A Short and Incomplete Introduction to Python

Part 9: Object-oriented programming

Riccardo Murri <riccardo.murri@uzh.ch>
S3IT: Services and Support for Science IT,
University of Zurich

June 8, 2017

Objects

What's an *object*?

A Python object is a bundle of variables and functions.

What variable names and functions comprise an object is defined by the object's *class*.

From one class specification, many objects can be *instanciated*. Different instances can assign different values to the object variables.

Variables and functions in an instance are collectively called *instance attributes*; functions are also termed *instance methods*.

Example: the `datetime` object, I

```
>>> from datetime import date
>>> dt1 = date(2012, 9, 28)
>>> dt2 = date(2012, 10, 1)
```

To instanciate an object, call the class name like a function.

Example: the `datetime` object, II

```
>>> dir(dt1)
['__add__', '__class__', ..., 'ctime', 'day',
'fromordinal', 'fromtimestamp', 'isocalendar',
'isoformat', 'isoweekday', 'max', 'min', 'month',
'replace', 'resolution', 'strftime', 'timetuple',
'today', 'toordinal', 'weekday', 'year']
```

The `dir` function can list all objects attributes.

Note there is no distinction between instance variables and methods!

Example: the `datetime` object, III

```
>>> dt1.day
28
>>> dt1.month
9
>>> dt1.year
2012
```

Access to object attributes is done by suffixing the instance name with the attribute name, separated by a dot ".".

Example: the `datetime` object, IV

```
>>> dt1 = date(2012, 9, 28)
>>> dt2 = date(2012, 10, 1)

>>> dt1.day
28
>>> dt2.day
1
```

The same attribute can have different values in different instances!

Instance methods

```
>>> dt1.isoformat()
'2012-09-28'
```

Invoke an instance method just like any other function.

Objects *vs* modules

Modules are also namespaces of variables and functions.

The dot operator '.' is also used to access variables and functions from modules. The `dir()` function is also used to list variables and functions from modules.

But each module has *one and only one* instance in a Python program.

User-defined classes

A *2D vector* in Python

This code defines a Python object that implements a 2D vector.

```
class Vector(object):
  """A 2D Vector."""
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def add(self, other):
    return Vector(self.x+other.x,
                  self.y+other.y)
  def mul(self, scalar):
    return Vector(scalar*self.x, scalar*self.y)
  def show(self):
    return ("<%g,%g>" % (self.x, self.y))
```

Source code available at:

https://raw.github.com/gc3-uzh-ch/python-course/master/vector.py

What does `Vector` do?

We can create vectors by initializing them with the two coordinates (x, y):

```
>>> u = Vector(1,0)
>>> v = Vector(0,1)
```

The `show` method shows vector coordinates:

```
>>> u.show()
'<1,0>'
>>> v.show()
'<0,1>'
```

The `add` method implements vector addition:

```
>>> w = u.add(v)
>>> w.show()
'<1,1>'
```

The `mul` method implements scalar multiplication:

```
>>> v2 = v.mul(2)
>>> v2.show()
'<0,2>'
```

User-defined classes, I

```
class Vector(object):
  """A 2D Vector."""
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def add(self, other):
    return Vector(self.x+other.x,
                  self.y+other.y)
  def mul(self, scalar):
    return Vector(scalar*self.x, scalar*self.y)
  def show(self):
    return ("<%g,%g>" % (self.x, self.y))
```

A class definition starts with the keyword `class`. The class definition is indented relative to the `class` statement.

User-defined classes, II

```
class Vector (object) :
  """A 2D Vector."""
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def add(self, other):
    return Vector(self.x+other.x,
                  self.y+other.y)
  def mul(self, scalar):
    return Vector(scalar*self.x, scalar*self.y)
  def show(self):
    return ("<%g,%g>" % (self.x, self.y))
```

This identifies user-defined classes.

(Do not leave it out or you'll get an "old-style" class, which is deprecated behavior.)

User-defined classes, II

```
class Vector(object):
    """A 2D Vector."""
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def add(self, other):
        return Vector(self.x+other.x,
                      self.y+other.y)
    def mul(self, scalar):
        return Vector(scalar*self.x, scalar*self.y)
    def show(self):
        return ("<%g,%g>" % (self.x, self.y))
```

Classes can have docstrings.
The content of a class docstring will be shown as help text for that class.

User-defined classes, IV

```
class Vector(object):
  """A 2D Vector."""
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def add(self, other):
    return Vector(self.x+other.x,
                  self.y+other.y)
  def mul(self, scalar):
    return Vector(scalar*self.x, scalar*self.y)
  def show(self):
    return ("<%g,%g>" % (self.x, self.y))
```

The **def** keyword introduces a method definition.

Every method *must* have at least one argument, named **self**.

The `self` argument

Every method of a Python object always has `self` as first argument.

However, you do not specify it when calling a method: it's automatically inserted by Python:

```
>>> class ShowSelf(object):
...   def show(self):
...     print(self)
...
>>> x = ShowSelf() # construct instance
>>> x.show() # 'self' automatically inserted!
<__main__.ShowSelf object at 0x299e150>
```

The `self` name is a reference to the object instance itself. You *need to* use `self` when accessing methods or attributes of this instance.

Name resolution rules, I

Within a function body, names are resolved according to the LEGB rule:

- L Local scope: any names defined in the current function;
- E Enclosing function scope: names defined in enclosing functions (outermost last);
- G global scope: names defined in the toplevel of the enclosing module;
- B Built-in names (i.e., Python's `__builtins__` module).

Any name that is not in one of the above scopes *must* be qualified.

So you have to write `self.x` to reference an attribute in this instance, `datetime.date` to mean a class defined in module `date`, etc.

Name resolution rules, II

```
import datetime as dt

def today():
    td = dt.date.today()
    return "today is " + td.isoformat()

def hey(name):
    print("Hey " + name + "; " + today())

hey("you")
```

Unqualified name within a function: resolves to a local variable.

Name resolution rules, III

```
import datetime as dt

def today():
  td = dt.date.today()
  return "today is " + td.isoformat()

def hey(name):
  print("Hey " + name + "; " + today())

hey("you")
```

Unqualified name: since there is no local variable by that name, it resolves to a module-level binding, i.e., to the `today` function defined above.

Name resolution rules, IV

```
import datetime as dt

def today():
  td = dt.date.today()
  return "today is " + td.isoformat()

def hey(name):
  print("Hey " + name + "; " + today())

hey("you")
```

Unqualified name: resolves to the `dt` name created at global scope by the `import` statement.

Name resolution rules, V

```
import datetime as dt

def today():
  td = dt.date.today()
  return "today is " + td.isoformat()

def hey(name):
  print("Hey " + name + "; " + today())

hey("you")
```

Qualified name: instructs Python to search the `date` attribute within the `dt` module.

Name resolution rules, VI

```
import datetime as dt

def today():
  td = dt.date.today()
  return "today is " + td.isoformat()

def hey(name):
  print("Hey " + name + "; " + today())

hey("you")
```

Qualified name: Python searches the `isoformat` attribute within the `td` object instance.

Name resolution rules, VI

```
class Vector(object):
  def __init__(self, x, y):
    self.x = x
    self.y = y
  # ...
```

Unqualified name: resolves to a local variable in scope of function `__init__`.

Name resolution rules, VII

```
class Vector(object):
  def __init__(self, x, y):
    self.x = x
    self.y = y
  # ...
```

Qualified names: resolve to attributes in object `self`.

(Actually, `self.x = ...` *creates* the attribute `x` on `self` if it does not exist yet.)

Object initialization

The `__init__` method has a special meaning: it is called when an instance is created.

```
class Vector(object):
  """A 2D Vector."""
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def add(self, other):
    return Vector(self.x+other.x, self.y+other.y)
  def mul(self, scalar):
    return Vector(scalar*self.x, scalar*self.y)
  def show(self):
    return ("<%g,%g>" % (self.x, self.y))
```

Constructors

The `__init__` method is the object constructor. It should *never* return any value.

You never call `__init__` directly, it is invoked by Python when a new object is created from the class:

```
# calls Vector.__init__
v = Vector(0,1)
```

The arguments to `__init__` are the arguments you should supply when creating a class instance.

(Again, minus the `self` part which is automatically inserted by Python.)

Exercise A: Add a new method `norm` the `Vector` class: if `v` is an instance of class `Vector`, then calling `v.norm()` returns the norm $\sqrt{v_x^2 + v_y^2}$ of the associated vector. (You will need the math standard module for computing square roots.)

Exercise B: Add a new method `unit` to the `Vector` class: if `v` is an instance of class `Vector`, then calling `v.unit()` returns the vector `u` having the same direction as `v` but norm 1.

Appendix

Everything is an object!

The `dir` built-in function is used to list the attributes of an object.

```
>>> dir("hello!")
```

Everything is an object!

The `dir` built-in function is used to list the attributes of an object.

```
>>> dir("hello!")
['__add__', '__class__', '__contains__',
 '__delattr__', '__doc__', '__eq__',
 ...
'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

. . . a string is an object!

Everything is an object!

```
>>> dir([1,2,3])
['__add__', '__class__', '__contains__',
...
'append', 'count', 'extend',
'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

...a `list` is an object!

Everything is an object!

Indeed, you can do:

```
>>> "hello world!".split()
['hello', 'world!']
```

```
>>> [1,1,2,3,5].count(1)
2
```

Everything is an object!

```
>>> dir(1)
```

Everything is an object!

```
>>> dir(1)
['__abs__', '__add__', '__and__',
...
'conjugate', 'denominator',
'imag', 'numerator', 'real']
```

...an `int` is an object!

```
>>> (1).numerator
2
>>> (1).denominator
1
```