University of Zurich[UZH]

Grid Computing Competence Center

Object-oriented Python, I

GC3: Grid Computing Competence Center,
University of Zurich

Sep. 28, 2012

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*.

Objects *vs* modules

Modules are also namespaces of variables and functions.

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

Example: the `datetime` object, I

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

Import the `date` class from the standard library module `datetime`

Example: the `datetime` object, II

```
>>> 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, III

```
>>> 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, IV

```
>>> 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, V

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

The same attribute can have different values in different instances!

No access control

There are no "public"/"private"/etc. qualifiers for object attributes.

Any code can create/read/overwrite/delete _any_ attribute on _any_ object.

There are *conventions*, though:

- "protected" attributes: `_name`
- "private" attributes: `__name`

(But again, note that this is not *enforced* by the system in any way.)

Equality, identity

The `is` operator returns `True` if two names refer to the same instance; the == operator compares the *values* of two objects.[1]

Note that two instances may be equal in any respect yet be different instances: *equality is not identity!*

```
>>> dt4 = date(2012,9,28)
>>> dt5 = date(2012,9,28)
>>> dt4 == dt5
True
>>> dt4 is dt5
False
```

[1] A class can define how exactly the == operator should carry out the comparison.

Instance methods

```
>>> dt1.strftime('%a %d %b')
'Fri 28 Sep'
```

Invoke an instance method just like any other function.

User-defined classes, I

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

```
class MinMax(object):

  def __init__(self):
    self.min = None
    self.max = None

  def send(self, val):
    if (self.min is None) or (val < self.min):
        self.min = val
    if (self.max is None) or (val > self.max):
        self.max = val
```

Source code available at:

http://www.gc3.uzh.ch/teaching/gc3pie2012/python/minmax.py

User-defined classes, II

```
class MinMax (object) :

  def __init__(self):
    self.min = None
    self.max = None

  def send(self, val):
    if (self.min is None) or (val < self.min):
        self.min = val
    if (self.max is None) or (val > self.max):
        self.max = val
```

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, III

```
class MinMax(object):

  def __init__(self):
    self.min = None
    self.max = None

  def send(self, val):
    if (self.min is None) or (val < self.min):
        self.min = val
    if (self.max is None) or (val > self.max):
        self.max = val
```

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

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.min` to reference an attribute in this instance, `datetime.date` to mean a class defined in module `date`, etc.

Object initialization

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

```
class MinMax(object):

  def __init__(self):
    self.min = None
    self.max = None

  def send(self, val):
    if (self.min is None) or (val < self.min):
        self.min = val
    if (self.max is None) or (val > self.max):
        self.max = val
```

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 MinMax.__init__
m = MinMax()
```

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: Change the `MinMax` class so that it can be initialized with a sequence; the `.min` and `.max` attributes should be initialized to the minimum and maximum of that sequence. Example:

```
>>> m = MinMax([1,2,3])
>>> m.min == 1
True
>>> m.max == 3
True
```

Exercise B: Augment the `MinMax` class so that it computes the average of the numbers given to it via the `send` method. Store this average in the `.average` instance attribute. Call the new class `MinMaxAvg`.