CSCI120

Introduction to Computer Science I using Python 3

Object-Oriented Programming

Before you can make an object in Python 3, you must first define the class...

Definition of the `Person` class:

```
class Person():
    ''' Instantiates a Person object with given name. '''
    def __init__(self, first_name, last_name):
        ''' Initializes private instance variables _firstname and _lastname.  '''
        self._firstname = first_name
        self._lastname = last_name

    def __str__(self):
        ''' Returns the state of the Person object. '''
        return self._firstname + " " + self._lastname
```

After the keyword `class`, you type the name of the class to be created. Notice by convention, the first letter of a class name is uppercase (e.g. `Person`).

After the class name (`Person`) is (optionally) a set of parentheses `()` which optionally contain the class called `object` (from which all classes originate) – the object class is included implicitly if it is not explicitly typed within the parentheses.

```
class Person():
```

The parentheses may contain the name of another class from which the current class inherits properties and methods. Inheritance is explained later in this presentation.

Note that the parentheses are themselves optional if you don't specify a class name - but you must include the colon :

```
class Person():
```

```
class Person:
```

is equivalent to

```
class Person():
```

is equivalent to

```
class Person(object):
```

Definition of the `Person` class:

```
class Person():
    ''' Instantiates a Person object with given name. '''
    def __init__(self, first_name, last_name):
        ''' Initializes private instance variables _firstname and _lastname.  '''
        self._firstname = first_name
        self._lastname = last_name

    def __str__(self):
        ''' Returns the state of the Person object. '''
        return self._firstname + " " + self._lastname
```

The highlighted text are *docstrings* for the class and methods

There is an important programming standard method of supplying documentation for functions, methods and classes. Programmers provide *docstrings* in their code. They are very easy to use and supply information that can be extracted by automated tools to get information about your functions, methods or classes.

The following is a quotation from python.org:

"The docstring for a function should summarize its behavior and document its arguments, return value(s), side effects, exceptions raised, and restrictions on when it can be called (all if applicable)."

```
def average_quiz(student_name, quiz1, quiz2):
    ''' (str, int, int) -> float
    Returns the average quiz score for the student. '''
    pass
```

docstrings start with triple quotes (triple single or triple double) immediately after the function header, indented (tabbed) to the same level as the code in the function body. In brackets it shows the parameter types followed by `->` and the return type of the function. In the example function above, there are three parameters of type str, int, int and the function returns an object of type float.

Often, when we a designing a program, we will provide a template for a function or class, just like the example above, which will be completed later. We can place the keyword `pass` in the function or class body which is just a placeholder for the code which will be added later. The `pass` is silently ignored when the program module runs...

To print the docstring of a class called `Person` from within the Python 3 code, you would type:

```
print(Person.__doc__)
```

or

```
print(Person.methodname.__doc__)
```

Note: `__doc__` *is a special attribute available to all classes and stores the docstrings for the class and its methods.*

To print the docstring of a class called `Person` from the command line in a Python shell window (after running the code containing the class), you would type:

```
>>> help(Person)
```

Note: `__doc__` *is a special attribute available to all classes and stores the docstrings for the class and its methods.*

Note: you can obtain help on built-in classes like `int`, `str`, `float`, etc in exactly the same way:

```
>>> help(str)
```

Definition of the `Person` class:

```
class Person():
    ''' Instantiates a Person object with given name. '''
    def __init__(self, first_name, last_name):
        ''' Initializes private instance variables _firstname and _lastname.  '''
        self._firstname = first_name
        self._lastname = last_name

    def __str__(self):
        ''' Returns the state of the Person object. '''
        return self._firstname + " " + self._lastname
```

The __init__ method:

```
class Person():
    ''' Instantiates a Person object with given name. '''
    def __init__(self, first_name, last_name):
        ''' Initializes private instance variables _firstname and _lastname.  '''
        self._firstname = first_name
        self._lastname = last_name
```

A class may define a special method named __init__ which does some initialization work and serves as a constructor for the class. Like other functions or methods __init__ can take any number of arguments. The __init__ method is run as soon as an object of a class is instantiated and class instantiation automatically invokes __init__() for the newly-created class instance. *This means that the programmer does not call the* __init__ *method directly!*

The `__init__` method:

```
class Person():
    ''' Instantiates a Person object with given name. '''
    def __init__(self, first_name, last_name):
        ''' Initializes private instance variables _firstname and _lastname.  '''
        self._firstname = first_name
        self._lastname = last_name
```

The first parameter of the `__init__` method is called `self` and stands for the name of the actual object being instantiated (created) by `__init__`.

In C++ and Java, `self` is called `this`.

The __init__ method:

```
class Person():
    ''' Instantiates a Person object with given name. '''
    def __init__(self, first_name, last_name):
        ''' Initializes private instance variables _firstname and _lastname.  '''
        self._firstname = first_name
        self._lastname = last_name
```

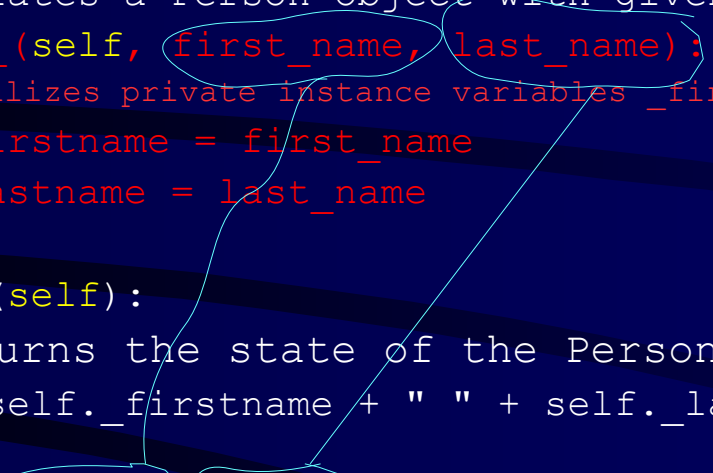
The __init__ method is used to provide initial values to the object's attributes (in this case _firstname and _lastname) when the object is first created.

Create two `Person` objects and call a method:

```
class Person():
    ''' Instantiates a Person object with given name. '''
    def __init__(self, first_name, last_name):
        ''' Initializes private instance variables _firstname and _lastname.  '''
        self._firstname = first_name
        self._lastname = last_name

    def __str__(self):
        ''' Returns the state of the Person object. '''
        return self._firstname + " " + self._lastname

person1 = Person("Anne","Dawson")
person2 = Person("Tom","Lee")
print(person1)
print(person2)
```



Initial values of the attributes are passed to the `__init__` method when the object is instantiated. *Note, you do not pass anything for the `self` parameter, that is handled automatically.*

Example Program: `13-01.py`

All example programs can be found here:

`http://www.annedawson.net/python3programs.html`

Object Oriented Programming Videos

Click here for the Object Oriented Programming videos:

Part 1: https://youtu.be/edsORpbGz2U

Part 2: https://youtu.be/UyMoN5I2S20

The `__str__` method:

```
def __str__(self):
    ''' Returns the state of the Person object. '''
    return self._firstname + " " + self._lastname
```

The `__str__` method is used to print the *state* of an object, i.e. the current values of the instance variables, in this case, `_firstname` and `_lastname`.

The `__str__` method:

```
def __str__(self):
   ''' Returns the state of the Person object. '''
    return self._firstname + " " + self._lastname
```

The `__str__` method, like the `__init__` method is not called directly in a program, but is invoked automatically when the `print()` function is passed an object, e.g.

```
print(person1)
```

Create two `Person` objects and invoke the `__str__` method:

```
class Person():
    ''' Instantiates a Person object with given name. '''
    def __init__(self, first_name, last_name):
        ''' Initializes private instance variables _firstname and _lastname.  '''
        self._firstname = first_name
        self._lastname = last_name

    def __str__(self):
        ''' Returns the state of the Person object. '''
        return self._firstname + " " + self._lastname

person1 = Person("Anne","Dawson")
person2 = Person("Tom","Lee")
print(person1)
print(person2)
```

Note, you do not pass anything for the `self` parameter, that is handled automatically.

```
Program output:
>>>
Anne Dawson
Tom Lee
>>>
```

Example Program: `13-01.py`

The four fundamental OOP concepts

There are four very important features of OOP which all programmers must master...

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation

Abstraction

Inheritance

Polymorphism

Encapsulation

Encapsulation is a mechanism of wrapping the data (instance variables) and code acting on the data (methods) together into a single unit (a capsule). Encapsulation provides data hiding from outside of the class, the details of how data are stored and manipulated are hidden.

Encapsulation

Encapsulation involves hiding your instance variables. This means making them “private”. Python doesn’t actually provide the ability to make instance variables totally private, but by giving instance variables a name starting with an underscore (_), this signals to the programmer that the variable should only be accessed by a method belonging to the same class.

Encapsulation

The methods of the class are made public so that code from outside of the class can call (invoke) the method to view or change the value of an instance variable.

Accessor methods view the value of an instance variable

```
def getFirstname(self):
    ''' Returns the instance variable _firstname. '''
    return self._firstname

def getLastname(self):
    ''' Returns the instance variable _lastname. '''
    return self._lastname
```

Accessor methods are read-only – they only inspect the value of an instance variable – they do not change its value. See program `13-02.py`

```
print(person1.getFirstname())
print(person1.getLastname())
```

Mutator methods change the value of an instance variable

```
def setFirstname(self,newFirstname):
   ''' Assign a value to the instance variable _firstname. '''
    self._firstname = newFirstname

def setLastname(self,newLastname):
   ''' Assign a value to the instance variable _lastname. '''
    self._lastname = newLastname
```

Mutator methods change the value of an instance variable.
See program `13-03.py`

```
person1.setFirstname("Annie")
print(person1.getFirstname())
```

Other methods used with objects

```
def reverseName(self):
    '''Reverses the full name   '''
    return self._lastname + " " + self._firstname
```

See program `13-04.py`

```
print(person1.reverseName())
```

Encapsulation

Abstraction

Inheritance

Polymorphism

Encapsulation leads to Abstraction...

Encapsulation ensures that the details of how data are stored and manipulated are hidden.

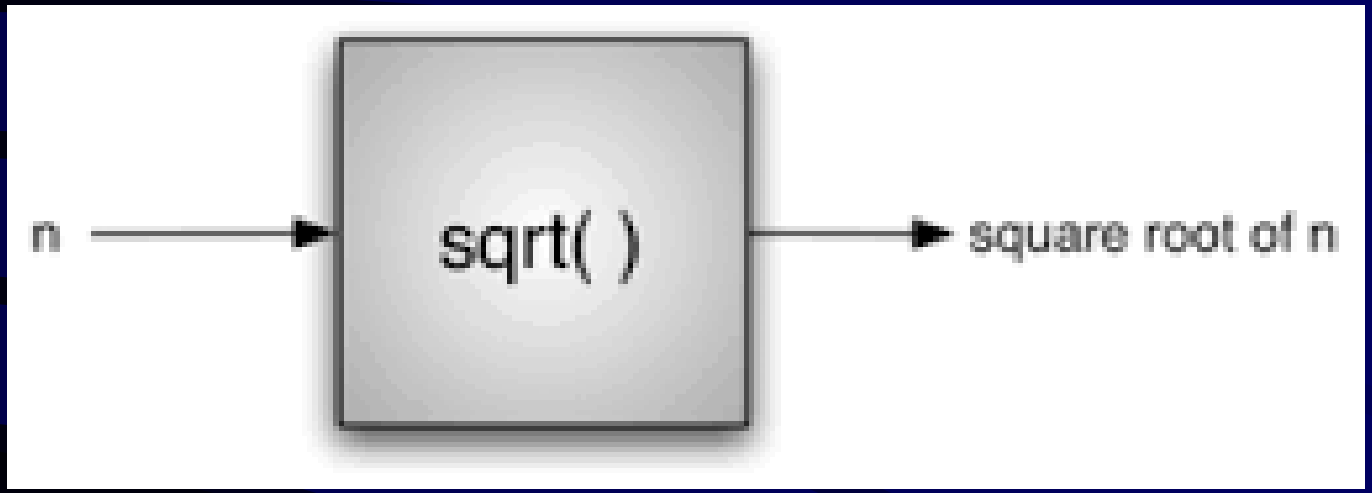
The user of a class's methods only needs to know the required input to the method (the arguments) and the expected output from the method (returned value). How the method is actually implemented is irrelevant to the user of the method.

Abstraction

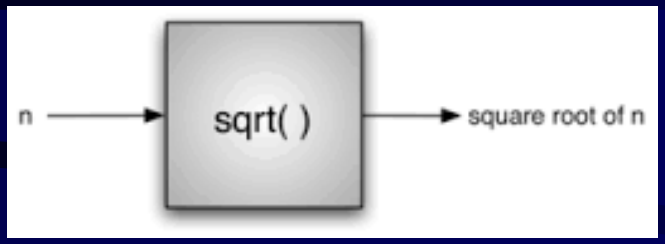
The user of a class's methods only needs to know the required input to the method (the arguments) and the expected output from the method (returned value). How the method is actually implemented is irrelevant.

Treating a method as a "black box" in this manner is called *abstraction*.

Abstraction – the Black Box model...



Abstraction – the Black Box model



As long as a calling object knows what to send to the method, and what is expected to come out of the method, then that's all that's required to be able to use the method.

How do we communicate to a programmer the behaviour (inputs and output) of a method?

The docstring explains the behaviour of a function or method (or class)

```
def average_quiz(student_name, quiz1, quiz2):
    ''' (str, int, int) -> float
    Returns the average quiz score for the student'''
    pass
```

Encapsulation

Abstraction

Inheritance

Polymorphism

Inheritance

Inheritance is a feature of object-oriented programming that allows us to define a new class (called a subclass, child class, or derived class) that is a modified version of an existing class (called the superclass, parent class or base class).

Inheritance

The subclass inherits all of the properties (instance variables) and methods of the superclass in addition to adding some of its own variables and methods.

Inheritance

```
class Student(Person):
```

Inheritance

We might, for example want to define a new class called Student which is a special case of a Person. The Student object needs all the attributes of the Person class (variable and methods) as well as some attributes of its own, like student number and GPA.

Inheritance

```
class Student(Person):
   '''Instantiates a Student object with given name. '''
   def __init__(self, first_name, last_name, student_number=0, G_P_A=0):
       '''Initializes variables _firstname, _lastname, _SN and _GPA. '''
       super().__init__(first_name, last_name) # import base's parameters
       '''Initializes instance variables _firstname and _lastname. '''
       self._SN = student_number
       self._GPA = G_P_A
```

Inheritance

See example program: 13-05.py

Encapsulation

Abstraction

Inheritance

Polymorphism

A subclass can change the behaviour of an inherited method. If a method defined in the subclass (Student) has the same name as a method in its superclass (Person), then the child's method will override the parent's method.

The feature of every OOP language that allows two classes to use the same method name but with different implementations is called polymorphism. The word polymorphism is derived from the Greek word meaning “many forms”.

```
print(student1.reverseName())
# The reverseName method of the Student class
# overrides the same method of the Parent class.
# This is an example of polymorphism
```

All example programs can be found here:

`http://www.annedawson.net/python3programs.html`

Object Oriented Programming Videos

Click here for the Object Oriented Programming videos:

Part 1: https://youtu.be/edsORpbGz2U

Part 2: https://youtu.be/UyMoN5I2S20

End of Python3_Prog_OOP.odp

Last updated: Friday 5th January 2018, 7:36 PT, AD