Quickly Learn Object Oriented Programming

Introduction

This tutorial explains how to get started with object-oriented programming, using code examples and brief, practical explanations.

This tutorial is for you if:

- You already know how to use classes and objects on other languages and you just want to quickly learn how to do it in Python.
- You don't know how to use objects and want to learn in an easy way without going through a lot of theory and talk.

This tutorial is NOT for you if:

- You're interested in advanced Python programming and want to learn the fine details of Python's inner working.

You can skip the introduction if you know what objects are.

What is an Object?

Creating an object is like creating your own variable type.

This object can include multiple variables (called attributes/properties) for storing different types of data. It can also have its own set of functions (called methods), to manipulate the object's properties.

You can create many objects of the same type (same class), each one with their own values.

If you were going to code a program for a bank, you could create an Object for user accounts. Each new user would be an object with its own information.

Example:

```
object_Account1 = BankAccount()    # Create 2 new Objects
object_Account2 = BankAccount()    # newAccount is the name of our own created typ

print object_Account1.userName    # Prints the name of the account
print object_Account1.balance    # Prints the account balance

print object_Account2.userName    # Prints a different name
print object_Account2.balance    # Prints a different balance
```

Both objects belong to the same class, so they have the same variables (also called properties or attributes) but different values for those variables.

Objects can also have their own functions (called methods) to operate on their attributes.

```
print object_Account1.withdrawMoney(100)
print object_Account2.depositMoney(300)
```

An object's attributes and methods are defined in its class.

Object Oriented Programming is supposed to be the most practical approach for dealing with today's programming situations.

Enough Theory.

So let's get into it right away and start punching some code.

Creating a Class with Methods and Properties:

```
class BankAccount:
    """ Class for Bank Accounts"""

    type = 'Normal Account'    # variable shared by all objects of the class

    def __init__(self, name):
        # default variables unique to each object:
        self.userName = name
        self.balance = 0.0
        self.balance = 0.0

    # Object Methods:

    def showBalance(self):
        print self.balance
        return

    def withdrawMoney(self, amount):
        self.balance -= amount
        return

    def depositMoney(self, amount):
        self.balance += amount
        return
```

The special method **__init__()** is automatically invoked when a new object is created.
You can pass arguments to it and use it to instantiate objects with customized initial data.

Create an Object from a Class and Use it:

```
object1 = BankAccount("Im 1")    # create new instance of ClassName
object2 = BankAccount("Me 2")    # create another instance

# access object properties
print object1.userName    #'Im 1'
print object2.userName    #'Me 2'

# access object methods
object1.depositMoney(100)
object2.depositMoney(200)

print object1.balance    # 'Im 1'
print object2.balance    # 'Me 2'
# same as:
object1.showBalance()    # 100
object2.showBalance()    # 200

print object1.type    # 'Normal Account'
print object2.type    # 'Normal Account'
```

By simply assigning a value, you can also assign a new property to an object accessible only to that object.
You can also delete an object property by using '**del**':

```
object1.discount_code = 'secret'

print object1.discount_code    # 'secret'
print object2.discount_code    # Throws Error: AttributeError: object2 instance ha

del object1.discount_code    # back to how it originally was
```

Quick Theory

You will eventually realize that everything in Python is an object. It means that everything in Python has a class. You can find out which class an object belongs to by using the default property **__class__**:

```
account3 = BankAccount('Me 3')
print account3.__class__    # __main__.BankAccount

string = 'Cat'
print string.__class__        # <type 'str'>
```

So yes, **[1,2].reverse()** is a method of the native class 'list'.

Inheritance

You can make a new class inherit all the original methods and attributes from another class, but you can still give the new class its own properties to extend its functionality:

```
class ExecutiveAccount( BankAccount ):
    # Overriden attribute
    type = 'Executive Account'

    # Extended functionality
    def requestCredit(self, amount):
        self.balance += amount

executive = ExecutiveAccount('CEO')
```

When you access an object property, the interpreter looks for it in the object class. If it's not there, the interpreter proceeds all the way up the class inheritance chain until the first occurence is found.

You can call the original property from an overriding class like this:

```
class SpecialExecutiveAccount( ExecutiveAccount ):
    def requestCredit(self, amount):
        self.balance = ExecutiveAccount.requestCredit(self, amount)
        return
```

__init__() methods are not inherited

You can use isinstance() to check if an object is an instance or subinstance of a class.
You can use issubclass() to check if one class derives from another:

```
isinstance(executive, SpecialExecutiveAccount)      # False
isinstance(executive, BankAccount)                  # True, derived
issubclass(SpecialExecutiveAccount, BankAccount)    # True
```

Multiple Inheritance

There is support for inherting from multiple classes:

```
class CustomizedClass( ExecutiveAccount, EnterpriseAccount, InternationalAccount )
    # . . .
```

If the interpreter doesn't find a requested attribute in the class, the interpreter proceeds left to right, in Executive and then in Enterprise and so on.

Iterators

You probably know that you can iterate through the elements of a Python object with for loops:

```
for i in ['a', 'b', 'c']: print i
# 'a'
# 'b'
# 'c'

for i in 'abc': print i # same as above,
```

You can also use iterators with the function **iter()**:

```
iterator = iter('abc')
iterator.next() # 'a'
iterator.next() # 'b'
iterator.next() # 'c'
```

You can add iterating functionality to your objects by defining the special **__iter__()** method, along with a next() method:

```
class IterableAccount( BankAccount ):
    def __init__(self, name):
        self.userName = name
        self.index = len(self.userName)

    def __iter__(self):
        return self

    def next(self):
        # iterators will iterate through the userName in reverse order
        if self.index == 0:
            raise StopIteration
        self.index -= 1
        return self.userName[self.index]

iterable = IterableAccount ('stack')
for i in iterable: print i
# 'k'
# 'c'
# 'a'
# 't'
# 's'
```

Sign up for our newsletter:

you@example.com

SIGN UP NOW