ACADEMIS

Create Better Science

Python & Biopython

Quick Reference

- Overview of Python
- Python Reference
- Biopython Reference
- NCBI EUtils
- Exercise for the os module

With contributions by

Kristian Rother, Allegra Via & Magdalena Rother

Python Reference

Version 1.3 – abriged October 2013

Contents

This document contains contributions by Magdalena Rother, Kaja Milanowska and Anna Philips.

Python Overview

Overview

Python is an interpreted language.

Python uses fully dynamic typing.

The Python interpreter generates intermediate code (.pyc files).

Guido van Rossum, the lead developer works for Google.

Strengths:

- Quick to write, no compilation
- Fully object-oriented.
- Many reliable libraries.
- All-round language.
- 100% free software.

Weaknesses:

- Writing very fast programs is not easy.
- No strict encapsulation.

Recommended Books

Learn Python the hard way
Zed Shaw

Python in a Nutshell
O'Reilly

The Python Cookbook
O'Reilly

Online Resources

Main documentation and tutorial:
http://www.python.org/doc

Tutorial for experienced programmers:
http://www.diveintopython.org

Tutorial for beginners:
http://greenteapress.com/thinkpython/thinkCSpy/html/

Python Reference Manual – language basics:
http://docs.python.org/lib/lib.html

Global Module Index – description of standard modules:
http://docs.python.org/modindex.html

The Python Command Line (CLI)

Overview

You can use any Python command from the interactive Python command line:

```
>>> print 4**216
>>> a = 'blue'
>>> print a
blue
>>>
```

Python works in the same way as it would in a program, except that results are automatically printed.

Define code blocks by indenting extra lines:

```
>>> for i in range(3):
…     print i
…
0
1
2
>>>
```

Tips

You can leave the command line by **Ctrl-Z** (Windows) or **Ctrl-D** (Linux).

The CLI works great as a pocket calculator.

Writing code blocks with 2+ lines in the CLI gets painful quickly.

From IDLE, you can execute a program in the command line by pressing F5.

Python Programs

All program files should have the extension **.py**

Indentation is a central element of Python syntax, marking code blocks. Code blocks should be indented by four spaces/one tab. Indentation must not be used for decorative purposes.

Only one command per line is allowed.

Developing programs on Unix

When developing on Unix, the first line in each Python program should be:

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

Starting programs on Windows

Often, it is useful to use a batch file to start a Python program on Windows. To do so, create an empty file with the ending .bat (Save as.. in IDLE), and write into it:

```
C:\Python26\python.exe my_script.py
```

Comments

In Python, single lines can be commented by the hash symbol:

```
# this is a comment.

print a + b  # adding the variables
```

Also, multi-line comments can be enclosed by triple quotes:

```
'''
This is a longer description
that stretches over multiple lines.
'''
```

The triple quoted comments can be used to generate documentation automatically.

Introspection and Namespaces

Introspection is a feature of Python by which you can examine objects (including variables, functions, classes, modules) at runtime.

Everything is an Object

One consequence of the dynamic typing is that Python can treat everything it manages technically in the same way. Everything is an object, meaning it has attributes. These attributes are objects themselves. Methods are simply attributes that can be called.

Each object has a name:

```
print x.__name__
```

and a type:

```
print type(x)
```

Built-in help

You can get context-sensitive help to functions, methods and classes that utilize the triple-quoted comments:

```
import time
print help(time.asctime)
```

Namespaces

In Python classes, functions, modules etc. have separate namespaces. A namespace is the set of attributes of an object. You can explore a namespace with dir():

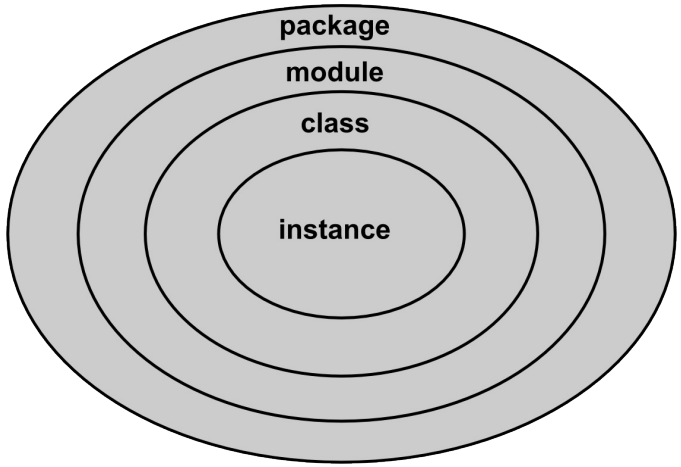
```
print dir()

import time

print dir(time)
```

Namespaces are nested

When using a name, Python first uses the local namespace (e.g. of a function/method), then in the class, module, package, and finally in the global namespace.



Variables and Data Types

Variables

Variable names may be composed of letters, underscores and, after the first position, also digits.

Lowercase letters are common, but uppercase is also allowed (usually used for constants).

```
invitation = 'Hello World'
a = 10
b = 3.0
PI = 3.1415
```

Arithmetical operations

Basic operations and variable assignments:

```
a = 7
b = 4
c = a - b  # 3
d = a*b    # 28
e = a/b    # 1
f = a%b    # modulo, 3
g = a**2   # 49
h = 3.0/b  # 0.75
i = 7.0//b # floor div, 1.0
```

Immutable data types

The basic immutable data types in Python are:

Boolean (`True / False`).

Integer (`0, 1, -3`).

Float (`1.0, -0.3, 1.2345`)

Strings (`'apple', "banana"`) - both single and double quotes are equivalent.

None (aka an empty variable).

Type conversions

Values of variables can be converted using:

`int(value)` → from a float or string.

`float(value)` → from a int or string.

`str(value)` → from any Python variable, including `None`, any function or object.

Strings

Accessing individual characters

Using square brackets, any character of a string can be accessed. The first character has the index 0.

```
s = 'Manipulating Strings'
print s[0]  # first character
print s[3]
print s[-1] # last character
print s[-2] # second last
```

Creating substrings

Substrings can be formed by applying square brackets with two numbers inside separated by a colon. The second number is not included in the substring itself.

```
s = 'Manipulating Strings'
print s[1:6]  # 'anipu'
print s[9:13] # 'ing '
print s[0:2]  # 'Ma'
print s[:3]   # 'Man'
print s[-4:]  # 'ings'
```

Length of strings

Is returned as an integer by the `len()` function.

```
print len('Manipulating Strings')
```

String functions

The following functions work on every string variable.

```
s = 'Manipulating Strings '
```

Changing case:

```
s.upper()
s.lower()
```

Removing whitespace at both ends:

```
s.strip()
```

Cutting a string into columns:

```
s.split(' ')
```

Searching for substrings:

```
print s.find('ing')
```

Replacing substrings:

```
print s.replace('Strings','text')
```

Removing whitespace at both ends:

```
s.startswith('Man')
s.endswith('ings')
```

Tuples and Lists

Creating tuples

A tuple is a sequence of elements that cannot be modified. They are useful to group elements of different type.

```
t = ('bananas','200g',0.55)
```

Creating lists

A list is a sequence of elements that can be modified. In many cases, all elements of a list will have the same type.

```
data = ['apples', 'bananas', 'oranges']
```

Accessing elements of lists and tuples

Using square brackets, any element of a list and tuple can be accessed. The first character has the index 0.

```
data = [1,2,3,4]
print data[0]    # first
print data[3]    # last
print data[-1]   # last
data[0] = 'kiwi' # assigning an element
```

Creating lists from other lists:

Lists can be sliced by applying square bracketsin the same way as with strings.

```
data = [0,1,2,3,4,5]
print data[1:3]  # [1,2]
print data[0:2]  # [0,1]
print data[:3]   # [0,1,2]
print data[-2:]  # [4,5]
m = data[:]      # creates a copy!
```

List methods

Lists provide many useful methods:

```
data[0] = 'kiwi' # assigns an element
```

Adding elements to a list:

```
data.append(5)   # adds at the end
```

Removing elements from a list:

```
data.remove(3)
data.pop()       # removes last element
```

Sorting a list:

data.sort()

Count elements:

data.count(3)

Built-in functions working on iterable types

Determining the length of sequences

The `len()` functions returns an integer with the length of an argument. It works with strings, lists, tuples, and dictionaries.

```
data = [0,1,2,3]
print len(data)  # 4
```

Creating lists of integer numbers

The `range()` function allows to create lists of numbers on-the-fly. There are two optional parameters for the start value and the step size.

```
data = range(4)        # [0,1,2,3]
data = range(1,5)      # [1,2,3,4]
data = range(2,9,2)    # [2,4,6,8]
data = range(5,0,-1)   # [5,4,3,2,1]
```

Summing up numbers

The `sum()` of a list of integer or float numbers or strings can be calculated by the `sum()` function.

```
data = [1,2,3,4]
print sum(data)        # 10
```

Enumerating elements of lists

The `enumerate()` function associates an integer number starting from zero to each element in a list. This is helpful in loops where an index variable is required.

```
fruits = ['apple', 'banana', 'orange']
for i, fruit in enumerate(fruits):
    print i, fruit
```

Merging two lists

The `zip()` function associates the elements of two lists to a single list or tuple. Excess elements are ignored.

```
fruits = ['apple','banana','orange']
prices = [0.55, 0.75, 0.80, 1.23]

for fruit,price in zip(fruits, prices):
     print fruit, price
```

Other functions

`sorted(data)` returns a sorted list

`map(f, data)` applies a function to all elements.

`filter(f, data)` returns elements for which f is True.

`reduce(f, data)` collapses the data into one value.

Dictionaries

Creating dictionaries

Dictionaries are an unordered, associative array. They have a set of key/value pairs. They are very versatile data structures, but slower than lists. Dictionaries can be used easily as a hashtable.

```
prices = {
     'banana':0.75,
     'apple':0.55,
     'orange':0.80
     }
```

Accessing elements in dictionaries

By applying square brackets with a key inside, the values of a dictionary can be requested. Valid types for keys are strings, integers, floats, and tuples.

```
print prices['banana']  # 0.75
print prices['kiwi']    # KeyError!
```

Looping over a dictionary:

You can access the keys of a dictionary in a for loop. However, their order is not guaranteed, then.

```
for fruit in prices:
     print fruit
```

Dictionary functions

There is a number of functions that can be used on every dictionary:

Checking whether a key exists:

```
prices.has_key('apple')
```

Getrieving values in a fail-safe way:

```
prices.get('banana') # 0.75
prices.get('kiwi')   # None
```

Setting values only if they dont exist yet:

```
prices.setdefault('kiwi',0.99)
prices.setdefault('banana',0.99)
# for 'banana', nothing happens
```

Getting all keys / values:

```
print prices.keys()
print prices.values()
print prices.items()
```

Input and Output

Text input and output

Reading from the text console

User input can be read from the keyboard with or without a message text. The value returned is always a string:

```
a = raw_input()

a = raw_input('Please enter a number')
```

Writing to the text console:

The Python print statement is very versatile and accepts almost any combination of strings, numbers, function calls, and arithmetic operations separated by commas.

```
print 'Hello World'
print 3 + 4
print 3.4
print """Text that stretches over
multiple lines."""
print 'number', 77
print
print int(a) * 7
```

String formatting

Variables and strings can be combined, using formatting characters. This works also within a print statement. In both cases, the number of values and formatting characters must be equal.

```
s = 'Result: %i'%(number)

print 'Hello %s!'%('Roger')

print '(%6.3f/%6.3f)'%(a,b)
```

The formatting characters include:

`%i` – an integer.

`%4i` – an integer formatted to length 4.

`%6.2f` – a float number with 6 digits (2 after the dot).

`%10s` – a right-oriented string with length 10.

Escape characters

Strings may contain also the symbols: `\t` (tabulator), `\n` (newline), `\r` (carriage return), and `\\` (backslash).

File operations

Opening a file for reading

Text files can be accessed using the `open()` function.
From an open file the contents can be read as a string.

```
f = open('my_file.txt')
text = f.read()
```

Opening a file for writing

Writing text to files is very similar. The only difference is the `'w'` parameter.

```
f = open('my_file.txt','w')
f.write(text)
```

Appending to a file

It is possible to add text to an existing file, too.

```
f = open('my_file.txt','a')
f.write(text)
```

Closing files

Closing files in Python is not mandatory.

```
f.close()
```

Tips for reading files

An open file behaves like a list of strings. Thus, it is possible to read it line by line without using `s.split('\n')`:

```
for line in open('my_file.txt'):
     name = line[:10].strip()
     print name.strip()
```

Tips for writing files

Analogously, if you have data as a list of strings, it can be written to a file as a one-liner. You should remember to add linebreaks:

```
lines = ['first line\n',
         'second line\n']
open('my_file.txt','w').writelines(lines)
```

Writing directory names in Python

When opening files, you can use full or relative directory names. However, you must replace the backslash '\' by a double backslash `'\\'` (because `'\'` is also used for `'\n'` and `'\t'`).

```
f = open('..\\my_file.txt')
f = open('C:\\python\\my_file.txt')
```

Control Flow Statements

Conditional statements with if

if statements are used to implement decisions and branching in the program. They must contain an 'if' block, and optionally none to many 'elif's and an 'else' block:

```
if fruit == 'apple':
    price = 0.55
elif fruit == 'banana':
    price = 0.75
elif fruit == 'orange':
    price = 0.80
else:
    print 'we dont have %s'%(fruit)
```

Boolean value of variables

Each variable can be interpreted as boolean logic. All values are treated as `True`, except for:

```
False
0
[]
''
{}
None
```

Code blocks

After an if statement, all indented commands are treated as a code block, and are executed if the condition applies.

The next unindented command Is executed in any case.

Comparison operators

An expression may contain any combination of:

`a == b, a != b` (equality)

`a < b, a > b, a <= b, a >= b` (relations)

`a or b, a and b, not a` (boolean logic)

`(a or b) and (c or d)` (priority)

`a in b` (inclusion, when `b` is a list)

Loops with for and while

Unconditional loops with for

`for` loops require a sequence of items that is iterated over. This can be a list, a tuple, a string, a dictionary or any Python object that behaves like one of them (e.g. a file). Examples:

```
for i in range(3):
     print i

for char in 'ABCD':
     print char

for elem in [1,2,3,4]:
     print elem
```

Conditional loops with while

While loops require a conditional expression at the beginning. The conditional expressions work in exactly the same way as `in if.. elif` statements.

```
i = 0
while i < 5:
     print i   i = i + 1
```

When to use for

When you want to do something for all elements of a list.

When the number of iterations is known at the beginning.

Printing numbers, concatenating a list of strings.

Modifying all elements of a list.

When to use while

When there is a exit condition.

When it may happen that nothing is done at all.

When the number of repeats depends on user input.

Searching for a particular element in a list.

Program Structures

Functions

Structuring code with functions

A function is an autonomous sub-program with local variables, input and output, and its own namespace.
In Python, a function definition must be followed by a code block:

```
def calc_price(fruit,n):
    '''Returns the price of fruits.'''
    if fruit=='banana':
        return 0.75 * n

print calc_price('banana',10)
```

Optional parameters

Function parameters may have default values. In that case, they become optional. **Do not use mutable types as default arguments!**

```
def calc_price(fruit, n=1): ..

print calc_prices('banana')
print calc_prices('banana',100)
```

List and keyword parameters

You can add a list `*args` for an unspecified number of extra parameters, or a dictionary `**kwargs` for keyword parameters.

```
def func(*args, **kwargs):
    print args, kwargs

func(1, 2, a=3, b=4)
```

Return values

A function may return values to the program part that called it.

```
return 0.75
```

Multiple values are returned as a tuple.

```
return 'banana', 0.75
```

In any case, the return statement ends the functions execution.

Modules

What is a module?

Any Python file (ending .py) can be imported from another Python script. A single Python file is also called a **module**.

Importing modules

To import from a module, its name (without .py) needs to be given in the `import` statement. It is strongly recommended to list the variables and functions required explicitly instead of using the `import *` option (helps with debugging). Package names may be given as well.

```
import fruit
from fruit import fruit_prices
from my_package.fruit import fruit_prices
```

What are the .pyc files for?

When importing, Python generates intermediate code files (.pyc) that help to execute programs faster. They are managed automatically, and dont need to be updated.

Packages

What is a package?

A package is a directory with Python files and a file `__init__.py`.

For big programs, it is useful to divide up the code among several directories.

To import the package from outside, there needs to be a file `__init__.py` (it may be empty).

The directory with the package needs to be in the Python search path (see below).

Where does Python look for modules and packages?

When importing modules or packages, Python looks in:

The current directory.

The site-packages folder (where Python is installed).

In the `PYTHONPATH` environment variable.

In Python the latter converts to the `sys.path` variable

```
import sys
print sys.path
```

Python library modules

The CSV module

The csv module reads tables from text files.

```
import csv
reader = csv.reader(open('my_file.csv'))
for row in reader:
    print row
```

Similarly, tables can be written to files:

```
write = csv.writer(open('my_file.csv','w'))
write.writerow(table)
```

Options of CSV file readers/writers

Both CSV readers and writers can handle a lot of different formats. Options you can change include:

`delimiter` : the symbol separating columns.

`Quotechar` : the symbol used to quote strings.

`Lineterminator` : the symbol at the end of lines.

```
reader = csv.reader(open('my_file.csv'),
delimiter='\t', quotechar='”')
```

Getting time and date

The time module offers functions for getting the current time and date.

```
import time
s = time.asctime()    # as string
i = time.time()       # as float
```

Accessing web pages

The HTML code of web pages and downloadable files from the web can be accessed in a similar way as reading files:

```
import urllib2
url = 'http://www.google.com'
page = urllib.urlopen(url)print page.read()
```

Regular Expressions

Regular expressions allow to perform string search & replace operations using patterns.

Searching

import re

```
text = 'all ways lead to Rome'
```

Search for a pattern in a string:

```
re.search('R...\s', text)
```

Find all words:

```
re.findall('\s(.o)', text)
```

Replacing

re.sub('R[meo]+','London', text)

How to find the right pattern for your problem:

Finding the right RegEx requires lots of trial-and-error search. You can test regular expressions online before including them into your program:

`http://www.regexplanet.com/simple/`

Characters used in RegEx patterns:

Some of the most commonly used characters in Regular Expression patterns are:

\d - decimal character [0..9]

\w - alphanumeric [a..z] or [0..9]

\A - start of the text

\Z - end of the text

[ABC] - one of characters A,B,C

. - any character

^A - not A

a+ - one or more of pattern a

a* - zero or more of pattern a

a|b - either pattern a or b matches

\s - empty space

Ignoring case

If the case of the text should be ignored during the pattern matching, add `re.IGNORECAS`E to the parameters of any re function.

Managing files and directories

Change the current directory

With the os module, you can change the current directory:

```
import os
os.chdir(''..\\python'')
```

Listing files in a directory

```
os.listdir(''..\\python'')
```

Check whether a file exists

```
print os.access('my_file.txt')
```

Installable Packages

Creating plots

The Matplotlib library contains a wide range of possibilities for creating graphs. See the 'gallery' link on http://matplotlib.sourceforge.net for examples. **It needs to be installed separately.**

from pylab import *

Number crunching

Numpy is a library that allows operating on arrays and matrices. **It needs to be installed separately.** See: http://numpy.scipy.org

import numpya = numpy.array([1,2,3,4,5,6])print a+10, a*a

Image manipulation

The Python Imaging Library PIL is a powerful tool for working with images (changing formats, resizing, cutting, drawing). **It needs to be installed separately.** See http://www.pythonware.com/products/pil .

1. Biopython Quick Reference

With contributions by Kristian Rother, Allegra Via, Magdalena Rother, Olga Sheshukova

Biopython is a Python library for reading and writing many common biological data formats. It contains some functionality to perform calculations, in particular on 3D structures. The library can be found at **www.biopython.org**.

1.1 Diving into Biopython

Getting started quickly:

```
import Bio
from Bio.Seq import Seq
dna = Seq("ACGTTGCA")
print dna
```

1.2 Reverse complement, transcribing & translating

```
dna.reverse_complement()
rna = dna.transcribe()
rna.translate()
```

1.3 Calculating GC-content

```
from Bio.SeqUtils import GC
GC(dna)
```

1.4 Calculating molecular weight (DNA only)

```
from Bio.SeqUtils import molecular_weight
molecular_weight("ACCCGT")
```

1.5 Loading sequences from a FASTA file

```
from Bio import SeqIO
for record in SeqIO.parse("ls_orchid.fasta", "fasta"):
    print record.seq, len(record.seq)
```

2. Using NCBI E-utilities

2.1 Documentation

http://www.ncbi.nlm.nih.gov/books/NBK25500/

2.2 Searching for papers in PubMed

http://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?db=pubmed&term=thermophilic,packing&rettype=uilist

2.3 Retrieving publication records in Medline format

http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=pubmed&id=11748933,11700088&retmode=text&rettype=medline

2.4 Searching for protein database entries by keywords

http://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?db=protein&term=cancer+AND+human

2.5 Retrieving protein database entries in FASTA format

http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=protein&id=1234567&rettype=fasta

2.5 Retrieving protein database entries in Genbank format

http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=protein&id=1234567&rettype=gb

2.6 Retrieving nucleotide database entries

http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nucleotide&id=9790228&rettype=gb

2.7 In Python

```
import urllib
result = urllib.urlopen("http...").read()
```

The os module

Insert the correct functions into the gaps.

The Python module [ ] is very useful for interactions with the operating system. In fact, it is a combination of two modules: os and [ ]. Before any of them can be used, the modules need to be activated by the [ ] statement.
Among the most frequently used operations is the [ ] function, that returns a list of all files in the given directory. If a program already has a filename, but it needs to be checked whether the file really exists, the function [ ] will return **True** or **False**. If a file needs to be deleted, this can be done using [ ].
A very useful feature of the **os.path** module is that it helps operating with directory names. Probably the most frequently used function is [ ], that separates a file name from directory names.
But **os** can do even more: You can use any shell command from a Python program with [ ] - However, this method has disadvantages: it depends on the operating system, and is a potentially insecure.

1. `os.access(fn,os.F_OK)`
2. `os.remove(filename)`
3. `os.path`
4. `os.listdir()`
5. `os.system(command)`
6. `os.path.split(os.getcwd())`
7. `import os`
8. `os`

K. Rother, A. Philips & K. Milanowska