Python programming — introduction to Python

Finn Årup Nielsen

Department of Informatics and Mathematical Modelling
Technical University of Denmark
Lundbeck Foundation Center for Integrated Molecular Brain Imaging
Neurobiology Research Unit,
Copenhagen University Hospital Rigshospitalet

August 28, 2009

First a case

Downloading a Web page

Read information from the Web (Martelli et al., 2005, p. 489)

```
# Import the 'urllib' library for Web page retrieval
from urllib import urlopen

url = 'http://neuro.imm.dtu.dk/tmp/w/index.php/' + \
      'Special:Ask/-5B-5Bfeed::+-5D-5D/-3FFeed/' + \
      'sort=/order=ASC/format=csv/sep=,/limit=100'

help('urllib')

# Get and read the web page
doc = urlopen(url).read()  # Read is built-in

print(doc)
```

Output from the print function

```
"Autoblog Green",http://feeds.autoblog.com/weblogsinc/autoblog
"Brussels Sunshine",http://blog.brusselssunshine.eu/feeds/posts/default
"Capital Eye",http://www.opensecrets.org/news/atom.xml
Causecast,http://feeds.feedburner.com/causecast/latest_news.rss
"Clean Fuels Blog",http://feeds.feedburner.com/CFDC?format=xml
"Close Concerns Weblog",http://closeconcerns.typepad.com/close_ ...
"Corporate Eye Corporate social responsibility",http://feeds. ...
"Corporate social responsibility (guardian)",http://www.guardian. ...
"Corpwatch Blog",http://www.corpwatch.org/rss.php
"Crane and Matten blog",http://craneandmatten.blogspot.com/feeds ...
"Dax Mahoney (blog)",http://daxmahoney.blogspot.com/feeds/posts/default
...
```

Reading the comma separated values

```
# Import a CSV reader/writer library.
import csv

web = urlopen(url)
# 'web' is now a file-like handle

blogs = csv.reader(web, delimiter=',', quotechar='"')
# 'blogs' is now an object that can be iterated over

# Iterate over 'blogs'
for blog in blogs:
    print(blog)
```

Outout from the print function

```
['Autoblog Green', 'http://feeds.autoblog.com/weblogsinc/autoblog']
['Brussels Sunshine', 'http://blog.brusselssunshine.eu/fe ...
['Capital Eye', 'http://www.opensecrets.org/news/atom.xml']
['Causecast', 'http://feeds.feedburner.com/causecast/latest_ ...
['Clean Fuels Blog', 'http://feeds.feedburner.com/CFDC?format=xml']
['Close Concerns Weblog', 'http://closeconcerns.typepad. ...
['Corporate Eye Corporate social responsibility', 'http:// ...
['Corporate social responsibility (guardian)', 'http://www ...
['Corpwatch Blog', 'http://www.corpwatch.org/rss.php']
['Crane and Matten blog', 'http://craneandmatten.blogspot.c ...
['Dax Mahoney (blog)', 'http://daxmahoney.blogspot.com/feed ...
['Dgoodr', 'http://feeds.feedburner.com/dgoodr?format=xml']
...
```

Adding the URLs to a Python list

```
# Each row is of a Python 'list' type
isinstance(blog, list)   # or type(blog)

blogs = csv.reader(urlopen(url), delimiter=',', quotechar='"')

# Create empty list
urls = []

for blog in blogs:
    # Python indexes from 0: '1' is the second column
    feed = blog[1]
    if len(feed):
        urls.append(feed)
```

The feeds

```
>>> urls[0]
'http://feeds.autoblog.com/weblogsinc/autoblog'

>>> doc = urlopen(urls[0]).read()
>>> print(doc[0:600])
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" media="screen" href="/~d/ ...
<channel>
<title>Autoblog</title>
<link>http://www.autoblog.com</link>
<description>Autoblog</description>
<image>
<url>http://www.blogsmithmedia.com/www.autoblog.com/media/feedlogo.
```

Reading feeds

Reading feeds from, e.g., blogs (Segaran, 2007, p. 229)

```
import feedparser

f = feedparser.parse(urls[0])
# Now 'f' is a kind of Python dictionary

>>> f.keys()
['feed', 'status', 'updated', 'version', 'encoding', 'bozo',
'headers', 'etag', 'href', 'namespaces', 'entries']

>>> f['entries'][0].keys()
['summary_detail', 'author', 'links', 'title', 'feedburner_origlink',
'tags', 'updated', 'comments', 'summary', 'guidislink',
'title_detail', 'link', 'id', 'updated_parsed']
```

Examining the feeds

The 7th tag of the 1st entry in the downloaded feed

```
>>> f['entries'][0]['tags'][6]['term']
u'obama administration'
```

All tags for all posts in one particular feed:

```
tags = [];
for e in f['entries']:
    for t in e['tags']:
        tags.append(t['term']);
```

Getting all feeds

```
from BeautifulSoup import BeautifulSoup          # HTML reading library
fs = [];
for url in urls:
 fs.append(feedparser.parse(url))
 fs[-1]['wordlist'] = []
 for e in fs[-1]['entries']:
  if e.has_key('summary'):
   fs[-1]['wordlist'].extend(''.join(BeautifulSoup( \
    e.summary).findAll(text=True)).split());
 print(url)

allwords = [word for f in fs for word in f['wordlist']]
```

Some statistics

```
float(len(allwords))/len(set(allwords))

wordcount = dict([ [t, allwords.count(t)] for t in set(allwords) ])

wordcount = {}
for term in allwords:
    wordcount[term] = wordcount.get(term, 1) + 1

items = [(v, k) for k, v in wordcount.items()]
items.sort()
items.reverse()

for n in range(0,2000):
    print('%3d: %4d %s' % (n+1, items[n][0], items[n][1]))
```

```
  1: 5205 the
  2: 3211 to
  3: 2922 of
```

Determine “interesting” words

Reading a stopword list

```
stopwords = [ line.strip() for line in open('stop_english1.txt', 'r') ]

wordcount = {}
for term in allwords:
    if term.lower() not in stopwords:
        wordcount[term] = wordcount.get(term, 1) + 1

items = [(v, k) for k, v in wordcount.items()]  # 'Inverting' the dict
items.sort()
items.reverse()

terms = []
for n in range(0,500):
    terms.append(items[n][1])
    print('%3d: %4d %s' % (n+1, items[n][0], items[n][1]))
```

Making a matrix

To make numerical processing in Python import the `numpy` module and then initialize the elements

```
import numpy                    # Import of the numerical module

M = numpy.matrix(numpy.zeros([len(fs), len(terms)]))
for n in range(len(fs)):
    for m in range(len(terms)):
        M[n,m] = fs[n]['wordlist'].count(terms[m])
```

The `M` matrix has the size feeds-by-terms and each element is set to the number of times a term (i.e., a word) occurs in a feed.

Multivariate analysis algorithm:

```
def nmf(M, components=5, iterations=5000):
    import random
    W = numpy.matrix(numpy.zeros([M.shape[0], components]))
    H = numpy.matrix(numpy.zeros([components, M.shape[1]]))
    for n in range(M.shape[0]):
        for m in range(components):
            W[n,m] = random.random()
    for n in range(components):
        for m in range(M.shape[1]):
            H[n,m] = random.random()
    for n in range(0, iterations):
        H = numpy.multiply(H, (W.T * M) / (W.T * W * H + 0.001))
        W = numpy.multiply(W, (M * H.T) / (W * (H * H.T) + 0.001))
        print "%d/%d" % (n, iterations)
    return (W, H)
```

Using the algorithm on the matrix

```
(W, H) = nmf(M)
```

. . . And then display the results:

```
for n in range(4):
    print n, "-----"
    w = []; h = []
    for m in range(W.shape[0]):
        w.append((W[m,n], fs[m]))
    for m in range(H.shape[1]):
        h.append((H[n,m], terms[m]))
    h.sort()
    h.reverse()
    w.sort()
    w.reverse()
    for n in range(0,20):
      print "%5.3f %10s %s" % (h[n][0], h[n][1], w[n][1]['feed']['title'])
```

Python introduction: Back to the basics

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)'
exec(s)
```

and evaluation

```
s = '1+1'
a = eval(s)
print(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 formating

if len(sys.argv) < 3:                # Conditionals should be indended
    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 identation
        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`?

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)`, `1.00000000000000000000000001 == 1`

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

Long: `123198098547612389132091820398123012 3`, `long(1)`, `5L`

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 does 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
```

...and a hack

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

Data types: Sequences

String: `'A string'`, `"Another"`, `'<a href="http://dtu.dk">DTU</a>'`, `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)`

Indexing with and function for sequences

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

```
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]
as = []
as.append(a)
a[1] = 2000
as.append(a)
```

Is there a problem?

And with an list of lists:

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

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

And with an list of lists:

```
a = [1, 2, 3]
as = []
as.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]
as = []
as.append(a)
import copy
as = copy.deepcopy(as)
a[1] = 2000
as.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!
(http://docs.python.org/dev/3.0/whatsnew/3.0.html)

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)                          # 0.3333333333333331
>>> mydivide(nominator=3.0, denominator=1.0)    # 0.3333333333333331
>>> mydivide(nominator=3.0, 1.0)                # Error!
>>> mydivide(3.0, nominator=1.0)                # 3
```

. . . Functions

Function call with a variable number of input arguments:

```
def myunion(x, *varargin):
    u = set(x)
    for y in varargin:
        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 `varargin` is a tuple:

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

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

```
class myint(int):                          # Inheritance from 'int'
    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 = myint(2)
I am the constructor
>>> a+2
5
>>> 2+a
4
```

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('Succes')
```

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
```

Libraries . . .

Python library = "module"

"Package" = A set of "modules" 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 changing 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/')
```

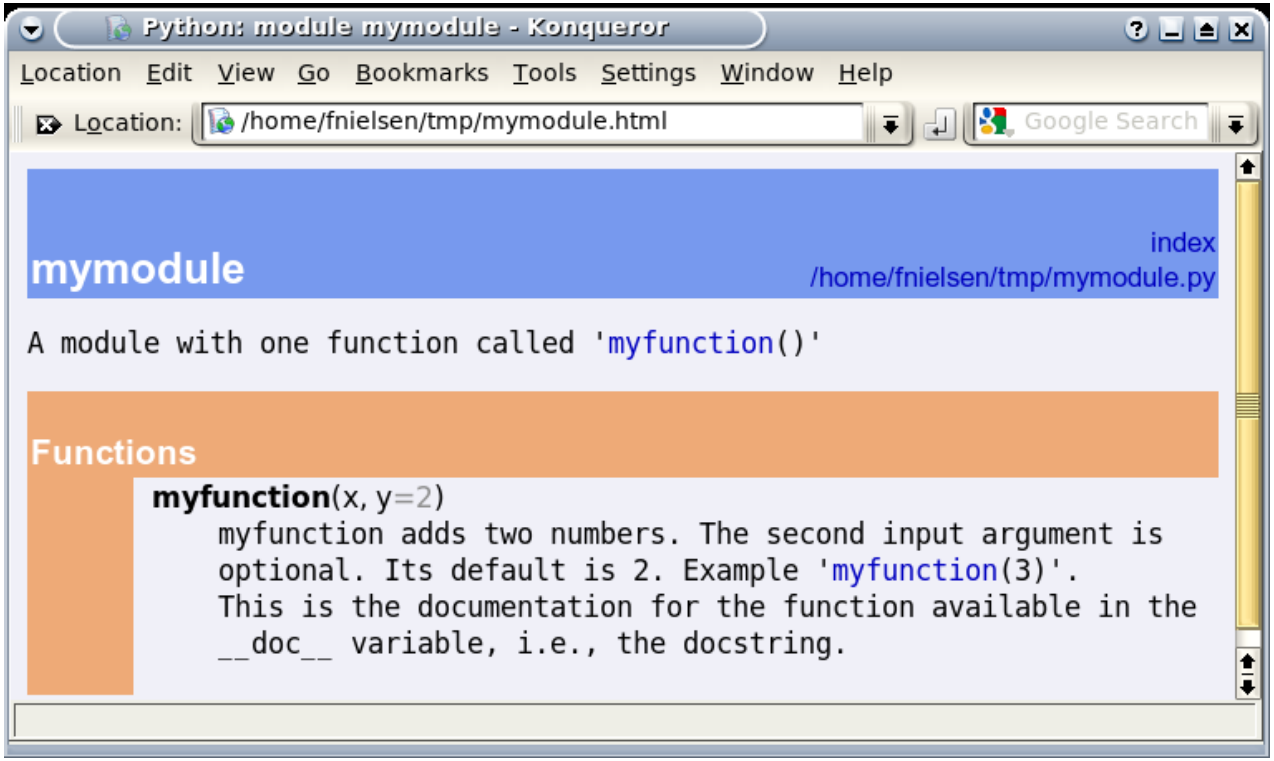
Documentation

```
"""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



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.

A problem

Consider a file with the following matrix $\mathbf{X}$:

```
1 2
3 4
```

Read and compute $\mathbf{Y} = 2 * \mathbf{X}$

A problem

Consider a file with the following matrix $\mathbf{X}$:

```
1 2
3 4
```

Read and compute $\mathbf{Y} = 2 * \mathbf{X}$

```
x = [ [ int(s) for s in line.split()] for line in open('tmp.txt') ]
y = []
for xrow in x:
  yrow = []
  for element in xrow:
      yrow.append(2*element)
  y.append(yrow)
```

A problem

Consider a file with the following matrix $\mathbf{X}$:

```
1 2
3 4
```

Read and compute $\mathbf{Y} = 2 * \mathbf{X}$

Or with numpy

```
from numpy import *

x = [ [ int(s) for s in line.split()] for line in open('tmp.txt') ]
y = 2*matrix(x)
```

Debian packages

python-beautifulsoup, python-feedparser, python-gnuplot, python-imaging, python-json, python-librdf, python-libsvm, python-matplotlib, python-nifti, python-numpy, python-nose, python-pysqlite2, python-rdflib, python-scipy, python-sparse, python-xml

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, second edition. ISBN 3540294155.

Martelli, A., Ravenscroft, A. M., and Ascher, D., editors (2005). *Python Cookbook*. O'Reilly, Sebastopol, California, 2nd edition.

Segaran, T. (2007). *Programming Collective Intelligence*. O'Reilly, Sebastopol, California.