Brief summary of basic Python syntax

Joakim Sundnes[1,2]   Hans Petter Langtangen[1,2]   Ola Skavhaug[3]

Center for Biomedical Computing, Simula Research Laboratory[1]

Dept. of Informatics, University of Oslo[2]

mCASH (formerly Dept. of Informatics, University of Oslo)[3]

Aug 25, 2014

Contents

- Python variables; numbers, strings, lists++
- Simple control structures
- Functions
- Reading and writing files
- Using and writing modules
- Doc strings

Sources for more information and documentation

- H.P. Langtangen and G. K. Sandve: Illustrating Python via Bioinformatics Examples: PDF or HTML
- pydoc anymodule, pydoc anymodule.anyfunc
- Python Library Reference (go to *index* in the upper right corner)
- Python 2.7 Quick Reference
- Python Global Module Index
- Think Python (textbook)
- Dive Into Python (texbook)
- Think Like a Computer Scientist (textbook)
- Unix Need-to-know
- Emacs Need-to-know

Video material

- A Gentle Introduction to Programming Using Python
- Introduction to Computer Science and Programming
- Learning Python Programming Language Through Video Lectures
- Python Programming Tutorials Video Lecture Course
- Python Videos, Tutorials and Screencasts

First Python encounter: a scientific hello world program

```python
#!/usr/bin/env python

from math import sin
import sys

x = float(sys.argv[1])
print "Hello world, sin({0}) = {1}".format(x, sin(x))
```

Running the script from the command line

Code in file `hw.py`.

Run with command:

```
> python hw.py 0.5
Hello world, sin(0.5) = 0.479426.
```

Linux alternative if file is executable (`chmod a+x hw.py`):

```
> ./hw.py 0.5
Hello world, sin(0.5) = 0.479426.
```

Interactive Python & IPython

- Typing `python` gives you an interactive Python shell
- IPython is better, can also run scripts:
  `In [1]: run hw.py 3.14159`
- IPython is integrated with Python's pdb debugger
- pdb can be automatically invoked when an exception occurs
- IPython supports tab completion, additional help commands, and much more, …

Dissection of `hw.py` (1)

On Unix: find out what kind of script language (interpreter) to use:

```
#!/usr/bin/env python
```

Access library functionality like the function `sin` and the list `sys.arg` (of command-line arguments):

```
from math import sin
import sys
```

Read 1st command line argument and convert it to a floating point object:

```
x = float(sys.argv[1])
```

Dissection of `hw.py` (2)

Print out the result using a format string:

```
print "Hello world, sin({0}) = {1}".format(x, sin(x))    # v2.x
print("Hello world, sin({0}) = {1}".format(x, sin(x)))   # v3.x
```

or with complete control of the formating of floats (printf syntax):

```
print "Hello world, sin({x:g}) = {s:.3f}".format(x=x, s=sin(x))
print("Hello world, sin({x:g}) = {s:.3f}".format(x=x, s=sin(x)))
```

Python variables

Variables are not declared

Variables hold references to objects

```
a = 3         # ref to an int object containing 3
a = 3.0       # ref to a float object containing 3.0
a = '3.'      # ref to a string object containing '3.'
a = ['1', 2]  # ref to a list object containing
              # a string '1' and an integer 2
```

Test for a variable's type:

```
if isinstance(a, int): # int?
if isinstance(a, (list, tuple)): # list or tuple?
```

Common types

- Numbers: `int`, `float`, `complex`
- Sequences: `str`, `list`, `tuple`, `ndarray`
- Mappings: `dict` (dictionary/hash)
- User-defined type (via user-defined class)

Simple Assignments

```
a = 10           # a is a variable referencing an
                 # integer object of value 10

b = True         # b is a boolean variable

a = b            # a is now a boolean as well
                 # (referencing the same object as b)

b = increment(4) # b is the value returned by a function

is_equal = a == b # is_equal is True if a == b
```

Lists and tuples

```python
mylist  = ['a string', 2.5, 6, 'another string']
mytuple = ('a string', 2.5, 6, 'another string')
mylist[1]  = -10
mylist.append('a third string')
mytuple[1] = -10  # illegal: cannot change a tuple
```

A tuple is a constant list (known as an *immutable* object, contrary to *mutable* objects which can change their content)

List functionality

Construction	Meaning
a = []	initialize an empty list
a = [1, 4.4, 'run.py']	initialize a list
a.append(elem)	add elem object to the end
a + [1,3]	add two lists
a.insert(i, e)	insert element e before index i
a[3]	index a list element
a[-1]	get last list element
a[1:3]	slice: copy data to sublist (here: index 1, 2)
del a[3]	delete an element (index 3)
a.remove(e)	remove an element with value e
a.index('run.py')	find index corresponding to an element's value
'run.py' in a	test if a value is contained in the list
a.count(v)	count how many elements that have the value v
len(a)	number of elements in list a
min(a)	the smallest element in a
max(a)	the largest element in a
sum(a)	add all elements in a
sorted(a)	return sorted version of list a
reversed(a)	return reversed sorted version of list a
b[3][0][2]	nested list indexing
isinstance(a, list)	is True if a is a list
type(a) is list	is True if a is a list

Dictionary functionality

Construction	Meaning
a = {}	initialize an empty dictionary
a = {'point': [0,0.1], 'value': 7}	initialize a dictionary
a = dict(point=[2,7], value=3)	initialize a dictionary w/string keys
a.update(b)	add key-value pairs from b in a
a.update(key1=value1, key2=value2)	add key-value pairs in a
a['hide'] = True	add new key-value pair to a
a['point']	get value corresponding to key point
for key in a:	loop over keys in unknown order
for key in sorted(a):	loop over keys in alphabetic order
'value' in a	True if string value is a key in a
del a['point']	delete a key-value pair from a
list(a.keys())	list of keys
list(a.values())	list of values
len(a)	number of key-value pairs in a
isinstance(a, dict)	is True if a is a dictionary

String operations

```python
s = 'Berlin: 18.4 C at 4 pm'
s[8:17]            # extract substring
':' in s           # is ':' contained in s?
s.find(':')        # index where first ':' is found
s.split(':')       # split into substrings
s.split()          # split wrt whitespace
'Berlin' in s      # test if substring is in s
s.replace('18.4', '20')
s.lower()          # lower case letters only
s.upper()          # upper case letters only
s.split()[4].isdigit()
s.strip()          # remove leading/trailing blanks
', '.join(list_of_words)
```

Strings in Python use single or double quotes, or triple single/double quotes

Single- and double-quoted strings work in the same way:
'some string' is equivalent to "some string"

Triple-quoted strings can be multi line with embedded newlines:

```python
text = """large portions of a text
can be conveniently placed inside
triple-quoted strings (newlines
are preserved)"""
```

Raw strings, where backslash is backslash:

```python
s3 = r'\(\s+\.\d+\)'
# in an ordinary string one must quote backslash:
s3 = '\\(\\s+\\.\\d+\\)'
```

Simple control structures

Loops:

```python
while condition:
    <block of statements>
```

Here, condition must be a boolean expression (or have a boolean interpretation), for example: i < 10 or !found

```python
for element in somelist:
    <block of statements>
```

Conditionals/branching:

```python
if condition:
    <block of statements>
elif condition:
    <block of statements>
else:
    <block of statements>
```

Looping over integer indices is done with `range`

```python
for i in range(10):
    print(i)
```

Remark:

range in Python 3.x is equal to xrange in Python 2.x and generates an *iterator* over integers, while range in Python 2.x returns a list of integers.

Examples on loops and branching

```python
x = 0
dx = 1.0
while x < 6:
    if x < 2:
        x += dx
    elif 2 <= x < 4:
        x += 2*dx
    else:
        x += 3*dx
    print 'new x:', x
print 'loop is over'
```

(Visualize execution)

```python
mylist = [0, 0.5, 1, 2, 4, 10]
for i, x in enumerate(mylist):
    print i, x
print 'loop is over'
```

(Visualize execution)

Functions and arguments

User-defined functions:

```python
def split(string, char):
    position = string.find(char)
    if  position > 0:
        return string[:position+1], string[position+1:]
    else:
        return string, ''

# function call:
message = 'Heisann'
print(split(message, 'i'))
# prints ('Hei', 'sann')
```

Positional arguments must appear before keyword arguments:

```python
def split(message, char='i'):
    # ...
```

`eval` and `exec` turn strings into live code

Evaluating string expressions with `eval`:

```python
>>> x = 20
>>> r = eval('x + 1.1')
>>> r
21.1
>>> type(r)
<type 'float'>
```

Executing strings with Python code, using `exec`:

```python
import sys
user_expression = sys.argv[1]

# Wrap user_expression in a Python function
# (assuming the expression involves x)

exec("""
def f(x):
    return %s
""" % user_expression)

# or

f = eval('lambda x: %s' % user_expression)
```

File reading

Reading a file:

```python
infile = open(filename, 'r')
for line in infile:
    # process line

lines = infile.readlines()
for line in lines:
    # process line

for i in xrange(len(lines)):
    # process lines[i] and perhaps next line lines[i+1]

fstr = infile.read()    # fstr contains the entire file
fstr = fstr.replace('some string', 'another string')
for piece in fstr.split(';'):
    # process piece (separated by ;)

infile.close()
```

File writing

```python
outfile = open(filename, 'w')    # new file or overwrite
outfile = open(filename, 'a')    # append to existing file

outfile.write("""Some string
....
""")
outfile.writelines(list_of_lines)

outfile.close()
```

Using modules

Import module:

```python
import sys
x = float(sys.argv[1])
```

Import module member `argv` into current namespace:

```python
from sys import argv
x = float(argv[1])
```

Import everything from sys (not recommended)

```python
from sys import *
x = float(argv[1])

flags = ''
# Ooops, flags was also imported from sys, this new flags
# name overwrites sys.flags!
```

Import argv under an alias:

```python
from sys import argv as a
x = float(a[1])
```

Making your own Python modules

- Reuse scripts by wrapping them in classes or functions
- Collect classes and functions in library modules
- How? just put classes and functions in a file `MyMod.py`
- Put `MyMod.py` in one of the directories where Python can find it (see next slide)

Examples:

```python
import MyMod
# or
import MyMod as M    # M is a short form
# or
from MyMod import *
# or
from MyMod import myspecialfunction, myotherspecialfunction
```

How Python can find your modules?

Python has some "official" module directories, typically

```
/usr/lib/python2.7
/usr/lib/python2.7/site-packages
/usr/lib/python3.4
/usr/lib/python3.4/site-packages
```

+ current working directory

The environment variable `PYTHONPATH` may contain additional directories with modules

```
> echo $PYTHONPATH
/home/me/python/mymodules:/usr/lib/python3.4:/home/you/yourlibs
```

Python's `sys.path` list contains the directories where Python searches for modules, and `sys.path` contains "official" directories, plus those in PYTHONPATH

Search path for modules can be set in the script

Add module path(s) directly to the `sys.path` list:

```python
import sys, os

sys.path.insert(
    0, os.path.join(os.environ['HOME'], 'python', 'lib')

...

import MyMod
```

Packages

- A class of modules can be collected in a *package*
- Normally, a package is organized as module files in a directory tree
- Each subdirectory has a file `__init__` (can be empty)
- Documentation: Section 6 in the Python Tutorial

Example tree:

```
MyMod
    __init__.py
    numerics
        __init__.py
        pde
            __init__.py
            grids.py     # contains fdm_grids object
```

Can import modules in the tree like this:

```python
from MyMod.numerics.pde.grids import fdm_grids

grid = fdm_grids()
grid.domain(xmin=0, xmax=1, ymin=0, ymax=1)
...
```

Test block in a module

Module files can have a test/demo section at the end:

```python
if __name__ == '__main__':
    infile = sys.argv[1]; outfile = sys.argv[2]
    for i in sys.argv[3:]:
        create(infile, outfile, i)
```

- The block is executed *only if* the module file is run as a program
- The tests at the end of a module often serve as good examples on the usage of the module

Public/non-public module variables

Python convention: add a leading underscore to non-public functions and (module) variables

```
_counter = 0

def _filename():
    """Generate a random filename."""
    ...
```

After a `from MyMod import *` the names with leading underscore are *not* available:

```
>>> from MyMod import *
>>> _counter
NameError: name '_counter' is not defined
>>> _filename()
NameError: name '_filename' is not defined
```

But with `import MyMod` and `MyMod.` prefix we get access to the non-public variables:

```
>>> import MyMod
>>> MyMod._counter
4
>>> MyMod._filename()
'temp_756241.dat'
>>> MyMod._counter = 42
```

Installing modules

- Python has its own tool, Distutils, for distributing and installing modules
- Installation is based on the script `setup.py`

Standard command for installing Python software with `setup.py` file:

```
> sudo python setup.py install
```

Notice.

If your package contains Python modules and extension modules written in C, the latter will be automatically compiled (using the same compiler and options as used for Python itself when it was installed).

Controlling the installation destination

`setup.py` has many options, see the Installing Python modules

Install in some user-chosen local directory (no need for sudo now):

```
> python setup.py install --prefix=$HOME/install
# copies modules  to $HOME/install/lib/python2.7/site-packages
# copies programs to $HOME/install/bin
```

Make sure that

- `$HOME/install/lib/python2.7/site-packages` is in your PYTHONPATH
- `$HOME/install/bin` is in your PATH

In `~/.bashrc`:

```
export PATH=$HOME/install/bin:$PATH
export PYTHONPATH=$HOME/install/lib/python2.7/site-packages:$PYTHONPAT
```

Writing your own `setup.py` script

Suppose you have a module in `mymod.py` that you want to distribute to others such that they can easily install it by `setup.py install`.

```
from distutils.core import setup
name='mymod'

setup(name=name,
      version='0.1',
      py_modules=[name],         # modules to be installed
      scripts=[name + '.py'],    # programs to be installed
      )
```

Now, `setup.py` will be installed both as a module and as an executable script (if it has a test block for sensible code).

Can easily be extended to install a package of modules, see the introduction to Distutils

Use doc strings in functions, classes, and modules!

Doc strings = first string in a function, class, or file (module)

```
def ignorecase_sort(a, b):
    """Compare strings a and b, ignoring case."""
    return cmp(a.lower(), b.lower())
```

Doc strings in modules are a (often long multi-line) string starting in the top of the file

```
"""
This module is a fake module
for exemplifying multi-line
doc strings.
"""

import sys
import collections

def somefunc():
    ...
```

Doc strings serve many purposes

- Documentation in the source code
- Online documentation
  (Sphinx can automatically produce manuals with doc strings)
- Balloon help in sophisticated GUIs (e.g., IDLE)
- Automatic testing with the doctest module