

Python programming — introduction to Python

Finn Årup Nielsen

DTU Compute
Technical University of Denmark

September 9, 2013

Content

Invoking Python

A basic program

Datatypes, sequences, control structures, functions, object-orientation

File processing, Exception, Generators

Libraries

Documentation, Testing, checking coding style with pylint

Invoking python . . .

From the command line with no argument and interactive:

```
$ python  
>>> 1+1
```

With the file `mypythonscript.py` with the following content

```
print(1+1)
```

From the command line with a python function:

```
$ python mypythonscript.py
```

From the command line with a python function:

```
$ python  
>>> import mypythonscript
```

... Invoking python ...

With a shell-like program `myscript`

```
#!/usr/bin/python  
print(1+1)
```

Executing the script as a standard (UNIX) program

```
$ chmod u+x myscript  
$ ./myscript
```

Or execute it from within Python

```
>>> import os  
>>> os.system('myscript')
```

... Invoking python ...

Construct a string with the Python code for execution

```
s = 'a = 1+1; print(a)'\nexec(s)
```

and evaluation

```
s = '1+1'\na = eval(s)\nprint(a)
```

or a script

```
execfile('myscript.py')
```

... Invoking python

`mymodule.py` with the following content

```
def myfunction():  
    print(1+1)  
def myotherfunction():  
    print(2+2)
```

Load the library and call the functions in the library:

```
$ python  
>>> import mymodule  
>>> mymodule.myfunction()  
>>> mymodule.myotherfunction()
```

Invoking python: IPython

“An Enhanced Interactive Python” with automatic completion and some more help.

```
$ ipython
```

```
In [1]: a = 1 + 1
```

```
In [2]: ?a
```

```
Type:          int
```

```
Base Class:    <type 'int'>
```

```
String Form:   2
```

```
Namespace:    Interactive
```

```
Docstring:
```

```
    int(x[, base]) -> integer
```

A python program

```
import math, sys                # Importing modules.

def formatresult(res):          # Define function. Remember colon!
    """This is the documentation for a function."""
    return "The result is %f" % res # Percentage for formatting

if len(sys.argv) < 3:           # Conditionals should be indented
    print("Too few input argument")
elif len(sys.argv) > 10:        # Not 'elsif' or 'elseif'
    print("Too many input argument")
else:
    res = 0;                    # Semicolon not necessary. Considered bad style
    for n in range(1, len(sys.argv)): # Not first element in loop
        try: res += float(sys.argv[n]) # One-liner: no indentation
        except: pass # One-liner!
    print(formatresult(res))
```


Print

The print function can print almost anything:

```
print(math)                # An imported module
print(sys.argv)            # Some variable from a module
print(range(1, len(sys.argv))) # A result from a call
print("""
Here goes
some text
""")
```

It is possible to call print as a statement:

```
print math
print "Hello"
```

However, in Python 3.0 it is no longer allowed.

Examining the Python program

What does the `len()` function returns?

What does the `range()` function returns?

What is in `sys.argv`?

Code available at:

<https://gist.github.com/1194493>

Data types: Simple types

None: `None==None` is true, `None==False` is false, `not(None)` is true!

Boolean (from Python 2.2.1): All true: `True`, `False==False`, `bool(42)`, `not(42==True)`, `(True or False)` and `not(False)`, `True==1`, `not(1.000001 == 1)`, `1.0000000000000000000000000000001 == 1`

Integer: `32`, `int('42')`, `1/3` (yes, still integer!), `int(True)`, `int(3.14)`

Long: `1231980985476123891320918203981230123`, `long(1)`, `5L` (only available in Python 2, Python 3 just uses `int`)

Float: `4.`, `4.0`, `float(42)`, `1.0/3`, `complex(1j).real`, `complex(1j).imag`, `4.2e3`, `4.2E+3`, `3**-1`, `float('nan')`, `float('inf')` (Python 2.4/2.5 issue)

Complex: `complex(3)`, `complex(1j)`, `complex('4+j')`, `complex(1,2)`, `1+1j`

Things that do NOT work...

```
double(3.14)
single(3.14)
int('3.14')
int(None)
import math; math.sqrt(-1)      # import cmath; cmath.sqrt(-1)
1+j                              # use 1+1j..
3+'4'
float(0)/float(0)                # Not 'not a number'

float(1/3)                       # is not 0.3333. float(1./3) is
                                # (in Python 2)
```

... and a hack (for Python 2)

```
from __future__ import division
1/3                               # result is a float
```

Data types: Sequences

String: `'A string', "Another", 'DTU', str(32), str(True), "Escape \" quotation", 'Wo' + 'rd', 'Hm' + 'm'*10`
`'''A ' is not necessary''', """Multiline string
A newline""", 'The results are %.02f and %d' % (3.14159, 5), repr(42), '42', repr('42')`

List: `[1, 2, 3], ['heterogeneous', 3], ['w', 'o', 'r', 'd'], list("word"), [['list'], ['of', 'lists']], list(('a', 1))`

Tuple: `(1, 2), ('a', 1), ('remember comma',) tuple(['a', 1])`

(xrange): `xrange(2), xrange(2, 4), xrange(0, 10, 2)` (only Python 2)

Indexing with and function for sequences

# Python	Result	Matlab	
<code>a = [5, 6, 7, 8]</code>	#	<code>a = [5 6 7 8]</code>	
<code>a[0]</code>	# 5	<code>a(1)</code>	First element
<code>a[2:4]</code>	# [7, 8]	<code>a(3:4)</code>	Third and fourth
<code>a[-1]</code>	# 8	<code>a(end)</code>	Last element
<code>a[-2]</code>	# 7	<code>a(end-1)</code>	Second last
<code>a[2:]</code>	# [7, 8]	<code>a(3:end)</code>	From third element
<code>a[::2]</code>	# [5, 7]	<code>a(1:2:end)</code>	Every second element
<code>a[::-1]</code>	# [8, 7, 6, 5]	<code>a(end:-1:1)</code>	Reverse
<code>len(a)</code>	# 4	<code>length(a)</code>	Length
<code>[min(a), max(a)]</code>	# [5, 8]	<code>[min(a) max(a)]</code>	Extreme elements

It also works for other sequences, such as

```
a = 'Der kom en soldat'[0:4]
a = (5, 6, 7, 8)
a = [{1: 2}, [3, 4], 5, 'Six']
```

Functions for lists and string

Some of the functions for lists (modifies a!):

```
a = [5, 6, 7, 8]
a.pop()           # a = [5, 6, 7]
a.append(2)       # a = [5, 6, 7, 2]
a.sort()          # a = [2, 5, 6, 7]
a.reverse()       # a = [7, 6, 5, 2]
```

Some of the functions for strings (leaves a unchanged):

```
a = 'Der kom en soldat'
a.split()         # ['Der', 'kom', 'en', 'soldat']
a.upper()         # 'DER KOM EN SOLDAT'
a.title()         # 'Der Kom En Soldat'
import string
string.join(a.split(), '-') # 'Der-kom-en-soldat'
```

Lists and copy . . .

With

```
a = 'Der kom en soldat'
```

write this in one line:

```
import string  
string.join(a.split(), '-')          # 'Der-kom-en-soldat'
```


Lists and copy . . .

With

```
a = 'Der kom en soldat'
```

write this in one line:

```
import string
string.join(a.split(), '-')          # 'Der-kom-en-soldat'

'-'.join(a.split())
```

Lists and copy . . .

```
a = [1, 2, 3]
```

```
b = a
```

```
a[1] = 2000
```

```
b
```

What happens here? What is b?

Lists and copy . . .

```
a = [1, 2, 3]
```

```
b = a
```

```
a[1] = 2000
```

```
b
```

What happens here? What is b?

```
b = [1, 2, 3]
```

```
b = [2000, 2, 3]
```

```
b = [1, 2000, 3]
```

How do we solve it?

... List and copy ...

Google: Python + lists + copy

Multiple ways:

```
a = [1, 2, 3]
```

```
b = a[:]
```

```
b = list(a)
```

```
b = []; b.extend(a)
```

```
b = [];
```

```
for n in range(len(a)):
```

```
    b.append(a[n])
```

```
b = [ e for e in a ]
```

```
a[1] = 2000
```

```
b
```

And with an list of lists:

```
a = [1, 2, 3]           # A list
aa = []                # A list of lists
aa.append(a)
a[1] = 2000
aa.append(a)
```

Is there a problem?

And with an list of lists:

```
a = [1, 2, 3]
aa = []
aa.append(a)
a[1] = 2000
aa.append(a)
```

Is there a problem? Yes. So how do we fix it?

And with an list of lists:

```
a = [1, 2, 3]
aa = []
aa.append(a)
a[1] = 2000
as.append(a)
```

Is there a problem? Yes. So how do we fix it? Same as before or:

```
a = [1, 2, 3]
aa = []
aa.append(a)
import copy
aa = copy.deepcopy(aa)
a[1] = 2000
aa.append(a)
```

The problem with “range”

The following

```
for n in range(1000000000):  
    print(n)
```

will `MemoryError!` in Python 2.4.4 due to memory allocation in the `range` function, but this code with `xrange` works ok:

```
for n in xrange(1000000000):  
    print(n)
```

But `xrange` is “no longer exists” in Python 3.0!

Data types

Dictionary (Python hash): `{}`, `{ 'three': 3, 5: 'five' }`
`{ 'Danmark': 'Copenhagen', 'Botswana': 'Gaborone' }`,
`dict([['Danmark', 'Copenhagen'], ['Botswana', 'Gaborone']])`

Set (distinct unordered): `set()`, `set([1, 2])`, `set([2, 1])`,
`set((2, 1))`, `set(range(0,5)) - set(range(4,10))`,
`set([1, 2]) | set([3, 4])`

Frozenset (immutable): `frozenset()`, `frozenset([1, 2])`
`frozenset([frozenset([1, 2]), frozenset([1, 2, 3])])`

Dictionaries

```
>>> d = {'Danmark': 'Copenhagen', 'Botswana': 'Gaborone'}
```

```
>>> d.keys()
```

```
['Danmark', 'Botswana']
```

```
>>> d.values()
```

```
['Copenhagen', 'Gaborone']
```

```
>>> for (k,v) in d.items():
```

```
>>>     print(k + ' has the capital ' + v)
```

```
Danmark has the capital Copenhagen
```

```
Botswana has the capital Gaborone
```

Control structures: if, for and while

```
xs = [ float(i)/64.0 for i in range(-150, 41) ]
ys = [ float(i)/16.0 for i in range(-25,26) ]
for y in ys:
    s = ''
    for x in xs:
        z = 0j; i = 0
        while i < 10:
            z = z**2 + x+y*1j
            if abs(z) > 2:
                break # Get out of inner loop
            i += 1
        if abs(z) <= 2:
            s += '*'
        else:
            s += ' '
    print(s + '|')
```

Control structures: if, for and while

Other control flows: for-continue, for-else, while-else, if-elif-else

```
for i in range(-10,10):
    if i <= 0:
        continue
    print('Positive: ' + str(i))
```

```
a = 9
for i in range(-10,10):
    if i == a:
        print(str(a) + ' found!')
        break
else:
    print(str(a) + ' was not in the list')
```

Change the inner while to a for loop in the Mandelbrot program

Control structures: if, for and while

```
xs = [ float(i)/64.0 for i in range(-150, 41) ]
ys = [ float(i)/16.0 for i in range(-25, 26) ]
for y in ys:
    s = ''
    for x in xs:
        z = 0j
        for i in range(10):
            z = z**2 + x+y*1j
            if abs(z) > 2:
                s += ' ';
                break
        else:
            s += '*'
    print(s + '|')
```

Functions . . .

Defining and using a function in the interactive Python:

```
>>> def myadd(x,y):  
...     return x+y  
...  
>>> myadd(1,2)
```

The function (reference) can be copied:

```
>>> myadd2 = myadd  
>>> myadd2(2,3)
```

The original deleted and the “copy” still there:

```
>>> del myadd  
>>> myadd2(2,3)
```

... Functions ...

With default input argument:

```
>>> def myadd(x,y=2):  
...     return x+y  
...  
>>> myadd(1)
```

And with named arguments:

```
>>> def mydivide(denominator, nominator):  
...     return denominator/nominator  
...  
>>> mydivide(1.0, 3.0) # 0.33333333333333331  
>>> mydivide(nominator=3.0, denominator=1.0) # 0.33333333333333331  
>>> mydivide(nominator=3.0, 1.0) # Error!  
>>> mydivide(3.0, nominator=1.0) # 3
```

... Functions

Function call with a variable number of input arguments (matlab: `varargin`):

```
def myunion(x, *args):  
    u = set(x)  
    for y in args:  
        u = u.union(set(y))  
    return u
```

`myunion` may now be called with different number of input arguments:

```
>>> myunion([1])  
set([1])  
>>> myunion([1], [1, 2, 4], [3, 4])  
set([1, 2, 3, 4])
```

In the latter case the variable `args` is a tuple:

```
([1, 2, 4], [3, 4])
```


... Functions

It is also possible to handle unknown named arguments:

```
def my_key_union(x, **kwargs):
    union = set(x)
    for key, value in kwargs.items():
        if value:
            union.add(key)
    return union
```

Example call:

```
>>> my_key_union(['the'], stop=True, halt=False)
set(['the', 'stop'])
```

Object-orientation with class: $2 + 2 = 5$

```
class MyInteger():
    def __init__(self, integer):
        print "I am the constructor"
        self.integer = integer
    def __add__(self, integer):          # Overloaded '+' operator
        if self.integer == 2 and integer == 2:
            return 5
        else:
            return self.integer + integer
```

```
>>> a = MyInteger(2)
I am the constructor
>>> a+2
5
>>> 2+a
4
```

Methods and variables in classes

`__init__()` Constructor, called when an instance is made

`__class__` Type of object, e.g., `<type 'list'>`

`__doc__` The documentation string: used for `help()`

`__str__()` Method used for the `print` function

`__getitem__()` Get element: `a.__getitem__(2)` the same as `a[2]`

`__call__()` The method called when the object is a function

And several more ...

`_your_internal_method()` Use initial underscored to indicate a “private” method

`your_method()` Ordinary “public” methods

Variables in classes

```
class Person():
    count = 0                                # Static/class variable
    def __init__(self, name, city):
        self.name = name                    # Variable for the instance
        self.city = city                    # Variable for the instance
        Person.count += 1
    def number_of_persons(self):
        return Person.count
```

```
First = Person("Finn", "Lyngby")
First.number_of_persons()                  # 1
Second = Person("Sine", "Lyngby")
Second.number_of_persons()                 # 2
First.name                                  # Finn
```

Derived classes

```
class Vehicle():
    def is_movable(self): return True

class Car(Vehicle):
    def __init__(self, color): self.color = color
    def has_wheels(self): return True

class CrashedCar(Car):
    # overloading the grandparent 'is_movable' method
    def is_movable(self): return False
```

Instantiating the child class:

```
>>> my_car = CrashedCar('silvergray')
>>> my_car.has_wheels()
True
>>> my_car.is_movable()
False
```

File processing . . .

Writing to a file:

```
fid = open('test.txt', 'w')
fid.write('Hello\nWorld')
fid.close()
```

Reading a file:

```
fid = open('test.txt', 'r')
s = fid.read()                # Read the entire file
fid.close()
```

```
fid = open('test.txt', 'r')
for line in fid:              # Using file identifier as iterator
    print("Line: " + line.strip())
```

```
fid.close()
```

... File processing

Counting the number of lines in the following file:

http://neuro.imm.dtu.dk/software/brede/code/brede/data/stop_english1.txt

... File processing

Counting the number of lines in the following file:

http://neuro.imm.dtu.dk/software/brede/code/brede/data/stop_english1.txt

One solution:

```
fid = open('stop_english1.txt')
k = 0
for line in fid:
    k = k + 1
print(k)
```


... File processing

Counting the number of lines in the following file:

http://neuro.imm.dtu.dk/software/brede/code/brede/data/stop_english1.txt

One solution:

```
fid = open('stop_english1.txt')
k = 0
for line in fid:
    k = k + 1
print(k)
```

Another solution — on one line:

```
len([ line for line in open('stop_english1.txt')])
```

Exceptions

```
try:
    [ int(line) for line in open('stop_english1.txt') ]
except IOError, message:
    print('An IO error', message)
except ValueError, message:
    print('A value error', message)
else:
    print('Success')
```

Exceptions can be 'raised' with raise:

```
raise RuntimeError, 'Another error'
```

New exception types can be defined by subclassing the Exception class.

Exceptions example

Definition of a function that catches an exception and returns NaN (Not a number) on zero division:

```
def mydivide(a, b):
    try:
        return float(a)/b
    except ZeroDivisionError:
        return float('nan')
    except Exception, e:
        print 'Error:', e

>>> 1./0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division
>>> mydivide(1,0)
nan
```

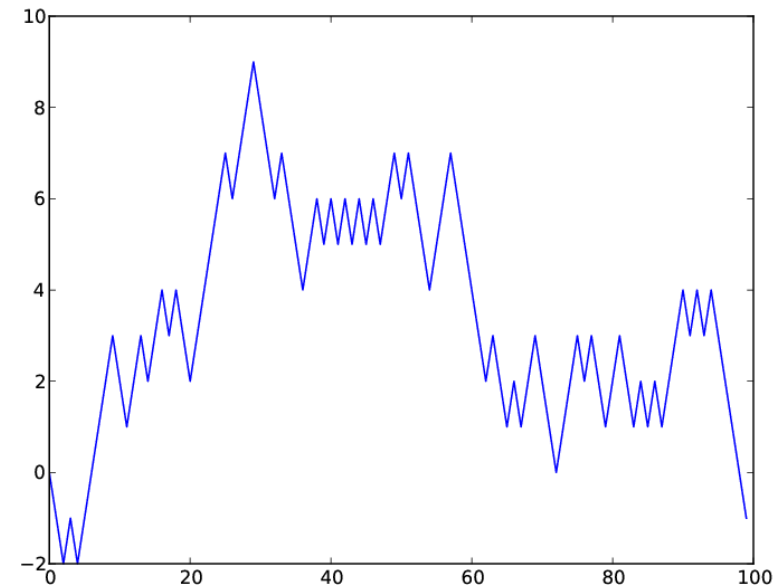
Generators

A function that “remembers”:

```
import random
import matplotlib.pyplot as plt
```

```
def randomwalk():
    x = 0
    while True:
        yield x          # 'yield' instead of 'return'
        x += 2 * (random.random() > 0.5) - 1
```

```
rw = randomwalk()      # 'rw' is a iterator-like object
awalk = [ rw.next() for n in range(100) ]
plt.plot(awalk); plt.show()
```



Libraries, modules, packages . . .

“Module”: Something that you can import with `import`

“Package” = A set of files in a directory (tree)

```
>>> sin(2.0)
```

leads to `NameError`: You need to load a module with `import`

```
import math
dir(math)          # look what is in the 'math' module
math.sin(2.0)
```

```
from math import sin
sin(2.0)
```

... Libraries ...

Loading all functions in the `math` module:

```
from math import *  
sin(2.0)  
cos(2.0)
```

Renaming a loaded function:

```
from math import sin as mysin  
mysin(2.0)
```

After you have made changes in a module it is necessary to reload it:

```
reload(mymodule)
```

Library loading example

Loading a module may also be done ‘within’ the code, e.g., in exception statements:

```
try:
    import urllib3 as urllib      # There is nothing called urllib3
except:
    try:
        import urllib2 as urllib # urllib2 might be installed
    except:
        import urllib as urllib

urllib.urlopen('http://neuro.imm.dtu.dk/')
```

Documentation

```
"""A module with one function called 'myfunction()'"""

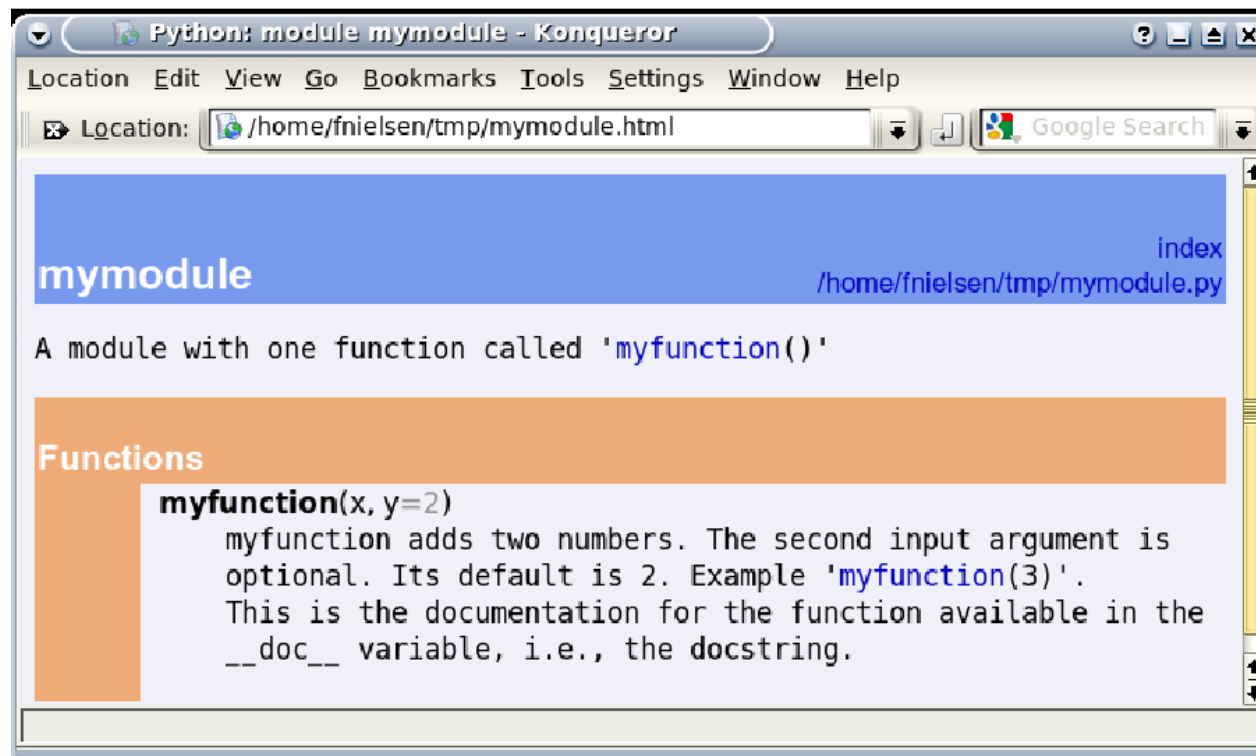
def myfunction(x, y=2):
    """
    myfunction adds two numbers. The second input argument is
    optional. Its default is 2. Example 'myfunction(3)':
    This is the documentation for the function available in the
    __doc__ variable, i.e., the docstring.
    """
    return x + y

print(myfunction.__doc__)
```


HappyDoc, *Epydoc*, *Pydoc*, *Docutils* tools ([Langtangen, 2005](#), section B.2), e.g., `pydoc` included in the basic Python distribution:

```
$ pydoc -w ./mymodule.py
```

This call produces the following HTML page



For documentation see also [Sphinx](#)

Testing

Using nose with `mymodule.py` containing

```
def myfunction(x, y=2):  
    return x+y  
  
def test_myfunction():  
    assert myfunction(1) == 3  
    assert myfunction(1,3) == 4
```

Run the program `nosetests` ([Campbell et al., 2009](#), p. 61–67)

```
$ nosetests mymodule.py
```

that discovers the `test_` functions. Or within Python:

```
>>> import nose, mymodule  
>>> nose.run(mymodule)
```

doctest = documentation + testing ...

mymodule.py with myfunction with Python code in the docstring:

```
def myfunction(x, y=2):
    """
    This function will add two numbers, e.g.,
    >>> myfunction(1,7)
    8

    The second argument is optional
    >>> myfunction(2)
    4
    """
    return x+y

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

... doctest

```
$ python mymodule.py
```

The body of the following conditional gets executed if the function is called as the main program (`__main__`), — rather than imported as a module:

```
if __name__ == "__main__":  
    import doctest  
    doctest.testmod()
```

`doctest.testmod()` will extract the code and the execution result from the docstrings (indicated with `>>>`), execute the extracted code and compare its result to the extracted result: Literal testing!

This scheme also works for errors and their messages.

Checking code style and quality

```
$ pylint a_python_module.py
```

This use of `pylint` may result in:

```
***** Module a_python_module
W: 14: Unnecessary semicolon
C:  1: Missing docstring
W:  5:formatresult: Redefining name 'res' from outer scope (line 14)
C: 14: Invalid name "res" (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
W: 17: No exception type(s) specified
C: 16: More than one statement on a single line
C: 17: More than one statement on a single line
W:  3: Unused import math
```

Summary

Python is object-oriented so you can define classes, derive them, define methods and variables within the class.

Python has a range of basic and more complex data types: integer, strings, dictionaries, ... Each type has lots of methods defined.

There is a standard set of control flow (if, else, for, while, ...)

You can write structure documentation together with the code.

There are support for testing, e.g., with nose (but also py.test and unittest).

You can check the style of your code with, e.g., pylint (but other tools exist)

References

Campbell, J., Gries, P., Montojo, J., and Wilson, G. (2009). *Practical Programming: An Introduction to Computer Science Using Python*. The Pragmatic Bookshelf, Raleigh.

Langtangen, H. P. (2005). *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer. ISBN 3540294155.