Object-oriented programming in Python

In this part of the Python programming tutorial, we will talk about object oriented programming in Python.

There are three widely used programming paradigms there: procedural programming, functional programming, and object-oriented programming. Pytho supports both procedural and object-oriented programming. There is some support for functional programming, too.

Object-oriented programming (OOP) is a programming paradigm that uses objects and their interactions to design applications and computer programs.

There are some basic programming concepts in OOP:

- Abstraction
- Polymorphism
- Encapsulation
- Inheritance

The *abstraction* is simplifying complex reality by modeling classes appropriate to the problem. The *polymorphism* is the process of using an operator or function in different ways for different data input. The *encapsulation* hides the implementation details of a class from other objects. The *inheritance* is a way to form new classes using classes that have already been defined.

Objects

Everything in Python is an object. Objects are basic building blocks of a Python OOP program.

object_types.py

```
#!/usr/bin/python

# object_types.py

import sys

def function(): pass

print type(1)
print type("")
print type([])
print type({})
print type(())
print type(object)
print type(function)
print type(sys)
```

In this example we show that all these entities are in fact objects. The `type()` function returns the type of the object specified.

```
$ ./object_types.py
<type 'int'>
<type 'str'>
<type 'list'>
<type 'dict'>
<type 'tuple'>
<type 'type'>
<type 'function'>
<type 'module'>
```

Integers, strings, lists, dictionaries, tuples, functions, and modules are Python objects.

The class keyword

The previous objects were all built-in objects of the Python programming language. The user defined objects are created using the `class` keyword. The cla is a blueprint that defines a nature of a future object. From classes we construct instances. An *instance* is a specific object created from a particular class. For example, Huck might be an instance of a Dog class.

first_object.py

```
#!/usr/bin/python

# first_object.py

class First(object):
   pass

fr = First()

print type(fr)
print type(First)
```

This is our first class. The body of the class is left empty for now. It is a convention to give classes a name that starts with a capital letter.

```
class First(object):
   pass
```

Here we define the `First` class. It inherits from the base `object`. This is so called new-style class definition.

```
fr = First()
```

Here we create a new instance of the `First` class. Or in other words, we instantiate the `First` class. The `fr` is a reference to our new object.

```
$ ./first_object.py
<class '__main__.First'>
<type 'type'>
```

Here we see that `fr` is an instance object of the `First` class.

Inside a class, we can define attributes and methods. An *attribute* is a characteristic of an object. This can be for example a salary of an employee. A *meth* defines operations that we can perform with our objects. A method might define a cancellation of an account. Technically, attributes are variables and methods are functions defined inside a class.

Object initialization

A special method called `__init__()` is used to initialize an object.

object_initialization.py

```
#!/usr/bin/python

# object_initialization.py

class Being(object):

    def __init__(self):
        print "Being is initialized"

Being()
```

We have a `Being` class. The special method `__init__()` is called automatically right after the object has been created.

```
$ ./object_initialization.py
Being is initialized
```

This is the example output.

Attributes

Attributes are characteristics of an object. Attributes are set in the `__init__()` method.

attributes.py

```
#!/usr/bin/python

# attributes.py

class Cat(object):
   def __init__(self, name):
      self.name = name

missy = Cat('Missy')
lucky = Cat('Lucky')

print missy.name
print lucky.name
```

In this code example, we have a `Cat` class. The special method `__init__()` is called automatically right after the object has been created.

```
def __init__(self, name):
```

Each method in a class definition begins with a reference to the instance object. It is by convention named `self`. There is nothing special about the `self` name. We could name it this, for example. The second parameter, `name`, is the argument. The value is passed during the class initialization.

```
self.name = name
```

Here we pass an attribute to an instance object.

```
missy = Cat('Missy')
lucky = Cat('Lucky')
```

Here we create two objects: cats Missy and Lucky. The number of arguments must correspond to the `__init__()` method of the class definition. The 'Missy and 'Lucky' strings become the `name` parameter of the `__init__()` method.

```
print missy.name
print lucky.name
```

Here we print the attributes of the two cat objects. Each instance of a class can have their own attributes.

```
$ ./attributes.py
Missy
Lucky
```

The attributes can be assigned dynamically, not just during initialization. This is demonstrated by the next example.

attributes_dynamic.py

```
#!/usr/bin/python

# attributes_dynamic.py

class Person(object):
   pass

p = Person()
p.age = 24
p.name = "Peter"

print "{0} is {1} years old".format(p.name, p.age)
```

We define and create an empty `Person` class.

```
p.age = 24
p.name = "Peter"
```

Here we create two attributes dynamically: age and name.

```
$ ./attributes_dynamic.py
24 is Peter years old
```

So far, we have been talking about the instance attributes. In Python there are also so called *class object attributes*. Class object attributes are same for all instances of a class.

class_attribute.py

```
#!/usr/bin/python

# class_attribute.py

class Cat(object):
   species = 'mammal'

   def __init__(self, name, age):

      self.name = name
      self.age = age


missy = Cat('Missy', 3)
lucky = Cat('Lucky', 5)

print missy.name, missy.age
print lucky.name, lucky.age

print Cat.species
print missy.__class__.species
print lucky.__class__.species
```

In our example, we have two cats with specific name and age attributes. Both cats share some characteristics. Missy and Lucky are both mammals. This is reflected in a class level attribute species. The attribute is defined outside any method name in the body of a class.

```
print Cat.species
print missy.__class__.species
```

There are two ways, how we can access the class object attributes. Either via the name of the `Cat` class, or with the help of a special `__class__` attribute.

```
$ ./class_attribute.py
Missy 3
Lucky 5
mammal
mammal
mammal
```

Methods

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are essential in the *encapsulation* concept of the OOP paradigm. For example, we might have a `connect()` method in our `AccessDatabase` class. We need not to be informe how exactly the method connect connects to the database. We only know that it is used to connect to a database. This is essential in dividing responsibiliti in programming, especially in large applications.

methods.py

```
#!/usr/bin/python

# methods.py

class Circle(object):

   pi = 3.141592

   def __init__(self, radius=1):
       self.radius = radius

   def area(self):
       return self.radius * self.radius * Circle.pi

   def setRadius(self, radius):
       self.radius = radius

   def getRadius(self):
       return self.radius


c = Circle()

c.setRadius(5)
print c.getRadius()
print c.area()
```

In the code example, we have a `Circle` class. We define three new methods.

```
def area(self):
    return self.radius * self.radius * Circle.pi
```

The `area()` method returns the area of a circle.

```
def setRadius(self, radius):
    self.radius = radius
```

The `setRadius()` method sets a new value for a radius attribute.

```
def getRadius(self):
    return self.radius
```

The `getRadius()` method returns the current radius.

```
c.setRadius(5)
```

The method is called on an instance object. The `c` object is paired with the `self` parameter of the class definition. The number 5 is paired with the `radius` parameter.

```
$ ./methods.py
5
78.5398
```

In Python, we can call methods in two ways. There are *bounded* and *unbounded* method calls.

bound_unbound_methods.py

```
#!/usr/bin/python

# bound_unbound_methods.py

class Methods:
   def __init__(self):
      self.name = 'Methods'

   def getName(self):
      return self.name


m = Methods()

print m.getName()
print Methods.getName(m)
```

In this example, we demostrate both method calls.

```
print m.getName()
```

This is the *bounded* method call. The Python interpreter automatically pairs the `m` instance with the self parameter.

```
print Methods.getName(m)
```

And this is the *unbounded* method call. The instance object is explicitly given to the `getName()` method.

```
$ ./bound_unbound_methods.py
Methods
Methods
```

Inheritance

The inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called *derived* classes, the classes that we derive from are called *base* classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classe (descendants) override or extend the functionality of base classes (ancestors).

inheritance.py

```
#!/usr/bin/python

# inheritance.py

class Animal(object):

   def __init__(self):
      print "Animal created"

   def whoAmI(self):
      print "Animal"

   def eat(self):
      print "Eating"


class Dog(Animal):

   def __init__(self):
      Animal.__init__(self)
      print "Dog created"

   def whoAmI(self):
      print "Dog"

   def bark(self):
      print "Woof!"

d = Dog()
d.whoAmI()
d.eat()
d.bark()
```

In this example, we have two classes: `Animal` and `Dog`. The `Animal` is the base class, the `Dog` is the derived class. The derived class inherits the functionality o the base class. It is shown by the `eat()` method. The derived class modifies existing behaviour of the base class, shown by the `whoAmI()` method. Finally, th derived class extends the functionality of the base class, by defining a new `bark()` method.

```
class Dog(Animal):

   def __init__(self):
      Animal.__init__(self)
      print "Dog created"
```

We put the ancestor classes in round brackets after the name of the descendant class. If the derived class provides its own `__init__()` method, it must explicitly call the base class `__init__()` method.

```
$ ./inherit.py
Animal created
Dog created
Dog
Eating
Woof!
```

Polymorphism

The polymorphism is the process of using an operator or function in different ways for different data input. In practical terms, polymorphism means that class B inherits from class A, it doesn't have to inherit everything about class A; it can do some of the things that class A does differently. (wikipedia)

```
#!/usr/bin/python

# basic.py


a = "alfa"
b = (1, 2, 3, 4)
c = ['o', 'm', 'e', 'g', 'a']

print a[2]
print b[1]
print c[3]
```

Python programming language uses polymorphism extensively in built-in types. Here we use the same indexing operator for three different data types.

```
$ ./basic.py
f
2
g
```

Polymorphism is most commonly used when dealing with inheritance.

```
#!/usr/bin/python

# polymorphism.py


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

   def talk(self):
      pass


class Cat(Animal):
   def talk(self):
      print "Meow!"


class Dog(Animal):
   def talk(self):
      print "Woof!"


a = Animal()
a.talk()

c = Cat("Missy")
c.talk()

d = Dog("Rocky")
d.talk()
```

Here we have two species: a dog and a cat. Both are animals. The `Dog` class and the `Cat` class inherit the `Animal` class. They have a `talk()` method, which gives different output for them.

```
$ ./polymorphism.py
Meow!
Woof!
```

Special Methods

Classes in Python programming language can implement certain operations with special method names. These methods are not called directly, but by a specific language syntax. This is similar to what is known as *operator overloading* in C++ or Ruby.

```
#!/usr/bin/python

# book.py

class Book:
   def __init__(self, title, author, pages):
      print "A book is created"
      self.title = title
      self.author = author
      self.pages = pages

   def __str__(self):
      return "Title:%s , author:%s, pages:%s " % \
              (self.title, self.author, self.pages)

   def __len__(self):
      return self.pages

   def __del__(self):
      print "A book is destroyed"


book = Book("Inside Steve's Brain", "Leander Kahney", 304)

print book
print len(book)
del book
```

In our code example, we have a book class. Here we introduce four special methods. The `__init__()`, `__str__()`, `__len__()` and the `__del__()` methods.

```
book = Book("Inside Steve's Brain", "Leander Kahney", 304)
```

Here we call the `__init__()` method. The method creates a new instance of a Book class.

```
print book
```

The print keyword calls the `__str__()` method. This method should return an informal string representation of an object.

```
print len(book)
```

The `len()` function invokes the `__len__()` method. In our case, we print the number of pages of our book.

```
del book
```

The del keyword deletes an object. It calls the `__del__()` method.

In the next example we implement a vector class and demonstrate addition and substraction operations on it.

```
#!/usr/bin/python

# vector.py


class Vector:

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

  def __str__(self):
    return repr(self.data)

  def __add__(self, other):
    data = []
    for j in range(len(self.data)):
      data.append(self.data[j] + other.data[j])
    return Vector(data)

  def __sub__(self, other):
    data = []
    for j in range(len(self.data)):
      data.append(self.data[j] - other.data[j])
    return Vector(data)


x = Vector([1, 2, 3])
y = Vector([3, 0, 2])
print x + y
print y - x
```

```
def __add__(self, other):
  data = []
  for j in range(len(self.data)):
    data.append(self.data[j] + other.data[j])
  return Vector(data)
```

Here we implement the addition operation of vectors. The `__add__()` method is called, when we add two `Vector` objects with the `+` operator. Here we add each member of the respective vectors.

```
$ ./vector.py
[4, 2, 5]
[2, -2, -1]
```

In this part of the Python tutorial, we have covered object-oriented programming in Python.

Home Contents Top of Page Previous Ne