OBJECT ORIENTED PROGRAMMING

SEBASTIAN HAHN

INTRODUCTION

Object-oriented programming (OOP) is a programming paradigm based on the concept of **objects**, which are data structures that contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

TERMINOLOGY

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 (e.g. x.size).

Method

A special kind of function that is defined in a class definition. (e.g. x.mean())

TERMINOLOGY

Data member

represents a variable or instance variable that holds data associated with a class and its objects.

Class variable

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 aren't used as frequently as instance variables are.

Instance variable

is defined inside a method and belongs only to the current instance of a class.

EXAMPLE - HUMAN

```
class Human(object):

    # definition of a class variable
    population = 0

    def __init__(self, name):
        # definition of an instance variable
        self.name = name
        Human.population += 1

    def say_hi(self):
        print("Hi, my name is {:}".format(self.name))

# make an instance of a class
jason = Human("Jason")
jason.say_hi()
```

```
Hi, my name is Jason
```

TERMINOLOGY

Instance

Represents an individual object of a certain class

`__init__()`

Special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.

`__del__()`

is a special method, which is called when an instance is destroyed; e.g. when it is no longer referenced.

EXAMPLE - HUMAN

```
class Human(object):

    def __init__(self, name):
        self.name = name
        print("I can see the light")

    def say_hi(self):
        print("Hi, my name is {:}".format(self.name))

    def __del__(self):
        print("Bye bye")

jason = Human("Jason")
jason.say_hi()
del jason
```

```
I can see the light
Hi, my name is Jason
Bye bye
```

SELF = REFERENCE TO CURRENT INSTANCE

- When defining your class methods, you must explicitly list `self` as the first argument for each method.
- However, when you call your class method from outside, you do not specify anything for the self argument; you skip it entirely, and Python automatically adds the instance reference for you

INHERITANCE, ENCAPSULATION AND POLYMORPHISM

Encapsulation

refers to the creation of self-contained modules that bind processing functions to the data.

Inheritance (passes "knowledge")

Classes are created in hierarchies, and inheritance lets the structure and methods in one class pass down the hierarchy. That means less programming is required when adding functions to complex systems. The ability to reuse existing objects is considered a major advantage of object technology.

INHERITANCE, ENCAPSULATION AND POLYMORPHISM

Polymorphism (takes any shape)

Object-oriented programming lets programmers create procedures for objects whose exact type is not known until runtime. It is the provision of a single interface to entities of different types.

ENCAPSULATION

In programming languages, encapsulation is used to refer to one of two related but distinct notions, and sometimes to the combination thereof:

- A language mechanism for restricting access to some of the object's components.
- A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.

ENCAPSULATION IN PYTHON

- Python does not really support encapsulation because it does not support data hiding through private and protected members (like e.g. Java, C++).
- However some pseudo-encapsulation can be done.
  - e.g. with double underline, i.e. `__attrName`, it can be referred to within the class itself as `self.__attrName`, but outside of the class, it is named `object._className__attrName`
- Therefore, while it can prevent accidents, this pseudo-encapsulation cannot really protect data from hostile code.
- This works for attributes and methods

EXAMPLE - HUMAN

```
class Human(object):
    def __init__(self, name, age):
        self.name = name
        self.__age = age

jason = Human("Jason", 33)
print("Name: {:}".format(jason.name))
print("Age: {:}".format(jason._Human__age))
# print("Age {:}".format(jason.age))
```

```
Name: Jason
Age: 33
```

INHERITANCE

In object-oriented programming, inheritance is when an object or class is based on another object or class, using the same implementation (inheriting from a class) specifying implementation to maintain the same behavior (realizing an interface; inheriting behavior). A mechanism to transfer the characteristics of a class to other classes that are derived from it.

EXAMPLE - HUMAN

```
class Human(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def say_hi(self):
        print("Hi, my name is {:}".format(self.name))

class Student(Human):
    def say_hi(self):
        print("Hey, I'm {:} and {:}".format(self.name, self.age))

class Teacher(Human):
    def say_bye(self):
        print("Bye")

jason = Student("Jason", 22)
jason.say_hi()
betty = Teacher("Betty", 44)
betty.say_hi()
betty.say_bye()
```

```
Hey, I'm Jason and 22
Hi, my name is Betty
Bye
```

EXAMPLE - PIZZA

```
import math

class Pizza(object):
    def __init__(self, radius):
        self.radius = radius
    def cf(self):
        return 2 * math.pi * self.radius
    def __str__(self):
        return "CF={:4.2f}".format(self.cf())

class Calzone(Pizza):
    def cf(self):
        c = super(Calzone, self).cf()
        return c / 2. + 2 * self.radius

p = Pizza(10)
print(p)
p2 = Calzone(10)
print(p2)
```

```
CF=62.83
CF=51.42
```

SPECIAL FUNCTION TO OVERRIDE

- String representation
  - `__str__()`, `__repr__()`, `__unicode__()`
- Customize attribute access
  - `__setattr__()`, `__getattr__()`, `__delattr__()`

FUNCTION OVERLOADING

Function overloading (or method overloading) is the ability to create multiple methods of the same name with different implementations. Calls to an overloaded function will run a specific implementation of that function appropriate to the context of the call, allowing one function call to perform different tasks depending on context.

```
def add_bullet(sprite, start, direction, speed):
def add_bullet(sprite, start, headto, spead, acceleration):
def add_bullet(sprite, curve, speed):
```

METHOD OVERLOADING IS NOT SUPPORTED IN PYTHON, BUT

```
class A:

    def method_a(self, i=None):
        if i == None:
            print 'first method'
        else:
            print 'second method', i
```

OPERATOR OVERLOADING

In programming, operator overloading (less commonly known as operator ad hoc polymorphism) is a specific case of polymorphism, where different operators have different implementations depending on their arguments. Operator overloading is generally defined by the language, the programmer, or both.

EXAMPLE - VECTOR

```
import math

class Vector(object):

    def __init__(self, *args):
        self.coords = args

    def __str__(self):
        return str(self.coords)

    def __add__(self, other):
        coords = tuple(map(sum, zip(self.coords, other.coords)))
        return Vector(*coords)

    def __getitem__(self, index):
        return self.coords[index]

v1, v2 = Vector(2, 10), Vector(5, -2)
print v1 + v2
print v2[1], v2.coords[1]
```

```
(7, 8)
-2 -2
```

OPERATORS TO OVERLOAD

More information can be found in the Python documentation, some examples:

- `__pos__()`, `__neg__()`, `__inv__()`, `__abs__()`, `__len__()`
- `__add__()`, `__sub__()`, `__and__()`, `__or__()`, `__xor__()`, ...

STATIC METHODS

- Static methods are a special case of methods.
- Sometimes, you'll write code that belongs to a class, but that doesn't use the object itself at all.
- Static methods have no `self` argument and don't require you to instantiate the class before using them

EXAMPLE - HUMAN

```
class Human(object):

    population = 0

    def __init__(self, name):
        self.name = name
        Human.population += 1

    @staticmethod
    def how_many():
        print("Population: {:}".format(Human.population))

    def __del__(self):
        Human.population -= 1

jason = Human("Jason")
betty = Human("Betty")
jason.how_many()
robin = Human("Robin")
Human.how_many()
```

```
Population: 2
Population: 3
```

DUCK TYPING

- In computer programming with object-oriented programming languages, duck typing is a layer of programming language and design rules on top of typing
- Duck typing is concerned with establishing the suitability of an object for some purpose
- With normal typing, suitability is assumed to be determined by an object's type only. In duck typing, an object's suitability is determined by the presence of certain methods and properties, rather than the actual type of the object

DUCK TYPING

The name of the concept refers to the duck test, attributed to James Whitcomb Riley, which may be phrased as follows:

> *"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."*

DUCK TYPING IN PYTHON

```
class Duck(object):
    def quack(self):
        print "Quack, quack!"
    def fly(self):
        print "Flap, Flap!"

class Person(object):
    def quack(self):
        print "I'm Quackin'!"
    def fly(self):
        print "I'm Flyin'!"

def in_the_forest(mallard):
    mallard.quack()
    mallard.fly()

in_the_forest(Duck())
in_the_forest(Person())
```

```
Quack, quack!
Flap, Flap!
I'm Quackin'!
I'm Flyin'!
```

BUILT-IN CLASS ATTRIBUTES

`__dict__`
Dictionary containing the class's namespace

`__doc__`
Class documentation string (or None if not defined)

`__name__`
Class name

`__module__`
Module in which the class is defined (this 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

EXAMPLE - EMPLOYEE

```
class Employee(object):
    """
    Class docstring.
    """
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

```
Employee.__doc__:
    Class docstring.

Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<type 'object'>,)
Employee.__dict__: {'__dict__': <attribute '__dict__' of 'Employee'
objects>, '__module__': '__main__', '__weakref__': <attribute '__wea
kref__' of 'Employee' objects>, '__doc__': '\n    Class docstring.\n
    ', '__init__': <function __init__ at 0x7fa9cc6aee60>}
```

MORE STUFF

- Functions like `issubclass(sub,sup)`, `isinstance(obj, Class)`, ...
- Subclass from several parents
  - `class SubClassName (ParentClass1[, ParentClass2, ...])`
- Defining class methods or abtract methods
- Inheritance from build-in types (e.g. lists, dicts)
- Metaclasses, class factory, descriptors

EXERCISE

Create a python module implementing

- `class Shape(object)`
  - Instance variable: color
- `class Rectangle(Shape)`
  - Instance variable: width, height
  - Methods: calculate_area()
- `class Circle(Shape)`
  - Instance variable: radius
  - Methods: calculate_area()