

Introduction into Python

Python 5: Classes, Exceptions, Generators and more

Daniel Polani

Concept: There are three different types of name spaces:

1. built-in names (such as `abs()`)
2. global names of a module
3. local names in functions

Note: same names in different name spaces denote *different* objects, they have no relation with each other. Remember, a variable instantiated (brought to life) in a function is local to this function.

Likewise, a name defined in a given module needs to be prefixed by the module name and the `.` qualifier.

Classes: Briefest Introduction

Idea: the concept of a *class* is at the core of object-oriented programming. A class defines a 'type' of objects and operators that can be applied to this 'type' of objects. It becomes particularly powerful through the concept of inheritance which allows one to group object types according to operations which can be applied to them in a similar coherent fashion.

Classes: Example

```
Code
class DeepThought:
    "An example class"
    answer = 42           # class global variable

    def respond(self):   # always use 'self' as first method argument
        return self.answer # DeepThought.answer would also have worked
                           # as a class-global variable

dt = DeepThought()     # create an instance of DeepThought

print dt.respond()     # call the 'respond' method of
                       # DeepThought for instance dt
```

Classes: Initialization

Construction: in general, classes need to be put in a defined initial state before use. In C++, this is done with a concept called *constructors*, but in Python, the mechanism is different.

Use `__init__(self)` method to define the initializer.

More Details: introduce method such as:

```
Code
def __init__(self):
    self.thinking = 0
```

Note: if you want to set an object-*local* variable, you have to qualify it by `self`.

Classes: Initialization Example

Full Example:

```
Code
class DeepThought:
    "An example class"
    answer = 42
    def __init__(self):
        self.thinking = 0 # initialize object-local variable
    def respond(self):
        self.thinking += 1 # modify this variable, only for
                           # this object
        return self.answer # inherits from class-global

dt = DeepThought() # one object
dt2 = DeepThought() # another object of same type
print dt.thinking # object local
print dt.respond(), dt.thinking, dt2.thinking # all object-local
                                                    # variables
print dt.respond(), dt.thinking, dt2.thinking
dt.answer += 17
print dt.respond(), dt.thinking, dt2.thinking
```

```
0
42 1 0
42 2 0
59 3 0
```

Classes: Methods call Methods

```
Code
class DeepThought:
    "An example class"
    answer = 42

    def __init__(self):
        self.thinking = 0

    def think(self):
        print "think..."

    def respond(self):
        self.think() # 'respond' calls 'think'
        return self.answer

dt = DeepThought()
print dt.respond()
```

```
think...
42
```

Inheritance

Note: we can create more specific classes from more general classes through inheritance

Consider: a more general class

```
Code
class Computer:
    "The general computer class."

    def __init__(self, name):
        self.calculate = 0
        self.name = name # sets computer name to parameter

    def think(self):
        self.calculate += 1
```

Inherited Class

```
Code
class DeepThought(Computer):
    "A particular computer."
    def __init__(self):
        Computer.__init__(self, "DeepThought")
        # initialize the 'Computer' in DeepThought-type computers
        self.answer = 42

    def think(self):
        # if that was not given,
        # it would just increment
        # self.calculate (try it!)

        print "think..."
        print self.answer

dt = DeepThought()
dt.think()
print dt.calculate
print dt.name
```

```
think...
42
0 # notice that DeepThought.think has been used here!
DeepThought
```

Daniel Polani: Introduction into Python – p.9??

Destructors

Destructor: method with fixed internal name `__del__()` is called when class is being destroyed, either explicitly (via `del` operator) or just by ending the scope of the variable.

```
Code
class Computer:
    def think(self):
        print "...."

class DeepThought(Computer):
    def __init__(self):
        print "Creating DeepThought..."
        self.answer = 42

    def __del__(self):
        print "DeepThought crashed..."

dt = DeepThought() # DeepThought is generated
dt.think()

# now dt is being removed, as its life span terminates
```

```
Creating DeepThought...
....
DeepThought crashed...
```

Daniel Polani: Introduction into Python – p.10??

Pretty-Printing Classes

Use: `__repr__` method

```
Code
class DeepThought:
    def __init__(self):
        print "Creating DeepThought..."
        self.answer = 0

    def think(self):
        self.answer = 42

    def __repr__(self):
        return "|" + str(self.answer) + "|" # return a string

dt = DeepThought()
dt.think()
print dt
```

```
Creating DeepThought...
|42|
```

Daniel Polani: Introduction into Python – p.11??

Some Notes on Classes

Private Variables: are denoted by `__` prefix. This is a *convention* and there is a way of addressing these variables. Nevertheless, if a variable is private, there is probably a good reason for that, and you should not consider it editable from outside (only class methods should be manipulating it).

Class Variables: come into existence dynamically by assignment.

Daniel Polani: Introduction into Python – p.12??

pickle

pickle: module supporting the *persistent* storage of data structures

Example:

```
Code
import pickle

a = (1,2,3,[5,6,7])
pickled_a = pickle.dumps(a)
print "a=", a, "\n"
print "pickled a=", '''', pickled_a, '''', "\n"
print "unpickled a=", pickle.loads(pickled_a)
```

will yield

```
a= (1, 2, 3, [5, 6, 7])
```

```
pickled a= " (I1
I2
I3
(lp0
I5
aI6
aI7
atp1
."
```

```
unpickled a= (1, 2, 3, [5, 6, 7])
```

Daniel Polani: Introduction into Python – p.13??

Daniel Polani: Introduction into Python – p.14??

pickle II

Note: you can store the string to a file, completely storing a data structure in your program storage on harddisk to recover it at a later time.

This language feature is called *persistence*. In Java, this would be called *serialization*.

Pickling works on Classes

```
Code
import pickle

class DeepThought:
    def __init__(self):
        print "Creating DeepThought..."
        self.answer = 0
    def think(self):
        self.answer = 42
    def __repr__(self):
        return "|" + str(self.answer) + "|" # return a string

dt = DeepThought()
dt.think()
print "Before", dt

pickled_thought = pickle.dumps(dt)
unpickled_thought = pickle.loads(pickled_thought)

print "After", unpickled_thought
```

```
Creating DeepThought...
Before |42|
After |42|
```

Daniel Polani: Introduction into Python – p.15??

Daniel Polani: Introduction into Python – p.16??

Exceptions

Usage: Exceptions are classes and can be used in different ways, e.g. with a specifying string to indicate what is wrong

```
Code
raise Exception, "nonsense"
```

```
Traceback (most recent call last):
  File "<stdin>", line 44, in ?
Exception: nonsense
```

Note: a more specific exception helps in identifying the problem more narrowly

```
Code
raise ValueError, 42
```

```
Traceback (most recent call last):
  File "<stdin>", line 44, in ?
ValueError: 42
```

However: it is not a good idea to extend the existing exception class hierarchy by too much of your own fancy exceptions unless absolutely necessary.

Iterators and yield

Iterations: consistent use in different contexts

```
Code
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

General Principle: whatever the container object (i.e. list, dictionary or other, it provides an iterator created by an `iter()` function which has a `next()` method which is called on each iteration. Once finished, it will throw a `StopIteration` exception.

Iterator: Example

```
Input
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

Iterator: Implementation

```
Code
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self): # that is called, e.g. via iter(),
                       # or indirectly via a for loop
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
Input
>>> for char in Reverse('spam'):
...     print char
... 
```

m
a
p
s

Generators (yield)

Generators: a powerful way to create iterators!

Advantages: Easier to use, almost like writing a function

```
Code
def reverse(sequence):
    i = len(sequence) - 1
    while i >= 0:
        yield sequence[i]
        i -= 1

for s in reverse([1,2,3,4]):
    print s
```

```
4
3
2
1
```

Note: the iterator generated by `reverse` remembers the states it has generated for each iteration

Bottom-Line: you can separate head and body of an iteration, one of the nicest properties of Python!