Scientific Programming

Lecture A05 - Designing programs

Andrea Passerini

Università degli Studi di Trento

2019/10/22

Acknowledgments: Alberto Montresor, Stefano Teso

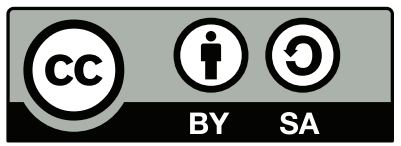


Table of contents

Definitions

Module

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended.

Package

A package is a collection of multiple modules and potentially other packages.

Python Standard Library

Python Standard Library

The Python Standard Library is installed by default together with Python 2 and 3. It provides a lot of packages for dealing with many different tasks.

`https://docs.python.org/2/library/`

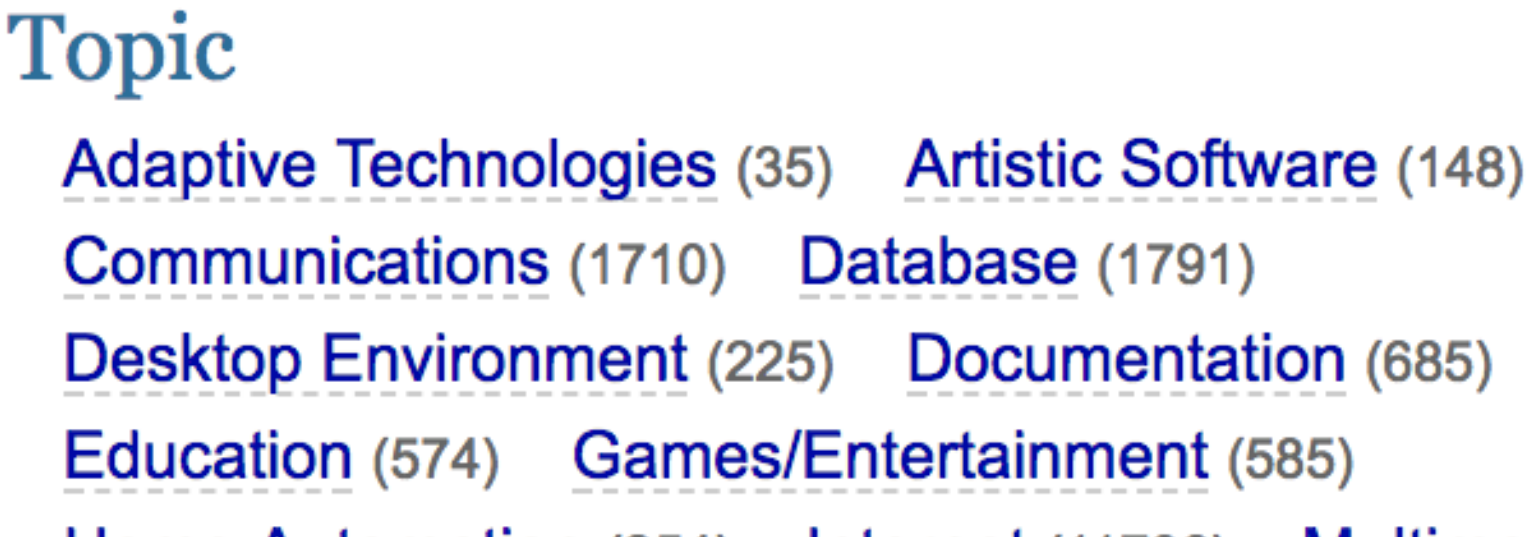
9. Numeric and Mathematical Modules
   - 9.1. `numbers` — Numeric abstract base classes
   - 9.2. `math` — Mathematical functions
   - 9.3. `cmath` — Mathematical functions for complex numbers
   - 9.4. `decimal` — Decimal fixed point and floating point arithmetic
   - 9.5. `fractions` — Rational numbers
   - 9.6. `random` — Generate pseudo-random numbers
   - 9.7. `itertools` — Functions creating iterators for efficient looping
   - 9.8. `functools` — Higher-order functions and operations on callable objects

Python Package Index

Python Package Index

The Python Package Index is a repository of software for the Python programming language, containing more than 100k packages. The packages and modules that are not included in the standard library need to be installed. More on that in the lab lessons.

`https://pypi.python.org/pypi`



Importing a module/package

In order to make use of a package, you have to first import it:

```
import numpy
```

Once imported, you can use its definitions (functions, variables, etc.) by prefixing them with the name of the module and a dot `.`

```
print(numpy.arccos(0))
```

If you try to import a package and get an error, it means that the module is not installed in your machine.

```
import iamnotinstalled
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError:  No module named iamnotinstalled
```

Importing a module/package

Sub-modules

You can import specific sub-modules using the notation `import module.submodule`; then, you can call functions included in it by prefixing it with `module.submodule`

```
import numpy
import numpy.linalg
A = numpy.matrix([[1,2], [3,4]])
print(numpy.linalg.eig(A))

(array([-0.37228132,  5.37228132]),
 matrix([[-0.82456484, -0.41597356],
         [ 0.56576746, -0.90937671]]))
```

Importing a module/package

Abbreviations

You can also abbreviate the name of the package with a shorthand, as follows:

```
import numpy as np
import numpy.linalg as la
A = np.matrix([[1,2], [3,4]])
print(la.eig(A))
```

Importing individual functions

Abbreviations

You can also import individual functions, as follows.

```
from numpy import arccos, arcsin
print(arccos(0))
print(arcsin(0))
```

```
1.57079632679
0.0
```

Importing all the functions functions

Abbreviations

You can also import all individual functions, as follows.

```
from math import *
print(factorial(5))
print(floor(3.45))
print(ceil(3.45))
print(sqrt(16))
print(pi)
```

```
120
3
4
4.0
3.141592653589793
```

Some comments

- `import package [as alias]`: reads the file `package.py` all attributes and inserts them in the namespace `package` (or `alias`, if present)
- `from package import attribute`: imports (some) attributes from file `package.py` and insert them in the current namespace
- When using `from`, you may have overlapping between attribute names. The last to be imported wins!

`__future__` module

The `__future__` "module" is a special module used to import Python 3 functionality into Python 2 programs. It can be useful for writing code compatible with both Python 2 and Python 3.

```
# Python 2.7
from __future__ import print_function
from __future__ import division
print(2/3)
```

```
0.666666666667
```

Defining modules

Writing (basic) Python modules is very simple. To create a module of your own, simply create a new `.py` file with the module name, and then import it using the Python file name (without the `.py` extension) using the `import` command.

Notes

- Each module has its own global scope
- The global scope spans a single file only

Table of contents

Input and output

Reading and Writing Text Files

In order to access the contents of a file (let's assume a text file for simplicity), we need to first create a handle to it. This can be done with the open() function.

Handle

A handle is simply an object that refers to a given file. It does not contain any of the file data, but it can be used together with other methods, to read and write from the file it refers to.

Built-in functions and methods

Result	Built-in function	Meaning
`file`	`open(str, [str])`	Get a handle to a file

Result	Method	Meaning
`str`	`file.read()`	Read all the file as a single string
`list of str`	`file.readlines()`	Read all lines of the file as a list of strings
`str`	`file.readline()`	Read one line of the file as a string
`None`	`file.write(str)`	Write one string to the file
`None`	`file.close()`	`Close the file` (i.e. flushes changes to disk)

Opening files

`open` function arguments

- The first argument to `open()` is the path of the file to be open
- The second argument is optional. It tells `open()` how we intend to use the file: for reading, for writing, etc.

Access modes

- `"r"`: we want to read from the file. This is the default mode.
- `"w"`: we want to write to the file, overwriting its contents.
- `"a"`: we want to append to the existing contents.
- `"b"`: the file will be read in binary mode

Opening files

Mode flags can be combined

- `"rw"`: we want to read and write (in "overwrite" mode).
- `"ra"`: we want to read and write (in "append" mode).

Opening files

Opening a file returns a specific object type.

```
f = open("data/table.csv", "r")
print(type(f))
print(f)
```

```
<class '\_io.TextIOWrapper'>
<\_io.TextIOWrapper name='data/table.csv' mode='r'
  encoding='US-ASCII'>
```

Closing files

Once you are done with a file (either reading or writing), make sure to call the `close()` method to finalize your operations.

```
file.close()
```

Once the file is closed, you cannot read or write on it anymore.

```
print(file.readlines())
```

```
Traceback (most recent call last):
  File "data/table.csv", line 3, in <module>
    print(file.readlines())
ValueError:  I/O operation on closed file.
```

Opening and closing

```
with open("data") as f:
    print(f.readlines())
```

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.

Reading files – Method 1

As a single, big string

```
contents = file.read()
print(contents)
```

```
surname,name,email address
passerini,andrea,andrea.passerini@unitn.it
bianco,luca,luca.bianco@fmach.it
leoni,david,david.leoni@unitn.it
```

`read()` makes sense if your file is small enough (i.e. it fits into the RAM) and it is not structured as a sequence of lines separated by newline characters.

Reading files – Method 2

As a list of lines (represented as strings)

```
lines = f.readlines()
print(lines)
```

```
['surname,name,email address\n',
 'passerini,andrea,andrea.passerini@unitn.it\n',
 'bianco,luca,luca.bianco@fmach.it\n',
 'leoni,david,david.leoni@unitn.it\n']
```

`readlines()` makes sense if your file is small enough and it is structured as a collection of lines.

Reading files – Method 3

One line at a time, sequentially, from the first onwards, using method `readline()`

```
f = open("table.csv")
line = f.readline()          # skip first line
line = f.readline()
while (line != ""):
    print(line, end="")
    line = f.readline()
```

```
passerini,andrea,andrea.passerini@unitn.it
bianco,luca,luca.bianco@fmach.it
leoni,david,david.leoni@unitn.it
```

`readline()` makes sense for very large files, because you can read one line at a time, without saturating the machine.

Reading files – Method 4

Using the iterator associated with the file object

```
f = open("table.csv")
line = f.readline()          # skip first line
for line in f:
    print(line, end="")
```

```
passerini,andrea,andrea.passerini@unitn.it
bianco,luca,luca.bianco@fmach.it
leoni,david,david.leoni@unitn.it
```

This approach makes sense for very large files, because you can read one line at a time, without saturating the machine.

Reading through the file as a stream

Warning

- Internally, Python keeps track of which lines of a file have already been read.
- Once a line has been read, it can not be read from the same file handle.
- This limitation affects all four methods.

```
f = open("table.csv")
lines = f.readlines()          # read entire file
for line in f:
    print(line, end="")
```

This code does not print anything.

Writing files

```
# Open a file for writing
f = open("result.txt", "w")

# TODO: write a complex calculation whose result is 42
result = 42

# Convert the result into a string, write a newline
f.write(str(result))
f.write("\n")

# Make sure that our writes are written to disk.
f.close()
```

Writing files

- Forgetting to close a file opened in read-only mode is not too harmful (you may exceed the maximum number of open files)
- Forgetting to close files opened in write mode can have serious consequences

Why?

Writes to files are not immediately written to disk, for efficiency. Instead, they are stored in memory until Python decides to flush them. `close()` is a way to tell Python to flush the changes.

Intuitively, this means that if you don't call `close()` and the program quits (because of an error, for instance), then your changes are not written to the file.

Some fancy ways to format strings

```
print('{0} and {1}'.format('spam', 'eggs'))
print('{1} and {0}'.format('spam', 'eggs'))
print('This {food} is {adjective}.'.format(
    food='spam', adjective='absolutely horrible'))
print('The value of PI is about {0:.3f}.'.format(3.14159265))
```

```
spam and eggs
eggs and spam
This spam is absolutely horrible.
The value of PI is approximately 3.142.
```

Some fancy ways to format strings

```
for x in range(1, 11):
    print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
```

```
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Exercise

```
f1 = open("result.txt", "w")
f1.write("Text 1\n")
f2 = open("result.txt", "w")
f2.write("Text 2\n")
f2.close()
f1.close()
```

Exercise

```
f1 = open("result.txt", "w")
f1.write("Text 1\n")
f2 = open("result.txt", "w")
f2.write("Text 2\n")
f2.close()
f1.close()
```

```
Text 1
```

Exercise

```
f1 = open("result.txt", "w")
f1.write("Text 1\n")
f2 = open("result.txt", "a")
f2.write("Text 2\n")
f2.close()
f1.close()
```

Exercise

```
f1 = open("result.txt", "w")
f1.write("Text 1\n")
f2 = open("result.txt", "a")
f2.write("Text 2\n")
f2.close()
f1.close()
```

```
Text 1
```

Exercise

```
f1 = open("result.txt", "w")
f1.write("Text 1\n")
f1.close()
f2 = open("result.txt", "a")
f2.write("Text 2\n")
f2.close()
```

Exercise

```
f1 = open("result.txt", "w")
f1.write("Text 1\n")
f1.close()
f2 = open("result.txt", "a")
f2.write("Text 2\n")
f2.close()
```

```
Text 1
Text 2
```