CMSC201 Computer Science I for Majors

Lecture 16 – Classes

Prof. Jeremy Dixon

Last Class We Covered

- Review of Functions
- Code Design
  - Readability
  - Adaptability
- Top-Down Design
- Modular Development

Any Questions from Last Time?

Today’s Objectives

- To reinforce what exactly it means to write “good quality” code
- To learn more about importing
- To better understand the usefulness of modules
- To learn what a class is, and its various parts
  - To cover vocabulary related to classes
  - To be able to create instances of a class

"Good Code"

- If you were to ask a dozen programmers what it means to write good code, you would get a different answer from each

- What are some characteristics that we have discussed that help you write "good code?"

8 Characteristics of Good Code

1. Readability
   – As we previously discussed, writing code that is easy to understand what it is doing
2. Adaptability (or Extensibility)
   – Relates to how easy it is to change conditions or add features or functionality to the code
3. Efficiency
   – Clean code is fast code

8 Characteristics of Good Code

4. Maintainability
   – Write it for other people to read!
5. Well Structured
   – How well do the different parts of the code work together? Is there a clear flow to the program?
6. Reliability
   – Code is stable and causes little downtime

8 Characteristics of Good Code

7. Follows Standards
   - Code follows a set of guidelines, rules and regulations that are set by the organization
8. Regarded by Peers
   - Good programmers know good code
   - You know you are doing a good programming job when your peers have good things to say about your code and prefer to copy and paste from your programs

Importing and Modules

Reusing Code

- If we take the time to write a good function, we might want to reuse it later!
- It should have the characteristics of good code
  - Clear, efficient, well-commented, and reliable
  - Should be extensively tested to ensure that it performs exactly as we want it to
  - Reusing bad code causes problems in new places!

Modules

- A ***module*** is a Python file that contains definitions (of functions) and other statements
  - Named just like a regular Python file:

    `myModule.py`

- Modules allow us to easily reuse parts of our code that may be generally useful
  - Functions like **`isPrime(num)`** or **`getValidInput(min, max)`**

Importing Modules

- To use a module, we must first ***import*** it
- There are three different ways of importing:

```
import somefile
from   somefile import *
from   somefile import className
```

- The difference is <u>what</u> gets imported from the file and <u>what name</u> refers to it after importing

`import`

- In Lab 9, when we practiced using pdb (Python debugger), we used the import command

```
import pdb
```

- This command imports the entire **`pdb.py`** file
  - Every single thing in the file is now available
  - This includes functions, classes, constants, etc.

import

- To use the things we've imported this way, we need to append the filename and a period to the front of its name
- To access a function called myFunction:

```
myModule.myFunction(34)
```

- To access a class method:

```
myModule.myClass.classMethod()
```

IMPORTANT!

Must include module name as namespace

`from someFile import *`

- Again, everything in the file **`someFile.py`** gets imported (we gain access to it)
  - The star (**`*`**) means we import every single thing from **`someFile.py`**
- Be careful!
  - Using this **`import`** command can easily overwrite an existing function or variable

`from someFile import *`

- When we use this import, if we want to refer to anything, we can just use its name

- We no longer need to use "**`someFile.`**" in front of the things we want to access

```
myFunction(34)
myClass.classMethod()
```

- These things are now in the current ***namespace***

`from someFile import X`

- Only the item **X** in **`someFile.py`** is imported

- After importing **X**, you can refer to it by using just its name (it's in the current namespace)
- But again, be careful!
  - This would overwrite anything already defined in the current namespace that is also called **X**

`from someFile import X`

```
from myModule import myClass
```

- We have imported this class and its methods

```
myClass.classMethod()
```

- But not the other things in myModule.py

`myFunction(34)` *(not imported)*

- We can import multiple things using commas:

```
from myModule import thing1, thing2
```

Directories for Imports

- *Where does Python look for module files?*
  - The list of directories where Python will look for the files to be imported is sys.path

Directories for Imports

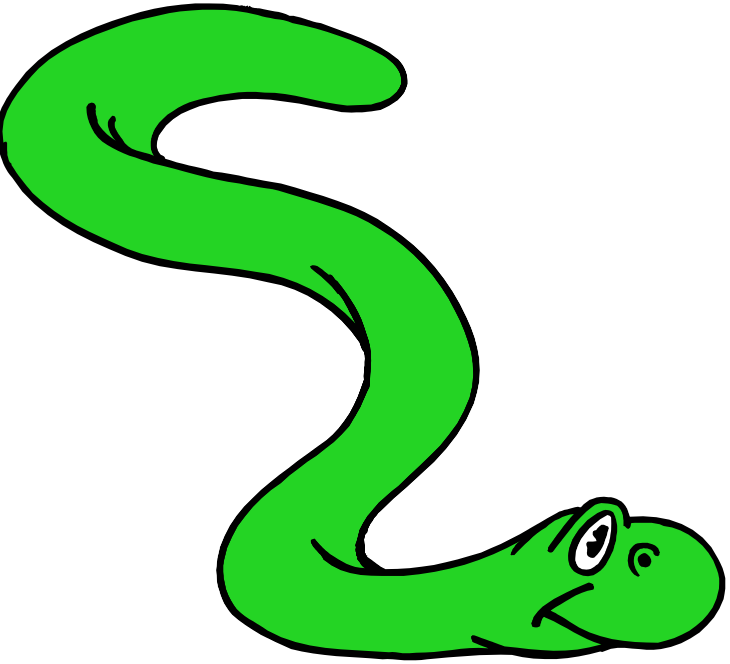
- This is just a variable named ‘path’ stored inside the ‘sys’ module

```
>>> import sys
>>> sys.path
['',
    '/Library/Frameworks/Python.framework/Versions/2.5/lib/
    python2.5/site-packages/setuptools-0.6c5-py2.5.egg’, …]
```

- To add a directory of your own to this list, append it to this list

```
sys.path.append(‘/my/new/path’)
```

Object Oriented Programming: Defining Classes



Classes

- A *class* is a special data type which defines how to build a certain kind of object.
- The *class* also stores some data items that are shared by all the instances of this class
- Classes are blueprints for something
- *Instances* are objects that are created which follow the definition given inside of the class

Classes

- In general, classes contain two things:
  1. Attributes of an object (data members)
     - Usually variables describing the thing
  2. Things that the object can do (methods)
     - Usually functions describing the action

Class Parts

- **Data member:** A class variable or instance variable that holds data associated with a class and its objects.
- **Method :** A special kind of function that is defined in a class definition.

Instances of a Class

- **Object:** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Class Description

- If a class describes a thing, we can think about it in terms of English
  - Object -> Noun
  - Attribute -> Adjective
  - Method (Function) -> Verb

Class to build dogs

Class Example

Characteristic of dog

Method (function) to add tricks

Creating a new dog named ‘Fido’

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

From: https://docs.python.org/2/tutorial/classes.html

Class Example

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

Creates an instance of dog (called an object)

Refer to Fido as “d” from then on

Add a trick to Fido called ‘roll over’

From: https://docs.python.org/2/tutorial/classes.html

Defining a Class

- Instances are objects that are created which follow the definition given inside of the class
- Python doesn't use separate class interface definitions as in some languages
- You just define the class and then use it

Everything an Object?

- Everything in Python is really an object.
  - We've seen hints of this already…
    ```
    “hello”.upper()
    list3.append(‘a’)
    ```
  - New object classes can easily be defined in addition to these built-in data-types.
- In fact, programming in Python is typically done in an object oriented fashion.

Methods in Classes

- Define a *method* in a *class* by including **function** definitions within the scope of the class block
- There must be a special first argument **`self`** in *all* of method definitions which gets bound to the calling instance
- There is usually a special method called **`__init__`** in most classes
- We'll talk about both later…

Class Example student

```
class student:
    def __init__(self, n, a):
        self.full_name = n
        self.age = a
    def get_age(self):
        return self.age
```

Using Class Student

```
def main():
    a = student("John", 19)
    print(a.full_name)
    print(a.get_age())
main()
```

Create new student object named “John”, aged 19

Print an attribute of the student

Call a method of student

Output

```
bash-4.1$ python class_student.py
John
19
bash-4.1$
```

Creating and Deleting Instances

Instantiating Objects

- There is no “new” keyword as in Java.
- Just use the class name with ( ) notation and assign the result to a variable
- `__init__` serves as a constructor for the class. Usually does some initialization work
- The arguments passed to the class name are given to its `__init__()` method
- So, the __init__ method for student is passed “Bob” and 21 and the new class instance is bound to b:

```
b = student(“Bob”, 21)
```

Constructor: __init__

- An __init__ method can take any number of arguments.
- Like other functions or methods, the arguments can be defined with default values, making them optional to the caller.
- However, the first argument self in the definition of __init__ is special…

Self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument self
- In __init__, self refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called
- Similar to the keyword this in Java or C++
- But Python uses self more often than Java uses this

Self

- Although you must specify `self` explicitly when *defining* the method, you don't include it when *calling* the method.
- Python passes it for you automatically

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
  self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

Deleting Instances

- When you are done with an object, you don’t have to delete or free it explicitly.
- Python has automatic garbage collection.
- Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
- Generally works well, few memory leaks
- There’s also no “destructor” method for classes

Access to Attributes and Methods

Definition of Student

```
def main():
    a = student("John", 19)
    print(a.full_name)
    print(a.get_age())
main()
```

Traditional Syntax for Access

```
>>> f = student(“Bob Smith”, 23)
>>> f.full_name # Access attribute
“Bob Smith”
>>> f.get_age() # Access a method
23
```

Accessing Unknown Members

- Problem: Occasionally the name of an attribute or method of a class is only given at run time…
- Solution:

```
getattr(object_instance, string)
```

- **`string`** is a string which contains the name of an attribute or method of a class
- **`getattr(object_instance, string)`** returns a reference to that attribute or method

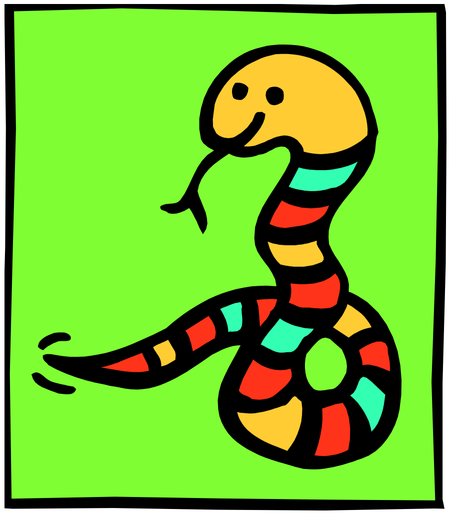
getattr(object_instance, string)

```
>>> f = student(“Bob Smith”, 23)
>>> getattr(f, “full_name”)
“Bob Smith”
>>> getattr(f, “get_age”)
 <method get_age of class studentClass at
  010B3C2>
>>> getattr(f, “get_age”)() # call it
23
>>> getattr(f, “get_birthday”)
# Raises AttributeError – No method!
```

hasattr(object_instance,string)

```
>>> f = student(“Bob Smith”, 23)
>>> hasattr(f, “full_name”)
True
>>> hasattr(f, “get_age”)
True
>>> hasattr(f, “get_birthday”)
False
```

Attributes



Attributes

- Two Kinds of Attributes (Data Members):
  1. Data Attributes also called Instance Variables
  2. Class Attributes also called Class Variables

Important: The word *attribute* and the word *variable* can be used interchangeably for this topic!

Data Attributes

- *Data* attributes or *instance* attributes
  - Variable owned by a *particular instance* of a class
  - Each instance has its own value for it
  - These are the most common kind of attribute

Data Attributes

- Data attributes are created and initialized by an `__init__()` method.
  - Simply assigning to a name creates the attribute
  - Inside the class, refer to data attributes using **`self`**
    - for example, **`self.full_name`**

```
class teacher:
  "A class representing teachers."
  def __init__(self,n):
      self.full_name = n
  def print_name(self):
      print(self.full_name)
```

Instance attribute

Method

Class Attributes

- *Class* attributes
  - Owned by the *class as a whole*
  - *All class instances share the same value for it*
  - Called "static" variables in some languages
  - Good for (1) class-wide constants and (2) building counter of how many instances of the class have been made

Class Attributes

- Because all instances of a class share one copy of a class attribute, when *any* instance changes it, the value is changed for *all* instances
- Class attributes are defined *within* a class definition and *outside* of any method
- Since there is one of these attributes *per class* and not one *per instance*, they're accessed via a different notation:
  - Access class attributes using **`self.__class__.name`** notation -- This is just one way to do this & the safest in general.

Class Attributes

```
class sample:
    x = 23
   def increment(self):
      self.__class__.x += 1
```

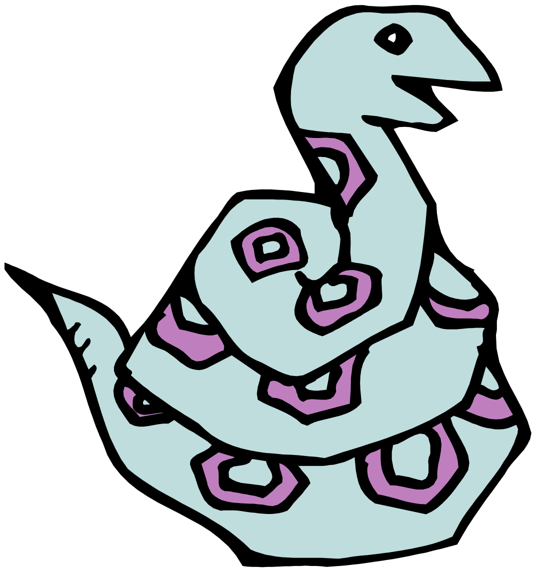
```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

Data vs. Class Attributes

```
class counter:
   overall_total = 0
        # class attribute
   def __init__(self):
       self.my_total = 0
         # data attribute
   def increment(self):
      counter.overall_total = \
      counter.overall_total + 1
      self.my_total = \
      self.my_total + 1
```

```
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

Inheritance



Inheritance

- *Inheritance* is used to indicate that one class will get most or all of its features from a parent class.

For example, computer science students are a specific type of student. Therefore, they probably share attributes with all students. We can use inheritance to use those already defined attributes and methods of students for our computer science students.

Subclasses

- A class can *extend* the definition of another class
  - Allows use (or extension) of methods and attributes already defined in the previous one.
  - New class: *subclass*. Original: *parent*, *ancestor* or *superclass*
- To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.
- Python has no 'extends' keyword like Java.
  - Multiple inheritance is supported.

Subclass Example

New subclass name

```
class cs_student(student):
```

Superclass or parent

Redefining Methods

- To *redefine a method* of the parent class, include a new definition using the same name in the subclass.
  - The old code won't get executed.
- To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent's version of the method.

```
parentClass.methodName(self, a, b, c)
```

  - **The only time you ever explicitly pass 'self' as an argument is when calling a method of an ancestor.**

Inheritance Example

```
class student:
   #"A class representing a student."

   def __init__(self,n,a):
       self.full_name = n
       self.age = a

   def get_age(self):
       return self.age
------------------------------------------------------------
class cs_student (student):
   #"A class extending student."

   def __init__(self,n,a,s):
       student.__init__(self,n,a) #Call __init__ for student
       self.section_num = s

   def get_age(self):    #Redefines get_age method entirely
       print ("Age: " + str(self.age))
```

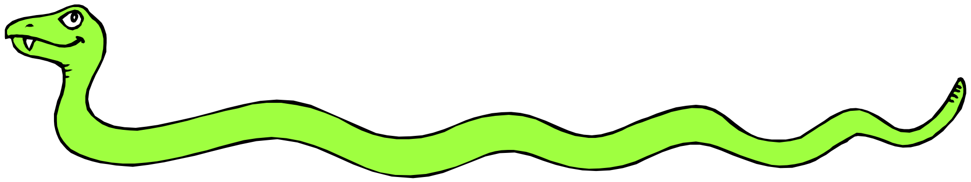
Extending __init__

- Same as for redefining any other method…
  - Commonly, the ancestor's **`__init__`** method is executed in addition to new commands.
  - You'll often see something like this in the **`__init__`** method of subclasses:

```
parentClass.__init__(self, x, y)
```

where parentClass is the name of the parent's class.

Special Built-In Methods and Attributes



Built-In Members of Classes

- Classes contain many methods and attributes that are included by Python even if you don't define them explicitly.
  - Most of these methods define automatic functionality triggered by special operators or usage of that class.
  - The built-in attributes define information that must be stored for all classes.
- All built-in members have double underscores around their names: `__init__` `__doc__`

Special Methods

- For example, the method `__repr__` exists for all classes, and you can always redefine it
- The definition of this method specifies how to turn an instance of the class into a string
  - **`print f`** sometimes calls **`f.__repr__()`** to produce a string for object f
  - If you type **`f`** at the prompt and hit ENTER, then you are also calling **`__repr__`** to determine what to display to the user as output

Special Methods - Example

```
class student:
    ...
     def __repr__(self):
       return “I’m named ” + self.full_name
    ...

>>> f = student(“Bob Smith”, 23)
>>> print f
I’m named Bob Smith
>>> f
“I’m named Bob Smith”
```

Special Methods

- You can redefine these as well:

`__init__`	: The constructor for the class
`__cmp__`	: Define how **==** works for class
`__len__`	: Define how **`len(`** obj **`)`** works
`__copy__`	: Define how to copy a class

- Other built-in methods allow you to give a class the ability to use [ ] notation like an array or ( ) notation like a function call

Special Data Items

- These attributes exist for all classes.

  `__doc__` : Variable for documentation string for class

  `__class__` : Variable which gives you a reference to the class from any instance of it

  `__module__` : Variable which gives a reference to the module in which the particular class is defined

  __dict__ :The dictionary that is actually the namespace for a class (but not its superclasses)

- Useful:
  - **`dir(x)` returns a list of all methods and attributes defined for object x**

Special Data Items – Example

```
>>> f = student(“Bob Smith”, 23)

>>> print f.__doc__
A class representing a student.

>>> f.__class__
< class studentClass at 010B4C6 >

>>> g = f.__class__(“Tom Jones”, 34)
```

Private Data and Methods

- Any attribute/method with 2 leading under-scores in its name (but none at the end) is **private** and can’t be accessed outside of class
- Note: Names with two underscores at the beginning ***and the end*** are for built-in methods or attributes for the class
- Note: There is no ‘protected’ status in Python; so, subclasses would be unable to access these private data either.

Any Other Questions?

Announcements

- Midterm Survey (on Blackboard)
  - Due by Friday, November 6th at 8:59:59 PM
- Project 1 is out
  - Due by Tuesday, November 17th at 8:59:59 PM
  - Do NOT procrastinate!
- Next Class: Objects Continued