CMPSC-122: Intermediate Programming

Spring 2018

Department of Computer Science & Engineering
The Pennsylvania State University

1. Object-Oriented Programming

In all the programs we have written until now, we have designed our program around functions (blocks of statements which manipulate data). This is called the *procedure-oriented* way of programming. There is another way of organizing your program which is to combine data and functionality and wrap it inside something called an object. This is called the *object-oriented* programming paradigm. Most of the time you can use procedural programming, but when writing large programs or have a problem that is better suited to this method, you can use *object-oriented* programming techniques.

Classes and objects are the two main aspects of *object-oriented* programming. A *class* creates a new type where objects are instances of the class. Objects can store data using ordinary variables that belong to the object. Variables that belong to an object or class are referred to as *fields*. Objects can also have functionality by using functions that belong to a class. Such functions are called *methods* of the class. This terminology is important because it helps us to differentiate between functions and variables which are independent and those which belong to a class or object.

1.1 Classes

We can create virtual objects in Python. This virtual objects can contain variables and methods. A *class* is just an organized collection of variables and functions. Classes are defined similarly to functions, with the following syntax:

```
class ClassName:
    statements
```

To use a class, you can create a representation of that class, called an *instance* or *object*, and assign it to a variable. For example:

```
class User:
    x=12345

    def sayHello(self):
        return ("Hello, world!!!")
```

Class User created, thus, User.x and User.sayHello() are valid attribute references, returning an integer and a function object, respectively.

```
john = User()
```

Instance of the class User stored in the variable *john* (virtual objects)

```
print(john.x)
print(john.sayHello())
```

Call methods owned by virtual objects

Output:
12345
'Hello, world!!!'

The *self* variable represents the object itself, and we use it inside procedures to refer to other functions and variables that are defined within the scope of our class. Calling *self*.variable or *self*.function tells Python to look only inside the scope of the class for the respective variable or function, ignoring those functions and variables that are defined either outside of our class or even within a function that belongs to our class. All procedures that are defined in a class must be given at least this one parameter as a means of referring to the class itself. The class instance is automatically passed in as an argument when the function is called.

Let's consider the example below:

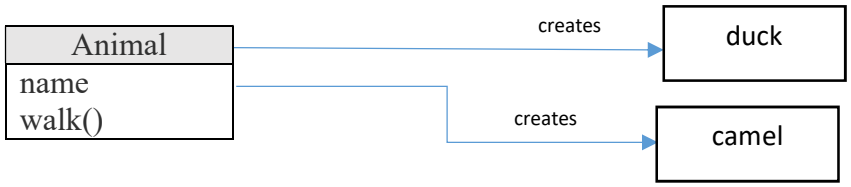
```
class Animal:
    def __init__(self,name):
        self.name = name

    def walk(self):
        print(self.name + ' walks.')


duck = Animal('Duck')
duck.walk()

camel = Animal('Bactrian Camel')
camel.walk()
```

In this code, we have created two objects called 'duck' and 'camel' from the class Animal. The class has a method (*walk*) that can be called on each object. We also have a method called *__init__*(), this is a method that is always called when a new object is created. Remember that the *self* keyword is required for every method. Once the object is created, we can call its methods and use its variables indefinitely. Every object of the same class has the same methods, but its variables contents may differ.



1.1.1 The *self*

Class methods have only one specific difference from ordinary functions: they must have an extra first name that has to be added to the beginning of the parameter list, but you DO NOT give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object itself, and by convention, it is given the name *self*. Although, you can give any name for this parameter, it is *strongly recommended* that you use the name self so any reader of your program will immediately recognize it.

1.2 Instantiation

The instantiation operation creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Python has a special built-in method named __init__(), which evaluates when the object is created, to handle this type of object initialization:

```
class MyClass:

    def __init__(self):
        self.i = 12345


a = MyClass()
print(a.i)          # 12345
```

When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance. So in this example, a new, initialized instance can be obtained by `a = MyClass()`. The __init__() method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to __init__(). For example:

```
class ComplexNumber:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

x = ComplexNumber(5.0, -3.5)
print(x.r, x.i)        # 5.0 -3.5
```

The __init__() method is called the *constructor* and is always called when creating an object. The variables owned by the class are known as class attributes.

1.3 Methods

We have already discussed that classes/objects can have methods just like functions except that we have an extra *self* variable. There are many method names which have special significance in Python classes. In Section 1.2, we discussed how the __init__ method is run as soon as an object of a class is instantiated (created). The method is useful to do any initialization (like passing initial values to your object) you want to do with your object. Usually, a method is called right after it is bound. In the *User* class example from Section 1.1, *john.sayHello()* will return the string 'Hello, world!!!'. However, it is not necessary to call a method right away: *john.sayHello* is a method object, and can be stored away and called at a later time. For example:

```
class User:
    x=12345

    def sayHello(self):
        return ("Hello, world!!!")

john = User()
srt_john = john.sayHello

while True:
   print(srt_john())
```

Without executing the code, what is the output of this program?

You may have noticed that *john.sayHello()* was called without an argument above, even though the function definition for *sayHello()* specified an argument. The particular thing about methods is that the instance object is passed as the first argument of the function: the call *john.sayHello()* is exactly equivalent to *User.sayHello(john)*. In general, calling a method with a list of *n* arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's instance object before the first argument.

In Python you can define a method in such a way that there are multiple ways to call it. Given a single method or function, we can specify the number of parameters ourselves. Depending on the function definition, it can be called with zero, one, two or more parameters. This is known as ***method overloading***. Not all programming languages support method overloading, but Python does.

1.4 Encapsulation

In an object-oriented Python program, you can restrict access to methods and variables. This can prevent the data from being modified by accident and is known as *encapsulation*.

1.4.1 Private methods

From the example below, we have created a class named *Car* which has two methods: drive() and updateSoftware(). When we create the car object *toyota*, it will call the private method __updateSoftware(). This function cannot be called on the object directly, only from within the class.

```
class Car:
    def __init__(self):
        self.__updateSoftware()

    def drive(self):
        print('Driving')

    def __updateSoftware(self):
        print('Updating software...')

toyota = Car()
print(toyota.drive())
```

```
Updating software…
Driving
```

Encapsulation prevents from accessing accidentally, but not intentionally. The private attributes and methods are not really hidden, they're renamed adding "_Car" in the beginning of their name. The method can actually be called using `toyota._Car__updateSoftware()`

1.4.2 Private variables

Variables can be private which can be useful on many occasions. A private variable can only be changed within a class method and not outside of the class. Objects can hold crucial data for your application and you do not want that data to be changeable from anywhere in the code.

```
class Car:
    __maxspeed = 0
    __name = ""

    def __init__(self):
        self.__maxspeed = 200
        self.__name = "Supercar"

    def drive(self):
        print('Driving. Maxspeed= ' + str(self.__maxspeed))

toyota = Car()
toyota.drive()
toyota.__maxspeed = 100
toyota.drive()
```

```
Driving. Maxspeed= 200
Driving. Maxspeed= 200
```

__maxspeed will no change because it is private

If you want to change the value of a private variable, a setter method is used. This is simply a method that sets the value of a private variable.

```
class Car:
    __maxspeed = 0
    __name = ""

    def __init__(self):
        self.__maxspeed = 200
        self.__name = "Supercar"

    def drive(self):
        print('Driving. Maxspeed= ' + str(self.__maxspeed))

    def setMaxSpeed(self,speed):
        self.__maxspeed = speed

toyota = Car()
toyota.drive()
toyota.setMaxSpeed(520)
toyota.drive()
```

```
Driving. Maxspeed= 200
Driving. Maxspeed= 520
```

Where privacy stops…

Other programming languages have protected class methods too, but Python does not. Encapsulation gives you more control over the degree of coupling in your code, it allows a class to change its implementation without affecting other parts of the code.

Summarizing:

Type	Description
public methods	Accessible from anywhere
private methods	Accessible only in their own class. Starts with two underscores
public variables	Accessible from anywhere
private variables	Accessible only in their own class or by a method if defined. Starts with two underscores

1.5 Class And Object Variables

Now that we have discussed the functionality part of classes and objects, we can learn more about the data part. The data part is related to ordinary variables that are bound to the namespaces of the classes and objects. This means that these names are valid within the context of these classes and objects only. There are two types of fields, class variables and object variables which are classified depending on whether the class or the object *owns* the variables respectively:

- Class variables are shared, which means they can be accessed by all instances of that class. There is only one copy of the class variable and when any one object makes a change to a class variable, that change will be seen by all the other instances.
- Object variables are owned by each individual object/instance of the class. In this case, each object has its own copy of the field and are not related in any way to the field by the same name in a different instance.

```
class Dog:
    kind = 'canine'          ⟹ class variable shared by all instances

    def __init__(self, name):
        self.name = name     ⟹ instance variable unique to each instance


d = Dog('Harry')
e = Dog('Pichi')
print(d.kind)       # canine
print(e.kind)       # canine
print(d.name)       # Harry
print(e.name)       # Pichi
```

1.6 Inheritance

One of the major benefits of object-oriented programming is reuse of code and one of the ways this is achieved is through the inheritance mechanism. Inheritance can be best imagined as implementing a type and subtype relationship between classes. Classes can inherit functionality of other classes. If an object is created using a class that inherits from a superclass, the object will contain the methods of both the class and the superclass. The same holds true for variables of both the superclass and the class that inherits from the super class. The following example illustrates inheritance in action:

The code below creates the classes *User* and *Student*. Although the *Student* class looks very much like a standard class, it has *User* as a parameter, which means all functionality of the class *User* is accessible in the *Student* class. Then it creates one instance called *jeff* which outputs its given name and one instance called *emily* that also outputs its given name and skill .

```
class User:
    name = ""

    def __init__(self, name):
        self.name = name

    def printName(self):
        print("Name  = " + self.name)

class Student(User):

    def __init__(self, name):
        self.name = name
    def skill(self):
        print("Programming in Python")


jeff = User("Jeff")
jeff.printName()

emily = Student("Emily")
emily.printName()
emily.skill()
```

jeff is an instance of *User* and can only access the method *printName*, while *emily* is an instance of *Student*, a class with inheritance from *User*, and can access both the methods in *Student* and *User*. Thus, the output of the above code is:

```
Name  = Jeff
Name  = Emily
Programming in Python
```

1.7 Polymorphism

Polymorphism is an important feature of class definition in Python that is utilized when you have commonly named methods across classes or subclasses. This allows functions to use objects of any of these polymorphic classes without needing to be aware of distinctions across the classes. Polymorphism can be carried out through inheritance, with subclasses making use of base class methods or overriding them. For example, a button is an object that comes in many types or forms (round button, check button, square button, button with image) but they do share the same logic: onClick(). We access them using the same method using polymorphism.

1.7.1 Polymorphism with a function

Let's illustrate polymorphism with a function using an example. We have created two classes, *Cow* and *Dog*, both can make a distinct sound. We then make two instances and call their action using the same method.

```
class Cow(object):
    def sound(self):
        print("Mooo")

class Dog(object):
    def sound(self):
        print("Woof woof")

def makeSound(animalType):
    animalType.sound()


cowObj = Cow()
dogObj = Dog()

makeSound(cowObj)
makeSound(dogObj)
```

```
cowObj.sound() ==
Cow.sound(cowObj)

dogObj.sound() ==
Dog.sound(dogObj)
```

```
Output:

Mooo
Woof woof
```

1.7.2 Polymorphism with abstract class

A class called as abstract defines abstract methods and child class must override these methods if you want to use them. The abstract methods always throw the exception NotImplementedError. Let's illustrate this type of polymorphism using an example. A *Document* object can be represented in various forms (PDF, Word, Excel, ...). The abstract structure is defined in *Document* class:

```
class Document:
    def __init__(self, name):
        self.name = name

    def show(self):
        raise NotImplementedError("Subclass must implement abstract method")
```

method can not be used because it always throws an error.
Subclass must implement abstract method

```
class PDF(Document):
    def show(self):
        print ("Show PDF document:", self.name)

class Word(Document):
    def show(self):
        print ("Show Word document:", self.name)
```

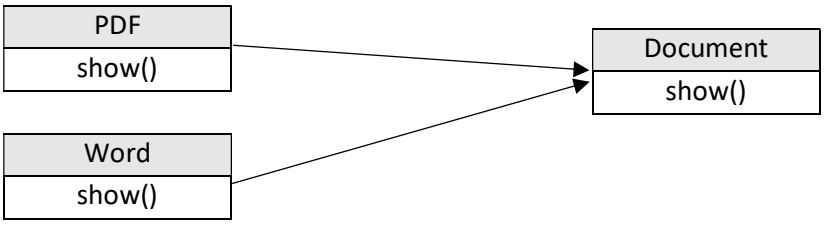
Child classes that override method of parent class (*Document*)

```
documents = [ PDF("Python tutorial"),
              Word("Java Tutorial"),
              PDF("C Tutorial") ]
for doc in documents :
    doc.show()
```

```
Output:

Show PDF document: Python tutorial
Show Word document: Java Tutorial
Show PDF document: C Tutorial
```

The abstract class *Document* does not have any implementation but defines the structure (in form of functions) that all forms must have. If we define the function *show*() then both the PDF and Word classes must have the show() function.



References

https://docs.python.org/3/tutorial/classes.html

https://www.digitalocean.com/community/tutorials/how-to-apply-polymorphism-to-classes-in-python-3