A not so short introduction to Python: part III

Luca Schenato

Research Institute for Hydrogeological Protection
Italian National Research Council
(CNR-IRPI)

03/31/2011

1 Files

2 Functions

3 Modules

4 Errors and Exceptions

5 Credit

1. Files
2. Functions
3. Modules
4. Errors and Exceptions
5. Credit

Working with files: basics

File management is natively implemented in Python: this allows for a very fast and easy file management.
Common operations on text and binary files, like opening and closing, reading and writing sequences of bits, . . . , can be performed with the following built-in functions:

Operation	Description
`fileout = open('file.txt','w')`	open file.txt returning the file object fileout with write permission
`filein = open('dati','r')`	open of dati with read permission
`s = filein.read()`	read entire filein into the string s

Working with files: basics

Operation	Description
`s = filein.read(N)`	read N bytes from filein into the string s
`s = filein.readline()`	read on one line from filein into the string s (only text files)
`ls = filein.readlines()`	read entire filein into the list of strings ls (only text files)
`fileout.write(s)`	write s into entire fileout
`fileout.writelines(ls)`	write the list of string ls into fileout
`fileout.close()`	close fileout

Working with files: basics

Operation	Description
`filein.tell()`	returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file.
`filein.seek(offset, fromwhat)`	point to the byte at offset bytes with respect to the byte at position fromwhat (0[beginning of file], 1[current position] or 2[end of file])

Working with files: basics

Hint

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent try-finally blocks.

```
>>> with open('/tmp/workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

Working with files: the pickle module

`read()` method only returns strings: to deal with numbers, retrurning strings have to be passed to casting functions, like `int()`.

However, when you want to save more complex data types like lists, dictionaries, or class instances, things could get a lot more complicated, but not in Python. Python provides a standard module called pickle.

Working with files:the pickle module

Pickling and unpickling

The pyckle module can take almost any Python object, and convert it to a string representation; this process is called pickling. Reconstructing the object from the string representation is called unpickling.

Operation	Description
pickle.dump(x, f)	pickle the object x into file f
x = pickle.load(f)	unpickle the object x from file f

1 Files

2 Functions

3 Modules

4 Errors and Exceptions

5 Credit

Functions

What are they?

Functions are essentially groups of instructions and statements, with optional input arguments (i.e. parameters) and optional output arguments.

Functions are useful because:

- allow for multiple usage of the same code;
- allow for a clear arrangement of the code and make the programming easier.

The syntax follows:

```
def function_name([list of parameters, divide by comma]):
body
return output_parameters # optional
```

Functions

Once defined, a function can be invoked easily, by digiting its name, followed by the list of the optional parameters.
As an example, look at the following function definitions:.

```
>>> # Fibonacci numbers module
... def fib(n):     # write Fibonacci series up to n
>>>     a, b = 0, 1
>>>     while b < n:
>>>        print b
>>>        a, b = b, a+b
...
>>> def fib2(n): # return Fibonacci series up to n
>>>     result = []
>>>     a, b = 0, 1
>>>     while b < n:
>>>         result.append(b)
>>>         a, b = b, a+b
>>>     return result
...
```

Functions

To call them, just invoke function name

```
>>> fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Variables passing

Please note that in Python, the variables are passed by value, never by reference: therefore, any changes to the parameter that take place inside the function have no affect on the original data stored in the variable. (This is rigorously true only for non mutable variables.)

Variables are first searched in the local namespace (inside the scope of the function) and if not found they are searched in the global namespace.

Functions

Look at the following examples:

```
>>> x = 10
>>> y = 20
>>> def f1():
...     x  = 0
...     print x
...     print y
...
>>> x
10
>>> y
20
>>> f1()
0
20
>>> x
10
>>> y
20
```

Functions

Look at the following examples:

```
>>> x = [1,2,3]
>>> def f2(x):
...     x.append(4)
...
>>> x
[1, 2, 3]
>>> f2(x)
>>> x
[1, 2, 3, 4]
>>> def f3(y):
...     x = [4,5,6]
...     x.append(y)
...     return x
...
>>> f3(6)
[4, 5, 6, 6]
>>> x
[1, 2, 3, 4]
```

Functions

Optional parameters can be defined in Python's function, as well: they assume a given value, if not specified otherwise:

```
>>> def f4(a,b = 1):
...     print a, b
...
>>> x = 1000
>>> y = 2000
>>> f4(x)
1000 1
>>> f4(x,y)
1000 2000
```

1. Files
2. Functions
3. **Modules**
4. Errors and Exceptions
5. Credit

Modules vs. scripts

When you quit the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost.

A script

A way to have your code at disposal whenever you want is by writing the input for the interpreter into a file and running it with that file as input instead. This is known as creating a script.

Modules vs. scripts

As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

A module

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module.

Definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

Modules

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`.

As an exaple, put the following statements in a file and call it fibo.py:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Modules

This module can be imported with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Modules

A variant of `import` allows for loading names from a module directly into the importing module's symbol table:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`).

Modules

Calling a module in a particular path

```
>>> import sys
>>> sys.path.append('particular path')
>>> import modulename
```

Very Important!!!

For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use `reload()`, e.g. `reload(modulename)`.

Modules as scripts

When you run a Python module with python fibo.py <arguments>: in this case the code in the module will be executed, just as if you imported it, but with the __name__ set to "__main__". This allows for an easy workaround to use the module as a script to launch a function of the module itself; try to add the following code at the end of the file fibo.py:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

By this, you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the "main" file:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Modules and namepace

Each module has its own set of names (variables and functions): this is known as namespace. By the `import` we can import the names between different namespaces. By the command `dir()` the list of names inside the namespace is shown:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
```

those are the names loaded by the default namespace. When some modules are imported additional names are loaded:

```
>>> from fibo import *
>>> dir()
['__builtins__', '__doc__', '__name__', 'fib', 'fib2']
```

. . . namespaces apply also to function. . . try to `print dir()` inside the scope of a function.

1 Files

2 Functions

3 Modules

4 Errors and Exceptions

5 Credit

Errors and Exceptions

We can divide errors in two main classes

- syntax errors;
- exceptions.

Syntax Errors

Syntax errors, aka parsing errors, are the most common kind of complaint you get while you are still learning Python (and even after):

```
>>> while True print 'Hello world'
  File "<stdin>", line 1, in ?
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little "arrow" pointing at the earliest point in the line where the error was detected. File name and line number are printed so you know where to look in case the input came from a script.

Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal, but most of them are not handled by programs. Here is an example:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message.

Handling Exceptions

It is possible to handle exceptions with the following control flow:

```
try:
codes to be controlled
except [exception1 to be handled]:
codes to be executed in case of error
except [exception2 to be handled]:
codes to be executed in case of error
[else:]
codes to be executed in case of non error
[finally:]
codes to be executed in any case, even if other
     exceptions are raised
```

Handling Exceptions

Here is an example:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and
     'str'
```

Credit

Credit goes to www.python.org and herein contents.