Introduction to Python Programming
Week 4

Object Oriented Programming

- OOP – Object Oriented Programming
  - Object-oriented programming (OOP) is a programming language model organized around objects rather than "actions" and data rather than logic.

- In python everything is an object
  - “a string”
  - 22 (an int)
  - 101.5 (a float)
  - even functions
- Operators such as '='. '<', ... are not objects but rather are methods (functions) of the object they operate on.
- In Python the base object is called object and it has a few default methods and attributes

Object Oriented Programming

- **Object-oriented programming (OOP)** is a programming language model organized around objects rather than "actions" and data rather than logic**.**

- **Definitions**
  - **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
  - **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are (a noun).
  - **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
  - **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
  - **Instance variable** − A variable that is defined inside a method and belongs only to the current instance of a class (a noun).
  - **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.

Object Oriented Programming

- **Definitions**
  - **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
  - **Instantiation** − The creation of an instance of a class.
  - **Method** − A special kind of function that is defined in a class definition (a verb).
  - **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.
  - **Operator overloading** − The assignment of more than one function to a particular operator.
  - **Decorator** - A decorator is the name used for a software design pattern. Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated.

Object Oriented Programming

- What is a class?
  - A class is data structure made up of methods and attributes, it is a blueprint for objects
    - methods (verb) – preform actions (like functions do)
    - attributes (noun) – describe (like variables do)
  - A simple python class called Thing

```
>>> class Thing():
>>>     pass
```

Creating Classes

```
class Thing():                          # Class
    """DocSting goes here"""            # docstring
    pass

thing1 = Thing()                        # Instantiation
thing2 = Thing()                        # Instance

type(thing1)
dir(thing1)
print(thing1.__doc__)
```

Creating Classes

```
>>> class Thing():
...     """DocSting goes here"""
...     pass
...
>>> thing1 = Thing()
>>> type(thing1)
<class '__main__.Thing'>
>>> type(thing2)
<class '__main__.Thing'>
```

Creating Classes (cont'd)

```
>>>> dir(thing1)
['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__']
>>> print(thing1.__doc__)
DocSting goes here
>>>
```

Base object: object

- Everything inherits from the class called object

```
>>> print(dir(object))
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
>>> print(object.__doc__)
The most base type
>>>
```

Object Oriented Programming

- In Python everything is an object
- See the program everythings_an_object.py
  - (*Code won't fit here so I'll demo instead*)

Good code

```
import sys #0

class Employee: #1
   """The summary line for a class docstring should fit on one line.

   If the class has public attributes, they may be documented here
   in an "Attributes" section and follow the same formatting as a
   function's "Args" section. Alternatively, attributes may be documented
   inline with the attribute's declaration (see __init__ method below).

   Properties created with the "@property" decorator should be documented
   in the property's getter method.

   Attributes:
       attr1 (str): Description of “attr1".
       attr2 (:obj:`int`, optional): Description of "attr2".

   """ #2
   # These are handles by the __init__
   empCount = 0         # Must be here #3
   # empNumber = 0      #
   # __secretNumber = 0 # This is like a private variable #4
   #
   def __init__(self, name, salary): #5
      self.name = name #6
      self.salary = salary
      Employee.empCount += 1
      self.empNumber = Employee.empCount
      self.__secretNumber = Employee.empCount
   #
   def displayCount(self): #7
     print("Total Employee %d" % Employee.empCount)
   #
   def displayEmployee(self):
      print("Name : ", self.name,  ", Salary: ", self.salary)
   #
   def displaySecret(self):
      print("Secret = %d" % self.__secretNumber)
   #
   def __str__(self): #8
      return "Name: %s\nSalary: %0.02f\nEmployee number: %d" % (self.name, self.salary,
self.empNumber)
   #
#
```

```
"""This would create first object of Employee class"""
emp1 = Employee("Zara", 2000) #9 #10
"""This would create second object of Employee class"""
emp2 = Employee("Manni", 5000)

emp1.displayEmployee()
emp2.displayEmployee()
print("Total Employee %d" % Employee.empCount)

print(emp1) #11 (part of the __str__ above)
print(emp2)

# Trying to access private data
try: #12 Exceptions
   print("#1: %d" % emp1.__secretNumber)
except AttributeError as e:
   print("#1: AttributeError as %s" % e)
except:
   print("First try fail: %s" % sys.exc_info[0])
#
```

```
# Accessing hidden data or Data not hiding ...
try: #12 cont. #13
   print(emp1._Employee__secretNumber)
   print("#2: %d" % emp1._Employee__secretNumber)
except TypeError as e:
   print("#2: TypeError as %s" % e)
except:
   print("#2: Second try fail: %s" % sys.exc_info[0])
#
```

Explaination of Employee

- #0 - Import the sys library for use with exceptions
- #1 - Class
- #2 - Docstring (demo a docstring)
- #3 - Class variable (? & Data member)
- #4 - Instance variable (& Data member & it's a private, more on that later)
- #5 - __init__() Constructor
- #6 - self.name (need to explain)
- #7 - a method
- #8 - __str__ (override)
- #9 - Instantiate (Employee("Zara", 2000) creates emp1)
- #10 - Instance (emp1 is an instance of class Employee)
- #11 - using the override
- #12 - Trying to access a private variable (throws an exception)
- #13 - Trying to access a private variable (so it's not really hidden)

Built-In Class Methods

- The getattr(obj, name[, default]) − to access the attribute of object.
- emp1.__getattribute__('var') # no __getattr__('var')
- The hasattr(obj,name) − to check if an attribute exists or not.
- The setattr(obj,name,value) − to set an attribute. If attribute does not exist, then it would be created.
- emp1.__setattr__('var', 8)
- The delattr(obj, name) − to delete an attribute
- emp1.__delattr__('var')

Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute

- __dict__ − Dictionary containing the class's namespace.
- __doc__ − Class documentation string or none, if undefined.
- __name__ − Class name of the class.
- __class__ − Class name of the Instance.
- __module__ − Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- __bases__ − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Destroying Objects (Garbage Collection)

- Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.
- Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.
- An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with del, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

Class Inheritance

- Inheritance allows you to create a new class that inherits the characteristics of a Parent class that may already have much of what you need.
- Parent class the class a child inherits from (which may be the child of another Parent class)
- Child class inherits methods and attributes from the parent
  - A child inherits from it's parent's class (which may have inherited from it's parent's class, which may … ad infinitum)
  - You can modify) override methods that don't meet your needs
  - Multiple inheritance is possible

Overloading

- Python supports overloading a function or an operator.
  - An operator usually has a function name used to overload the operator (__gt__ for '>')
- **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Operator overloading** − The assignment of more than one function to a particular operator.

And now for something completely different

- type(x) – returns the objects type (class name)
- dir(x) – returns the list of names in the current local scope
- These names can be variables, functions, object, methods and or attributes
- Note Because dir() is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

Copy an object

```
class Human(object):
    """Create a Simple human class"""
    def __init__(self, name, friend=None):
        self.name = name
        self.friend = friend
    def say_name(self):
        print("My name is " + self.name)
    def say_goodnight(self):
        if self.friend is None:
            print("Good night nobody.")
        else:
            print("Good night " + self.friend.name)
        #
    #
#
```

Copy an object

```
>>> fred = Human("Fred")
>>> fred.name = "Fred"
'Fred'
>>> c_fred = fred
>>> c_fred.name = 'Barney'
>>> fred.name
'Barney'
>>> c_fred.name
'Barney'
```

Copy an object

```
import copy
>>> c_fred = copy.deepcopy(fred)
>>> fred.name
'Fred'
>>> c_fred.name
'Fred'
>>> fred.name = "Fredrick"
>>> fred.name
'Fredrick'
>>> c_fred.name
'Fred'
```

Differences Python2 vs. Python3

Differences Python2 vs. Python3

- 2to3
- The print function
- Integer division
- Unicode
- xrange
- Raising exceptions
- Handling exceptions

Differences Python2 vs. Python3

- For-loop variables and the global namespace leak
- Comparing unorderable types
- Parsing user inputs via input()
- Returning iterable objects instead of lists
- Banker’s Rounding
- More articles about Python 2 and Python 3

2to3

- 2to3 - Python2 to Python3 converter

```
$ 2to3 --help
Usage: 2to3 [options] file|dir ...

Options:
  -h, --help            show this help message and exit
  -d, --doctests_only   Fix up doctests only
  -f FIX, --fix=FIX     Each FIX specifies a transformation; default: all
  -j PROCESSES, --processes=PROCESSES
...
```

The Print statement

- Python 2
- print "Python", python_version()
- Python 2.7.9
- print "Hello, World!"
- Hello, World!
- print("Hello, World!")
- Hello, World!
- print "text [", ; print "] print more text on the same line"
- text print more text on the same line
-

- Python 3
- print("Python", python_version())
- Python 3.4.2
- print("Hello, World!")
- Hello, World!
- print("text [", end="");print("] print more text on the same line")
- text [] print more text on the same line

Integer division

- print('3 / 2 =', 3 / 2)
-     ('3 / 2 =', 1)   # Python 2
-     3 / 2 = 1.5   # Python 3
- print('3 // 2 =', 3 // 2)
-     ('3 // 2 =', 1)   # Python 2
-     3 // 2 = 1   # Python 3
- print('3 / 2.0 =', 3 / 2.0)
-     ('3 / 2.0 =', 1.5)   # Python 2
-     3 / 2.0 = 1.5   # Python 3
- print('3 // 2.0 =', 3 // 2.0)
-     ('3 // 2.0 =', 1.0)   # Python 2
-     3 // 2.0 = 1.0   # Python 3
-

- print('3 / 2 =', 3 / 2)
-     ('3 / 2 =', 1)   # Python 2
-     3 / 2 = 1.5   # Python 3
- print('3 // 2 =', 3 // 2)
-     ('3 // 2 =', 1)   # Python 2
-     3 // 2 = 1   # Python 3
- print('3 / 2.0 =', 3 / 2.0)
-     ('3 / 2.0 =', 1.5)   # Python 2
-     3 / 2.0 = 1.5   # Python 3
- print('3 // 2.0 =', 3 // 2.0)
-     ('3 // 2.0 =', 1.0)   # Python 2
-     3 // 2.0 = 1.0   # Python 3
-

Unicode (strings)

- Python 2 has
  - str() types # ASCII (8 bits per character)
  - separate unicode()
  - but no byte type.
- Python 3 has
  - str() # unicode (utf-8) strings
  - 2 byte classes:
    - byte
    - Bytearrays
  - has no unicode() class

Strings-n-things

- There are two types that represent sequences of characters: bytes and str.
  - Instances of bytes contain raw 8-bit values.
  - Instances of str contain Unicode characters.
    - Unicode is more than an 8 bit value (I think)
    - Unicode != ASCII
    - Python 2 strings were based on ASCII

Strings-n-things

- Code

```
s = 'á'     # ua = u'á' – same thing, a str
b = s.encode('utf-8') # convert str to byte
x = b.decode('utf-8') # convert byte to str

y = '\xe1' # str, same as s above ('á')
```

xrange

- Python 2

```
# range()
for i in range(5):
    print(i)
# xrange()
for i in xrange(5):
    print(i)
```

- Python 3

```
for i in range(5):
    print(i)
```

No xrange()

Raising exceptions

- Python 2
  - raise IOError, "file error"
  - raise IOError("file error")

- Python 3
  - raise IOError("file error")

Handling exceptions

- Python 2

```
try:
    let_us_cause_a_NameError
except NameError, err:
    print(err, '--> our error message')
```

- Python 3 (and Python 2)

```
try:
    let_us_cause_a_NameError
except NameError as err:
    print(err, '--> our error message')
```

next() function and .next() method

- Python 2

```
my_generator = (letter for letter in 'abcdefg')
next(my_generator)
my_generator.next()
```

- Python 3

  Don't use .next() just use next()

```
my_generator = (letter for letter in 'abcdefg')
next(my_generator)
next(my_generator)
```

Comparing unorderable types

- Python 2

```
>>> print "[1, 2] > 'foo' = ", [1, 2] > 'foo'
[1, 2] > 'foo' =  False
```

- Python 3

  A TypeError is raised as warning if we try to compare unorderable types.

```
>>> print("[1, 2] > 'foo' = ", [1, 2] > 'foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: list() > str()
```

Parsing user inputs via input()

- Python 2
  - Returns an int (or exception)

    ```
    my_input = input('enter a number: ')
    ```

  - Returns a str

    ```
    my_input = raw_input('enter a number: ')
    ```

- Python 3
  - Returns a str

    ```
    my_input = input('enter a number: ')
    ```

  - Returns a exception (not in Python 3)

    ```
    my_input = raw_input('enter a number: ')
    ```

Returning iterable objects instead of lists

- Python 2

```
>>> print range(3)
[0, 1, 2]
>>> print type(range(3))
<type 'list'>
```

- Python 3

```
>>> print(range(3))
range(0, 3)
>>> print(type(range(3)))
<class 'range'>
>>> print(list(range(3)))
[0, 1, 2]
```

Functions and methods that don’t return lists anymore in Python 3

- zip()
- map()
- filter()
- dictionary’s .keys() method
- dictionary’s .values() method
- dictionary’s .items() method

Banker’s Rounding

- Python 3
  - decimals are rounded to the nearest even number
  - Although it’s an inconvenience for code portability, it’s supposedly a better way of rounding compared to rounding up as it avoids the bias towards large numbers.

•Banker's Rounding

- Python 2
  - round(15.5); round(16.5)

```
16.0
17.0
```

- Python 3
  - round(15.5); round(16.5)

```
16
16
```

URL References

- http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html
- https://jeffknupp.com/blog/2014/06/18/improve-your-python-python-classes-and-object-oriented-programming/
- https://docs.python.org/3/howto/pyporting.html#learn-the-differences-between-python-2-3
- https://docs.python.org/3/howto/pyporting.html
- https://simple.wikipedia.org/wiki/Object-oriented_programming