueprint” for creating specific cars.

Specific instances of the Car class:
- Red
- Ford
- Mustang
- Blue
- Toyota
- Prius
- Green
- Volkswagen
- Golf

Attributes of the Car class, such as color, make, and model, define key features of the objects.
Defining Our Own Type: Book class

Name of class: Capitalized by convention

class Book:

    '''This class represents a book'''
    # attributes go here
    # indented body of class definition

Creating instances of the class:

b1 = Book()  # b1 is an instance of class Book
b2 = Book()  # b2 is another (different) instance of class Book
Data Attributes or Instance Variables

- Objects have **state** which is typically held in **instance variables** or (in very Pythonic terms:) **attributes**.
  - Extra bits of information that are relevant to the class
- Example: The **Book** class may define **attributes** like the book’s name and author
  - Every Book instance has different attribute values
  - Useful to make these attributes accessible to all methods in the class
- We could assign these attributes directly to an instance of the class but **we should never do this:**
  ```python
  b1 = Book()
  b1.name = "Emma"
  b1.author = "Jane Austen"
  ```

**Improper** approach to assign attribute values. We’ll come back to this in a bit.
Defining Class Methods
Methods or Procedural Attributes

• Think of methods as **object-specific functions**

• They are defined as part of the class definition and describe how to interact with the class objects

• Example: Recall the following methods for the list class

```
In [1]: L = list()

In [2]: L.extend([1,2,3])

In [3]: L
Out[3]: [1, 2, 3]

In [4]: L.append(4)

In [5]: L
Out[5]: [1, 2, 3, 4]
```

`dot operator to “call” the method on the object`
Methods for Objects of Class

• In our list example, we called methods like `append()` and `extend()` on a particular list object `L`.

• We can define methods in our classes in a similar way.

• Consider this simple example:

```python
In [6]: class A:
   """Class to test the use of methods""
   def greeting(self):
       print("Hello")
```
Methods for Objects of Class

• In our list example, we called methods like `append()` and `extend()` on a particular list object `L`.

• We can define methods in our classes in a similar way.

• Consider this simple example:

```python
In [6]:
class A:
    """Class to test the use of methods""
    def greeting(self):
        print("Hello")
```

What is this?
Methods for Objects of Class

• To create methods that can be called on an instance of a class, they must have a parameter which takes the instance of the class as an argument.

• In Python, by convention, the first parameter is used as a reference to the calling instance. This parameter is usually called `self`.

```
In [6]: class A:
   
   """Class to test the use of methods""
   
   def greeting(self):
       print("Hello")
```
**self Parameter**

- Even though method definitions have `self` as the first parameter (and we use this variable inside the method body), we don’t pass this parameter explicitly when we invoke the methods.
- This is because whenever we call a method on an object, the object itself is implicitly passed as the first parameter.
- Methods are object-specific functions and this lets us access the object’s properties via the methods directly.
- In some languages (like Java) this parameter is implicit in method definitions but in Python it is explicit and by convention named `self`. 
Understanding Method Calls

• How do we call (or invoke) the greeting method?

  • We create an instance of the class and call the method on that instance using the dot operator as follows:

In [2]: a = A()

In [3]: a.greeting()

Hello
Understanding Method Calls

```
In [1]: class A:
    
    """Class to test the use of methods""
    
    def greeting(self):
        print("Hello")
```

• The following two calls are equivalent:

```python
a = A()

a.greeting()  # method 1

A.greeting(a)  # method 2
```

- Preferred/Standard way
- This works, but don’t do it.
Summary of Methods

• A method differs from a function only in two aspects:
  
  • Methods **belong to a class**, and are defined within a class
  
  • A method’s purpose is to provide a way to access/manipulate objects or instances of the class
  
  • The first parameter in the definition of a method attribute is **the reference to the calling instance**.
  
  • This parameter that references the calling object is (by convention) called **“self”**.
Python special methods:

__init__
Initializing a Class: __init__

- While Python allows you to assign attributes to instances of a class on the fly (and outside the class), it is not the proper way to do so.

- You should *never* assign or modify attributes of an object manually.

- Data attributes (or instance variables) should be initialized as part of the class definition.

- We can achieve this by the Python's special method __init__
  - Like a constructor in Java.

```python
In [1]: class TestInit:
   """This class will test when __init__ is called""
   def __init__(self):
       print("__init__ is called")

In [2]: obj = TestInit()
__init__ is called
```
Initializing a Class: `__init__`

- `__init__`: **Special method** that lets us define how to create an instance of a class, by initializing some data attributes

- This special method is **automatically called** when you create an instance (or object) of the class

- For example, see this test class:

```python
In [1]: class TestInit:
    """This class will test when `__init__` is called""
    def __init__(self):
        print("__init__ is called")

In [2]: obj = TestInit()

    __init__ is called
```
Book Class: **__init__**

```
In [3]: class Book:
    
    """This class represents a book"""
    def __init__(self, name=None, author=None):
        self.name = name
        self.author = author

In [4]: emma = Book('Emma', 'Jane Austen')

In [5]: emma.name
Out[5]: 'Emma'

In [6]: emma.author
Out[6]: 'Jane Austen'

In [7]: hp = Book('Harry Potter')

In [8]: hp.name
Out[8]: 'Harry Potter'
```

Default values of these attributes

self.var tells Python that we are making a new instance variable/attribute

This is the preferred way to define values for attributes
Python special variables:

__dict__, __slots__
Avoid Dynamically Created Attributes

- Attributes of objects are stored in a dictionary \_\_dict\_\_
- Like any other dictionary, you can add items to \_\_dict\_\_ on the fly and there are no predetermined set of keys
- This is why we can even dynamically add attributes to objects (even though this is not recommended)

```python
In [1]: class Book:
    
    """This class represents a book""
    def __init__(self, name=None, author=None):
        self.name = name
        self.author = author

In [2]: b = Book('Emma', 'Jane Austen')

In [3]: b.year = 1815

In [4]: b.year
Out[4]: 1815
```
Avoid Dynamically Created Attributes

- Attributes of objects are stored in a dictionary `__dict__`
- Like any other dictionary, you can add items to `__dict__` on the fly and there are no predetermined set of keys
- This is why we can even **dynamically** add attributes to objects (even though this is not recommended)

```python
In [5]: b.__dict__
```
```
{'name': 'Emma', 'author': 'Jane Austen', 'year': 1815}
```

```python
In [6]: b.ref = 9999777
```

```python
In [7]: b.__dict__
```
```
{'name': 'Emma', 'author': 'Jane Austen', 'year': 1815, 'ref': 9999777}
```
Fix Attributes via \_\_slots\_

- Dynamic creation and assignment of attributes via \_\_dict\_\_ is not desirable.

- Special variable \_\_slots\_
  provides a clean and efficient way to avoid this:
  - Instead of having a \texttt{dynamic} dict that stores the attributes, \_\_slots\_ provide a \texttt{static} structure which \texttt{prohibits} dynamic addition of attributes.
  - \_\_slots\_ stores all of the attributes of the objects of the class as \texttt{strings} in a \texttt{list}.
  - This should be reminiscent of \_\_all\_ special variable for functions.

```python
class Book:
    """This class represents a book""
    \_\_slots\_ = ['name', 'author']
def \_\_init\__(self, name=None, author=None):
    self.name = name
    self.author = author
```
Fix Attributes via **__slots__**

In [8]: class Book:
    """This class represents a book""
    __slots__ = ['name', 'author']
    def __init__(self, name=None, author=None):
        self.name = name
        self.author = author

In [9]: b = Book('Emma', 'Jane Austen')

In [10]: b.name
Out[10]: 'Emma'

In [11]: b.author
Out[11]: 'Jane Austen'

In [12]: b.__dict__

```
AttributeError: 'Book' object has no attribute '__dict__'
```

In [13]: b.year = 1815

```
AttributeError Traceback (most recent call last)
/var/folders/h8/n5myy3jddid7cfv42cw42flt80000gn/T/ipykernel_83812/2981247915.py in <module>
----> 1 b.year = 1815

AttributeError: 'Book' object has no attribute 'year'
```
More Methods for the Book Class
Methods and Data Abstraction

- Methods of a class typically fall into two categories
  - **accessor (getter) methods:** provide “read-only” access to the object’s attributes
  - **mutator (setter) methods:** let us modify the object’s attribute values
- Ideally, we do not allow the user direct access to the object’s attributes
  - Instead we **control access** through accessor and mutator methods
- This approach enforces **data abstraction**
  - Methods provide a “public interface” for manipulating objects
  - Attributes are part of the private implementation - keep them hidden from users
We define the following methods in the class definition of Book to provide an interface to our book objects:

- **numWordsName:** returns the number of words in the name of the book
- **sameAuthorAs:** takes another book object as a parameter and checks if the two books have the same author or not
- **yearSincePub:** takes in the current year and returns the number of years since the book was published

Find the implementation and invocations of these methods in the Jupyter Notebook for the lecture
```python
class Book:
    """This class represents a book with attributes name, author and year""

    __slots__ = ['name', 'author', 'year']  # attributes

    def __init__(self, name=None, author=None, year=None):
        self.name = name
        self.author = author
        self.year = year

    def numWordsName(self):
        """Returns the number of words in name of book""
        return len(self.name.split())

    def sameAuthorAs(self, other):
        """Check if self and other have same author""
        return self.author == other.author

    def yearsSincePub(self, currentYear):
        """Returns the number of years since book was published""
        return currentYear - self.year
```