Introduction to Python

Part 1: The Python Programming Language

Michael Kraus (michael.kraus@ipp.mpg.de)

Max-Planck-Institut für Plasmaphysik, Garching

17. November 2011

About Python

- generic, modern, interpreted and object oriented computing language
- open source (interpreter, libraries, IDEs)
- portable (available on Linux, Mac, Windows, ...)
- powerful data structures and practical shortcuts
- lots of modules for all purposes

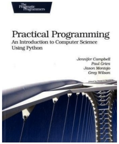
  (easy: install modules into home directory w/o additional configuration)
- extendable with C/C++ and Fortran (Cython, Weave, ctypes, f2py, ...)
- well suited for scientific applications
  - numpy: provides numerical array objects and routines to manipulate them
  - scipy: high-level data processing routines (regression, interpolation, ...)
  - matplotlib: 2D visualisation (publication-ready plots)
  - netcdf/hdf5: efficient data storage for scientific applications

  → glue between existing components: couple your already-written code with the plethora of available python modules

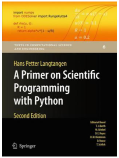
  → powerful but simple reading and writing of data in different file formats

  → analysis and visualisation of data (simulation, experiment)

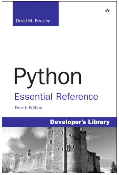
References



Practical Programming: An Introduction to Computer Science Using Python (2009)
Jennifer Campbell, Paul Gries, Jason Montojo, Greg Wilson



A Primer on Scientific Programming with Python (2011)
Hans Petter Langtangen



Python Essential Reference (2009)
David Beazley

References

- Python Scientific Lecture Notes
  http://scipy-lectures.github.com/
- Google's Python Class
  http://code.google.com/edu/languages/google-python-class/
- Python
  http://www.python.org/
- NumPy
  http://numpy.scipy.org/
- Matplotlib
  http://matplotlib.sourceforge.net/
- NetCDF, netcdf4-python
  http://www.unidata.ucar.edu/software/netcdf/
  http://code.google.com/p/netcdf4-python/

Python Basics

- *interpreted* (as opposed to *compiled*) language

→ one does not compile Python code before executing it

- *interactive mode* (advanced calculator)

→ type in python code directly into command line shell

→ gives immediate feedback for each statement

```
>>> print("Hello World!")
Hello World!
```

→ quick help: `help(function)`, `help(module)`, `dir(module)`, `dir(variable)`

```
>>> import math
>>> help(math)
help on module math:
[...]
DESCRIPTION
    This module is always available. It provides access to
    the mathematical functions defined by the C standard.
[...]
```

Python Basics: Indentation

- blocks of code are delimited by indentation levels

→ instruction under a flow-control keyword (`if`, `else`, `while`, `for`) or in the body of a function must be indented

- you can use as many spaces/tabs as you want (but consistently!)
- guarantees some structure of the program and allows to visualise this structure quite easily

→ correct:

```
>>> letters = ['a','b','c']
>>> for s in letters:
>>>     print(s)
```

→ wrong (will throw an exception, `IndentationError: unexpected indent`):

```
>>> letters = ['a','b','c']
>>> for s in letters:
>>> print(s)
```

```
>>> letters = ['a','b','c']
>>>     for s in letters:
>>>         print(s)
```

Python Basics: Types

- *dynamic typing*: an object's type can change during runtime

```
>>> a = 'Hello'
>>> type(a)
<type 'str'>
>>> a = 3
>>> type(a)
<type 'int'>
```

- scalar data types: integer, float (64bit), complex (native!), boolean:

```
>>> b = 3.
>>> type(b)
<type 'float'>
>>> c = 1. + 0.5j
>>> type(c)
<type 'complex'>
>>> d = (3 > 4)
>>> type(d)
<type 'bool'>
```

```
>>> print(a)
3
>>> print(b)
3.0
>>> print(c.real)
1.0
>>> print(c.imag)
0.5
>>> print(d)
False
```

Python Basics: Containers

- lists: ordered collection of objects, that may have different types

```
>>> l = [1, 2, 3, 4, 5]
>>> type(l)
<type 'list'>
```

→ indexing: access individual objects contained in the list (starting at 0)

```
>>> l[2]
3
```

→ counting from the end with negative indices:

```
>>> l[-1]
5
```

→ slicing (sublists): `l[start:stop:stride]` (note: `l[start:stop]` contains the elements with index `i` such that `start <= i < stop`)

```
>>> l[2:4]
[3, 4]
```

```
>>> l[3:]
[4, 5]
```

```
>>> l[::2]
[1, 3, 5]
```

Python Basics: Containers

→ append elements:

```
>>> l.append(6)
>>> l
[1, 2, 3, 4, 5, 6]
```

→ extent by another list:

```
>>> l.extend([7,8])
>>> l
[1, 2, 3, 4, 5, 6, 7, 8]
```

→ reverse order:

```
>>> l.reverse()
>>> l
[8, 7, 6, 5, 4, 3, 2, 1]
```

→ sort:

```
>>> l.sort()
>>> l
[1, 2, 3, 4, 5, 6, 7, 8]
```

Python Basics: Containers

→ pop last element:

```
>>> l.pop()
8
>>> l
[1, 2, 3, 4, 5, 6, 7]
```

→ remove some element:

```
>>> l.remove(4)
>>> l
[1, 2, 3, 5, 6, 7]
```

→ elements can be of different type:

```
>>> l.append('s')
>>> l
[1, 2, 3, 5, 6, 7, 's']
```

→ number of elements:

```
>>> len(l)
7
```

Python Basics: Containers

- tuples: immutable lists (lets you create simple data structures)

→ written between brackets, e.g. `u = (x,y)`:

```
>>> u = (0,2)
>>> u
(0, 2)
```

or just separated by commas:

```
>>> t = 12345, 67890, 'hello', u
>>> t
(12345, 67890, 'hello', (0, 2))
```

→ tuples are unpacked using list syntax:

```
>>> u[1]
2
```

→ elements of a tuple cannot be assigned independently:

```
>>> u[1] = 3
TypeError: 'tuple' object does not support item assignment
```

Python Basics: Containers

- sets: unordered collection with no duplicate elements

```
>>> basket = ['apple', 'orange', 'apple', \
              'pear', 'orange', 'banana']
>>> fruit = set(basket)
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
```

→ eliminates duplicate entries

→ membership testing:

```
>>> 'orange' in fruit
True
>>> 'melon' in fruit
False
```

Python Basics: Containers

→ create sets of characters:

```
>>> s = set('abcd')
>>> s
set(['a', 'c', 'b', 'd'])
>>> t = set(('e', 'f', 'g'))
>>> t
set(['e', 'g', 'f'])
```

→ add elements:

```
>>> s.add('a')
>>> s
set(['a', 'c', 'b', 'd'])
>>> t.update(['a', 'b'])
>>> t
set(['a', 'b', 'e', 'g', 'f'])
```

→ remove an element:

```
>>> t.remove('g')
>>> t
set(['a', 'b', 'e', 'f'])
```

Python Basics: Containers

- support for mathematical operations like union, intersection, difference, and symmetric difference
- letters in s but not in t:

```
>>> s-t
set(['c', 'd'])
```

- letters in either s or t:

```
>>> s | t
set(['a', 'c', 'b', 'e', 'd', 'f'])
```

- letters in both s and t:

```
>>> s & t
set(['a', 'b'])
```

- letters in t or s but not both:

```
>>> s ^ t
set(['c', 'e', 'd', 'f'])
```

Python Basics: Containers

- dictionaries: hash table that maps keys to values (unordered)

```
>>> tel = { 'peter': 5752, 'mary': 5578 }
>>> tel['jane'] = 5915
>>> tel
{'jane': 5915, 'peter': 5752, 'mary': 5578}
>>> tel['mary']
5578
>>> tel.keys()
['jane', 'peter', 'mary']
>>> tel.values()
[5915, 5752, 5578]
>>> 'peter' in tel
True
```

→ dictionaries can have keys and values with different types:

```
>>> d = { 'a': 1, 'b': 2, 3:'hello' }
>>> d
{'a': 1, 3: 'hello', 'b': 2}
```

Basic Python: Control Flow

- `if`/`elif`/`else`:

```
>>> if 2**2 == 4:
...     print('Obvious!')
...
Obvious!
```

→ no brackets enclosing the conditional statement

```
>>> a = 10
>>> if a == 1:
...     print('one')
... elif a == 2:
...     print('two')
... else:
...     print('a lot')
...
a lot
```

Basic Python: Control Flow

- `for`/`range`

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

→ `range([start,] stop[, step])` creates lists of arithmetic progressions

→ the arguments must be plain integers

→ the `start` argument defaults to `0`, the `step` argument defaults to `1`

- `while`

```
>>> z = 1. + 1.j
>>> while abs(z) < 100:
...     z = z**2 + 1.
...
>>> z
(-134+352j)
```

Basic Python: Control Flow

- break

```
>>> z = 1. + 1.j
>>> while abs(z) < 100:
...     if abs(z.imag) > 10:
...         break
...     z = z**2 + 1.
...
>>> z
(-11-16j)
```

- continue

```
>>> a = [1, 0, 2]
>>> for element in a:
...     if element == 0:
...         continue
...     print 1. / element
...
1.0
0.5
```

Basic Python: Control Flow

- iterate over *any* sequence (list, set, dictionary, string, file, ...)

```
>>> for word in ('cool', 'powerful', 'readable'):
...     print('Python is %s' % word)
...
Python is cool
Python is powerful
Python is readable
```

- iterate over a dictionary:

```
>>> d = {'a':1, 'b': 2, 'c': 1j }
>>> for key, val in d.iteritems():
...     print('Key: %s has value: %s' % (key,val) )
...
Key: a has value: 1
Key: c has value: 1j
Key: b has value: 2
```

→ `for key in d:` only gives you the keys, not the values
(but of course, values can be accessed via `d[key]` within the loop)

Basic Python: Functions

- defining functions:

```
>>> def test():
...     print('This is a test!')
...
>>> test()
This is a test!
```

- returning a value:

```
>>> def disk_area(radius):
...     return 3.14 * radius * radius
...
>>> disk_area(1.5)
7.0649999999999995
```

→ by default, functions return `None`

→ all values that the function should return... must be returned

Basic Python: Functions

- multiple values can be returned as a sequence:

```
>>> def integer_divide(x,y):
...     integer_part = x/y
...     the_rest = x%y
...     return integer_part, the_rest
...
>>> int, rest = integer_divide(10,3)
>>> print(int)
3
>>> print(rest)
1
```

- `map(f,s)` applies the function `f` to the elements of `s` and returns a list:

```
>>> def square(x):
...     return x*x;
>>> l = [1, 2, 3, 4]
>>> map(square,l)
[1, 4, 9, 16]
```

Basic Python: Functions

- optional parameters (*keyword* or *named arguments*):

```
>>> def say_my_name(name='How should I know?'):
...     return name
...
>>> say_my_name()
'How should I know?'
>>> say_my_name('Mike')
'Mike'
```

→ keyword arguments allow you to specify *default values*

→ the order of keyword arguments does not matter:

```
>>> def slicer(seq, start=None, stop=None, step=None):
...     return seq[start:stop:step]
...
>>> l = [1,2,3,4,5,6]
>>> slicer(l, start=1, stop=4)
[2, 3, 4]
>>> slicer(l, step=2, stop=5, start=1)
[2, 4]
```

Basic Python: Functions

- functions are objects

→ functions can be assigned to a variable:

```
>>> func = test
>>> func()
This is a test!
```

→ functions can be put into a list (or any collection):

```
>>> l = [1, 'a', test]
>>> l
[1, 'a', <function test at 0x10698e050>]
>>> l[2]()
This is a test!
```

→ functions can be passed as an argument to another function:

```
>>> def call_func(function):
...     function()
...
>>> call_func(test)
This is a test!
```

Basic Python: Functions

- docstrings: documentation about what the function does and it's parameters

```
>>> def funcname(params):
...     """
...     Concise one-line sentence describing the function.
...
...     Extended summary, can contain multiple paragraphs.
...     """
...     # function body
...     pass
...
>>> help(funcname)
funcname(params)
    Concise one-line sentence describing the function.

    Extended summary, can contain multiple paragraphs.
```

→ the docstring conventions webpage documents the semantics and conventions associated with Python docstrings

Basic Python: Scripts

- you do not want to type everything into the shell, especially for longer sets of instructions

→ write a text file containing code using an editor with syntax highlighting

→ execute by starting the interpreter with the filename as argument

- script: file with extension .py containing a sequence of instructions that are executed each time the script is called

test.py

```python
message = "How are you?"
for word in message.split():
    print(word)
```

→ execute the script on the shell:

```
> python test.py
How
are
you?
```

Basic Python: Modules

- put definitions in a file and use them as a *module* in other scripts

div.py

```
def integer_divide(x,y):
    integer_part = x/y
    the_rest = x%y
    return integer_part, the_rest
```

→ importing the module gives access to its objects via `module.object`:

```
import div
int, rest = div.integer_divide(10,3)
```

→ import objects from modules into the main namespace:

```
from div import integer_divide
int, rest = integer_divide(10,3)
```

→ prescribe a shortcut for the module:

```
import div as d
int, rest = d.integer_divide(10,3)
```

Basic Python: Modules

- create modules if you want to write larger and better organised programs

→ define objects (variables, functions, classes) that you want to reuse several times in your own modules

demo.py

```
"""A demo module."""

def print_b():
    """Prints b."""
    print 'b'

def print_a():
    """Prints a."""
    print 'a'

print("Module demo got loaded!")

c = 2
d = 2
```

Basic Python: Modules

→ import the demo module in another script and call its functions:

demo_test.py

```python
import demo

demo.print_a()
demo.print_b()

print(demo.c)
print(demo.d)
```

```
> python demo_test.py
Module demo got loaded!
a
b
2
2
```

Basic Python: Modules

→ take a look at the documentation:

```
>>> help(demo)
Help on module demo:

NAME
    demo - A demo module.

FILE
    /Users/mkraus/Lehre/CompPhys/Python/code/demo.py

FUNCTIONS
    print_a()
        Prints a.

    print_b()
        Prints b.

DATA
    c = 2
    d = 2
```

Basic Python: Modules

- you can define a `__main__` function that is executed if the module is executed as a script but not when the module is imported:

demo2.py

```python
"""Another demo module."""

def print_a():
    """Prints a."""
    print 'a'

if __name__ == '__main__':
    print_a()
```

→ execute the module on the shell:

```
> python demo2.py
a
```

→ load the module in interactive mode:

```
>>> import demo2
>>>
```

Basic Python: Modules

- a directory that contains many modules is called a *package*

→ a package is a module with submodules (which can have submodules themselves, etc.)

→ a special file called `__init__.py` (might be empty) tells Python that the directory is a Python package, from which modules can be imported

→ a package might be organised as follows:

```
Graphics/
    __init__.py
    Primitive/
        __init__.py
        lines.py
        text.py
        ...
    Graph2D/
        __init__.py
        plot2d.py
        ...
    Graph3D/
        __init__.py
        plot3d.py
        ...
    ....
```

Basic Python: Input/Output

- reading and writing of strings from and to files

→ write to a file:

```
>>> f = open('tempfile', 'w')
>>> type(f)
<type 'file'>
>>> f.write('This is a test \nand another test')
>>> f.close()
```

→ read from a file:

```
>>> f = open('tempfile', 'r')
>>> s = f.read()
>>> print(s)
This is a test
and another test
```

Basic Python: Input/Output

→ iterate over a file:

```
>>> f = open('tempfile', 'r')
>>> for line in f:
...     print(line)
...
This is a test

and another test
```

→ file modes:

- `r` : read-only
- `w` : write-only (create a new file or overwrite existing file)
- `a` : append
- `r+`: read and write

Basic Python: Exceptions

- exceptions are raised by different kinds of errors arising when executing Python code and normally terminate program execution:

```
>>> 1/0
ZeroDivisionError: integer division or modulo by zero

>>> 1 + 'e'
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> d = { 1:1, 2:2 }
>>> d[3]
KeyError: 3

>>> l = [1,2,3]
>>> l[3]
IndexError: list index out of range

>>> l.foobar
AttributeError: 'list' object has no attribute 'foobar'
```

→ you may (and often should) catch errors or define custom error types

Basic Python: Exceptions

- catching exceptions: `try`/`except`

exceptions1.py

```
while True:
    try:
        x = int(raw_input('Please enter a number: '))
        break
    except ValueError:
        print('That was no valid number.  Try again...')
```

```
> python exceptions1.py
Please enter a number: a
That was no valid number.  Try again...
Please enter a number: 1
```

Basic Python: Exceptions

- catching exceptions: `try`/`finally`

exceptions2.py

```
try:
    x = int(raw_input('Please enter a number: '))
finally:
    print('Thank you for your input.')
```

```
> python exceptions2.py
Please enter a number: 1
Thank you for your input.
```

```
> python exceptions2.py
Please enter a number: c
Thank you for your input.
ValueError: invalid literal for int() with base 10: 'c'
```

→ important for resource management (e.g. closing a file)

Basic Python: Exceptions

- raising exceptions to pass messages between parts of the code

exceptions3.py

```
def achilles_arrow(x):
    if abs(x - 1) < 1e-3:
        raise StopIteration
    x = 1 - (1-x)/2.
    return x

x = 0
while True:
    try:
        x = achilles_arrow(x)
    except StopIteration:
        print('Iteration has stopped')
        break
print("Result: " + str(x))
```

```
> python exceptions3.py
Iteration has stopped
Result: 0.9990234375
```

Basic Python: Exceptions

- reraise an exception:

exceptions4.py

```
while True:
    try:
        x = int(raw_input('Please enter a number: '))
        break
    except ValueError, e:
        print('That was no valid number.')
        raise e
```

```
> python exceptions4.py
Please enter a number: a
That was no valid number.
ValueError: invalid literal for int() with base 10: 'a'

> python exceptions4.py
Please enter a number: 1
```

Basic Python: Object Oriented Programming (OOP)

- python supports object oriented programming

→ helps to organise your code

→ simplifies reuse of code in similar contexts

- objects consist of internal data and methods that perform various kinds of operations on that data

- example: harmonic oscillator

→ attributes: mass `m`, spring constant `k`, initial displacement `x0`

→ methods: `total_energy()`, `period()`, `frequency()`, `acceleration(x)`, `velocity(x)`, `kinetic_energy(x)`, ...

Basic Python: Object Oriented Programming (OOP)

- classes gather custom functions and variables

student.py

```
class Student(object):
    def __init__(self, name):
        self.name = name
    def set_age(self, age):
        self.age = age
    def set_major(self, major):
        self.major = major
    def info(self):
        print('%s is a student in %s and %d years old.' \
                 % (self.name, self.major, self.age) )
```

```
>>> from student import Student
>>> anna = Student('anna')
>>> anna.set_age(21)
>>> anna.set_major('physics')
>>> anna.info()
anna is a student in physics and 21 years old.
```

Basic Python: Object Oriented Programming (OOP)

- inheritance allows to derive new classes from old ones that have the same methods and attributes but might be extended by new ones

```
>>> class MasterStudent(Student):
...     internship = 'mandatory, from March to June'
...
>>> james = MasterStudent('james')
>>> james.internship
'mandatory, from March to June'
```

→ you can access all methods and attributes defined in `Student` as well:

```
>>> james.set_age(23)
>>> james.age
23
>>> james.set_major('math')
>>> james.info()
james is a student in math and 23 years old.
```

Outlook: Scientific Computing and Plotting

- NumPy: fundamental package for scientific computing with Python
  - powerful N-dimensional array object and various derived objects (matrices)
  - routines for fast operations on arrays (selecting, sorting, shape manipulation, mathematical, I/O, ...)
  - modules for linear algebra, statistics, Fourier transforms, random numbers
  - it's fast: most routines are optimised, pre-compiled C code
- SciPy: numerical routines, data manipulation and analysis
- Matplotlib: 2D plotting
  - allows generation of plots, histograms, power spectra, bar charts, error charts, scatter plots, etc., with just a few lines of code
  - full control of line styles, font properties, axes properties, etc., via an object oriented interface or via a set of functions similar to MATLAB
- MayaVi: 3D plotting and interactive scientific data visualisation
- HDF5/NetCDF: self-describing and efficient storage of scientific data