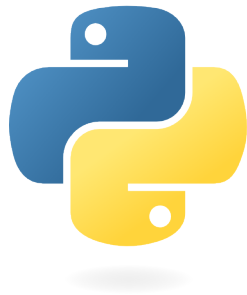


Programming in Python – An example-driven introduction



PyWorkshop - June 09 2008

Jochen Martin Eppler

Honda Research Institute Europe GmbH
63073 Offenbach/Main, Germany



- The Python programming language
- Basic data types
- Control structures
- Writing functions
- Object oriented programming
- Functional programming
- File system operations and file access
- Literature

- Commands: **import**
- Type names: ***long***
- Module names: **math**
- Other code: **x = True**
- Code blocks:

```
import numpy
print numpy.__version__
1.0.4
```

- Shell code

```
jochen@winston:~$ python --version
Python 2.5.2
```

- If questions occur, feel free to ask right away!

- Modern object-oriented programming language
- Very clear, readable syntax, thus easy to learn
- Extensive standard library: Batteries included!
- Third party extensions for virtually every task
 - Graphical user interfaces
 - Scientific computing
 - Database manipulation
- Free specification: Everyone can implement the language
 - CPython is the reference implementation

- Ipython
 - An interactive shell for with enhanced introspection, code highlighting and tab completion
 - Built on top of an existing Python interpreter
 - Ipython1 provides a distributed execution environment
- Jython
 - Another Python interpreter, written in Java instead of C
- IronPython
 - A Python implementation for the .NET framework
 - Integrates nicely with other .NET languages

- Just code, no clutter!
- No curly braces: Indentation structures the code

```
if x > 0:  
    print 'Hello World!'  
    update_values(x)  
y = math.exp(-t / tau)
```

- Other whitespace has no meaning
- Commands end with the line, but can be extended using \
- Multiple statements on one line can be separated by ;
- Comments start with a #
- Many editors have good support for the Python syntax

- Python programs can be run either interactively or as scripts stored in a file
- The interpreter is started by calling **python**

```
jochen@winston:~$ python
>>> print 'Hello world!'
Hello world!
>>> x = 3
>>> print x + 5
8
```

- Scripts are supplied as arguments to the interpreter

```
jochen@winston:~$ python hello_world.py
Hello world!
```

- **python -i** gives an interactive prompt after the script

- The default extension for Python files is **.py**
- Scripts start with the interpreter they should use

```
#!/usr/bin/env python
print 'Hello world!'
```

- Optionall, you can specify the file encoding in line 2

```
# -*- coding: utf-8 -*-
print 'Total: 42 €'
```

- Scripts have to be executable: **chmod u+x <file>**
- Execute scripts as standalone programs

```
jochen@winston:~$ ./hello_world.py
Hello world!
```

- The standard library is split into modules

```
import sys, os
print sys.platform
linux2
```

- Members can be imported into the global namespace

```
from sys import api_version
from os import *
print api_version
1013
```

- New names can be given to imported modules

```
import numpy.linalg as peter
print peter.norm([1, 1, 1])
1.73205080757
```

- No type declarations, just variable definitions

```
f = 2.5
print f, type(f)
2.5 <type 'float'>
```

```
f = 3
print f, type(f)
3 <type 'int'>
```

```
f = 'words'
print f, type(f)
words <type 'str'>
```

```
f = [1, 2, 3]
print f, type(f)
[1, 2, 3] <type 'list'>
```

- Nonetheless type-safe

```
i = 2.5 + 'words' # Error!
```

- Types can be converted via the new type's constructor

```
i = int(2.5)
print i, type(i)
2 <type 'int'>
```

- Plain integers (***int***) have at least 32 bit
- Long integers (***long***) have *unlimited precision*
- Floating point numbers (***float***) correspond to C doubles
- Complex numbers (***complex***) consist of two ***floats***

```
i = 5  
print type(i)  
<type 'int'>
```

```
j = i * 2**64  
print type(j)  
<type 'long'>
```

```
f = float(i)  
print type(f)  
<type 'float'>
```

```
c = complex(i, f)  
print type(c)  
<type 'complex'>
```

- Hexadecimal notation: $0xFF = 255$
- Octal notation: $01337 = 735$

- Standard math is part of the interpreter
 - Operators: `+`, `-`, `*`, `/`, `**`, `%`, `//`
 - Assignment: `=`, `+=`, `-=`, `*=`, `/=`, `**=`, `%=`, `//=`
- Don't be surprised by calculation results!

```
x = 1 / 10
```

```
y = 1 / 10.0
```

```
print x, type(x)
```

```
0 <type 'int'>
```

```
print y, type(y)
```

```
0.1 <type 'float'>
```

```
x = 1.0//2.0
```

```
y = 1//2
```

```
print x, type(x)
```

```
0.0 <type 'float'>
```

```
print y, type(y)
```

```
0 <type 'int'>
```

- Higher mathematical functions are provided by [math](#)
 - **`exp`**, **`sin`**, **`cos`**, **`tan`**, **`floor`**, **`ceil`**, **`log`**, ...

- Booleans (*bool*): **True**, **False**
- Logical Boolean operators: **not**, **and**, **or**
- Bit-wise Boolean operators: **&**, **|**, **^**
- Comparison: **==**, **!=**, **<**, **<=**, **>**, **>=**, **is** [**not**], [**not**] **in**

```
i = 1
```

```
j = 1
```

```
print i == j
```

```
True
```

```
print i is j
```

```
True
```

```
c = 2
```

```
d = 2.0
```

```
print c == d
```

```
True
```

```
print c is d
```

```
False
```

- Numbers **!= 0** are regarded as **True**
- Use **None** (of type *NoneType*) for uninitialized variables

- *list* is the most common sequence type

```
lst = [4, 3, 2.0, 0, 'peter']
lst[3] = 1
print lst
[4, 3, 2.0, 1, 'peter']

print len(lst)
4
```

- A *tuple* is basically an immutable list

```
t = (1, 2, 'sam')
print t[0]
1

t[2] = 0      # Error!

t2 = (1)
t3 = (1,)
print t2, type(t2)
1 <type 'int'>
print t3, type(t3)
(1,) <type 'tuple'>
```

- Create lists from other lists

```
lst = [1, 2, 3]
lst2 = [x**2 for x in lst]
print lst2
[1, 4, 9]
```

- Conditions can be applied

```
lst3 = [(x, x**2) for x in lst2 if x < 5]
print lst3
[(1, 1), (4, 16)]
```

- Multiple for or if clauses can be used

```
lst4 = [x * y[0] for x in lst2 for y in lst3]
print lst4
[1, 4, 4, 16, 9, 36]
```

- Homogeneous arrays are available in the **array** module
- They are more efficient than *lists* or *tuples*
- A type code is used at creation time *c*, *i*, *l*, *f*, *d*, ...

```
from array import array

a = array('i', [1, 2, 3])
print type(a)
array('i')

b = array('d', [1.0, 2.0, 3.0])
lst = list(b)
print b, lst
array('d', [1.0, 2.0, 3.0]) [1.0, 2.0, 3.0]
print type(lst)
<type 'list'>
```

- Modifying the order of elements

```
lst = [4, 2, 3, 1]
lst.sort()
print lst
[1, 2, 3, 4]

lst.reverse()
print lst
[4, 3, 2, 1]
```

- Element removal

```
lst.remove(4)
print lst
[3, 2, 1]
```

- Concatenation of *lists*

```
lst2 = [4, 5]
lst += lst2
print lst
[3, 2, 1, 4, 5]

lst.extend(lst2)
print lst
[3, 2, 1, 4, 5]
```

- **lists** with identical elements

```
lst = [1] * 4
print lst
[1, 1, 1, 1]
```

```
lst2 = [(1, 2)] * 3
print lst2
[(1, 2), (1, 2), (1, 2)]
```

- Ranges of numbers

```
r = range(5)
print r
[0, 1, 2, 3, 4]
```

```
s = range(1, 5)
print s
[1, 2, 3, 4]
```

```
t = range(0, 20, 2)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- **tuple** creation

```
u = 1, 2, 3
print u, type(u)
(1, 2, 3) <type 'tuple'>
```

- More-dimensional lists are lists of lists of lists of ...
- Selections can be extracted

```
lst = [1, 2, 3, 4, 5, 6]
```

```
print lst[:]           print lst[1:2]
[1, 2, 3, 4, 5, 6]    [2]
print lst[1:]         print lst[: -1]
[2, 3, 4, 5, 6]      [1, 2, 3, 4, 5]
print lst[:1]         print lst[-3: -1]
[1]                  [4, 5]
```

- *lists* used as stack

```
lst.append(7)
print lst           print lst.pop(), lst
[1, 2, 3, 4, 5, 6, 7] 7 [1, 2, 3, 4, 5, 6]
```

- Sequence indices start at 0

- **set** is a set in the mathematical sense

```
l = [1, 2, 3, 4, 4]
```

```
s = set(l)
```

```
t = set([1, 2, 5])
```

```
u = set([1, 2])
```

```
print l, s
```

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

```
print s.union(t)
```

```
set([1, 2, 3, 4, 5])
```

```
print s.intersection(t)
```

```
set([1, 2])
```

```
print s.difference(t)
```

```
set([3, 4])
```

```
print s.issuperset(u)
```

```
True
```

```
print s.symmetric_difference(t)
```

```
set([3, 4, 5])
```

- Text has to be delimited by quotes

```
s1 = "This is a 'string'"
s2 = 'This is a "string"'
s3 = "This is a \"string\""

print s1 == s2          print s2 == s3
False                  True
```

- Multi-line strings

```
s4 = """This is a very long string, which even
extends over several lines."""
```

- Raw strings don't resolve special characters

```
s5 = r'The tab (\t) in here is not resolved'
print s5
The tab (\t) in here is not resolved
```

- split

```
str = 'This is a string'

print str.split(' ')
['This', 'is', 'a', 'string']
```

- find

```
print str.find('string')
10
```

- replace

```
print str.replace('string', 'sentence')
This is a sentence
print str
This is a string
```

- ...

- ***dict*** implements an associative array

```
d = {'peter' : '0761-4760498',  
     'sam'   : '07436-229' }
```

```
print d.keys()  
['peter', 'sam']
```

```
print d.values()  
['0761-4760498', '07436-229']
```

```
print d['peter']  
0761-4760498
```

- Dictionaries are not ordered!
- Anything except ***list*** and ***dict*** can be used as key

- Assignment of mutable objects assigns a reference

```
i = 1          lst = [1, 2, 3]
j = i         m = lst
print i, j    print lst, m
1 1          [1, 2, 3] [1, 2, 3]
j = 2        m[0] = 3
print i, j    print lst, m
1 2          [3, 2, 3] [3, 2, 3]
```

- `copy` (`deepcopy` for nested objects) makes a real copy

```
from copy import copy
n = [1, 2, 3]; o = copy(n)
o[0] = 3
print n, o
[1, 2, 3] [3, 2, 3]
```

- The `is` operator checks object equality

- `print` writes (formatted) text to the screen

```
i = 5; f = 2.5; g = 4.0; h = 2**64; s = 'VAT'

print 'Number of things:', i
Number of things: 5
print 'We have %i things' % i
We have 5 things
print '%i * %f = %f' % (i, f, i*f)
5 * 2.500000 = 12.500000
print '%i * %.2f = %.2f' % (i, f, i*f)
5 * 2.50 = 12.50
print '%f vs. %g vs. %g' % (f, g, h)
2.500000 vs. 4 vs. 1.84467e+19
print '%4i vs. %4i' % (i, i)
    5 vs. 0005
print '%s is %i%%' % (s, i)
VAT is 5%
```

- if for flow control

```
if x > 0:
    y = 5

if x in [1, 2, 3]:
    print 'x is valid!'

if 0 <= x < 42:
    print 'x is from the interval [0, 42)'
```

- elif to define branches

- else to catch remaining cases

```
if object == 'circle':
    print 0
elif object == 'rectangle':
    print 4
else:
    print 'unknown number of vertices'
```

- for loops assign a variable in each iteration

```
for i in l:                for j in xrange(5):
    print i + 1,           print j + 1,
1 2 3 4 5                 0 1 2 3 4

lst = ['a', 'b', 'c', 'd', 'e']
for m, n in enumerate(lst):
    print '%i:%s,' % (m, n),
0:a, 1:b, 2:c, 3:d, 4:e,

d = {'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4}

for k in d:                for v in d.values():
    print k,               print v,
a c b d                   1 3 2 4
```

- Note the order of elements for the dictionary!

- while loops run as long as the condition is **True**

```
x = 0
while x < 4:
    print x,
    x += 1
else:
    print 'done!'
```

0 1 2 3 done!

```
x = 0
while True:
    x += 1,
    if x == 2:
        continue
    print x - 1,
    if x >= 4:
        break
```

0 2 3

- **break** leaves the loop without evaluating the else branch
- **continue** will skip the rest and re-evaluate the condition

- Re-occurring code can be put into functions

```
def fib(n):  
    """Print the Fibonacci series up to n - 1"""  
    a, b = 0, 1  
    while b < n:  
        print b,  
        a, b = b, a + b
```

```
fib(1000)
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

- Functions can be copied by assignment

```
f = fib
```

```
f(1000)
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

- Arguments can have default values

```
def func(a, b = 1, c = 2)
    return b, a + b + c

print func(3, c = 3)
(1, 8)
```

- Arguments can be taken from *tuples* and *dicts*

```
t = (4, 5, 6); d = {'a': 7, 'b': 8, 'c': 9}

print func(*t)           print func(**d)
(5, 15)                 (8, 24)
```

- Unpacking *tuples*

```
m, n = func(2)
print '((%i %i))' % (n, m)
((4 1))
```

- Additional arguments are allowed

```
def func1(a, b = 2, *args):  
    print args, type(args)
```

```
def func2(a, b = 2, **kwargs):  
    print kwargs, type(kwargs)
```

```
func1(1, 2, 3)  
(3,) <type 'tuple'>
```

```
func2(1, 2, c = 3)  
{'c': 3} <type 'dict'>
```

- Arguments are passed by assignment, not by copy

```
def func3(a):  
    a['a'] = 0
```

```
def func4(a):  
    a = 5
```

```
d = {'a': 1, 'b': 2}  
func3(d); print d  
{'a': 0, 'b': 2}
```

```
i = 15  
func4(i); print i  
15
```

- Creating one result at a time

```
def fib_gen(n):  
    """Return one Fibonacci element at a time"""  
    a, b = 0, 1  
    while b < n:  
        yield b  
        a, b = b, a + b  
  
for x in fib_gen(1000):  
    print x,
```

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

- Generators return objects called *iterators*
- xrange and enumerate are built-in examples

- Classes can encapsulate variables and functions

```
class shape:  
    """A generic base class for shapes"""  
  
    self.area = None  
    self.perimeter = None
```

- Single or multiple inheritance

```
class rect(shape):  
  
    def __init__(self, a, b)  
        """Initialize the member variables"""  
  
        area = a * b  
        self.area = area  
        self.perimeter = 2 * a + 2 * b
```

- Arithmetic: `__add__`, `__sub__`, `__mul__`, `__div__`, ...

```
class A:  
  
    def __init__(self, member):  
        self.member = member  
  
    def __add__(self, other):  
        return A(self.member + other.member)  
  
a = A(7); b = A(3)  
  
c = a + b                a += b  
print c.member          print a.member  
10                      10
```

- If `+` is used, Python calls `__add__` internally

- Comparison: `__eq__`, `__ne__`, `__lt__`, `__le__`, ...

```
class A:  
  
    def __init__(self, member):  
        self.member = member  
  
    def __eq__(self, other):  
        return self.member == other.member  
  
a = A(7); b = A(6); c = A(7)  
  
print a == b, a == c  
False True
```

- `__cmp__` is used if rich comparison is not defined, it must return -1 if `self < other`, 0 if `self == other`, +1 if `self > other`

- Printing objects

```
class A:

    def __init__(self, member):
        self.member = member

    def __str__(self):
        return 'A, member is %i' % self.member

a = A(7)
print a
A, member is 7
```

- `__hash__` must return an integer, which is equal for objects that compare equal
- Objects defining `__hash__` can be used as key in **dicts**

- All errors are derived from class **Exception**

```
try:
    x = int(num)
except ValueError, err:
    print type(err)
    print 'num cannot be converted to int'
    x = None
except:
    print 'An unknown error occurred'
    raise Exception('Something went wrong!')
```

- Own exceptions are created by inheritance

```
Class MyException(Exception):
    def __init__(self, msg):
        Exception.__init__(self, msg)
```

- Nameless functions for functional programming

```
def f(x): return x      <=>      lambda x: x
```

- Can be turned into named functions by assignment

```
f = lambda x, y: x + y  
print f(1, 2)  
3
```

- Function generators

```
def func_gen(a):  
    return lambda x: x**a  
  
pow2 = func_gen(2)      pow3 = func_gen(3)  
print pow2(2)          print pow3(2)  
4                      8
```

- map

```
l1 = [1, 2, 3, 4]; l2 = ['a', 'b', 'c', 'd']  
  
print map(lambda x: x + 2, l1)  
[3, 4, 5, 6]
```

- reduce

```
print reduce(lambda x, y: x + y, l1)  
10
```

- filter

```
print filter(lambda x: bool(x > 2), l1)  
[3, 4]
```

- zip

```
print zip(l1, l2)  
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

- Access modes: **r**, **w**, **a**

```
f = file('file.dat', 'w')  
  
f.write('This is the first line\n')  
f.write('Now a number: ' + str(23) + '\n')  
f.close()
```

- Reading

```
f = file('file.dat', 'r')  
  
while True:                                for l in f:  
    l = f.readline()                        print l,  
    if l == '': break                         
    print l,  
  
f.close()
```

- Storing complete data structures

```
import pickle

f = file('file.dat', 'w')
d = {'a': [1, {'b': 2}]}

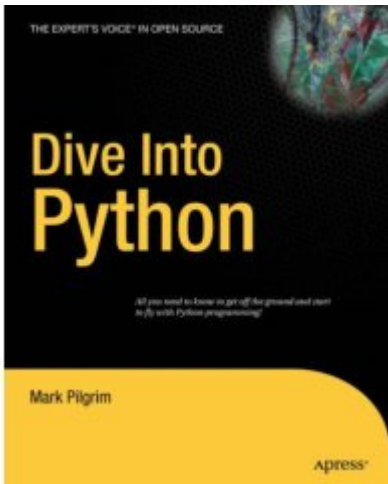
pickle.dump(d, f)          print d == d2
d2 = pickle.load(f)       True
```

- shelve

```
import shelve

d = shelve.open('file.dat')
d['a'] = 1

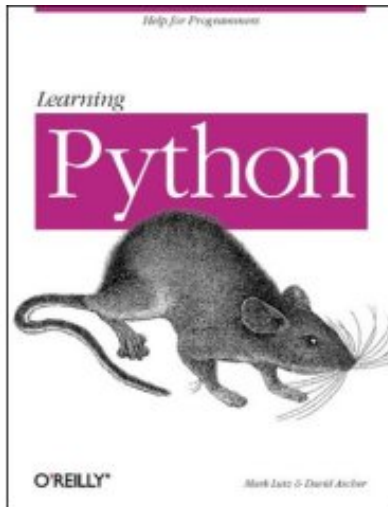
print d                    d.close()
{'a': 1}
```



- Dive Into Python
Second edition

Mark Pilgrim

ISBN: 978-1590593561



- Learning Python
Third edition

Mark Lutz

ISBN: 978-0596513986

- The Python documentation index
<http://docs.python.org>
- Python library reference
<http://docs.python.org/lib>
- Dive into Python
<http://www.diveintopython.org>
- Activestate Python cookbook
<http://aspn.activestate.com/ASPN/Cookbook/Python>