



Python Programming Language

- Full-featured, easy to learn, matured
- Efficient high-level data structures
- Simple but effective approach to OOP
- Short and elegant syntax
- Big library of standard objects and modules
- Documentation (read!!!)
 - Python Web site: <http://www.python.org>
 - Library Reference: <http://docs.python.org/lib/lib.html>
 - Language Reference: <http://docs.python.org/ref/ref.html>



Python License

- Python is a property of the Python Software Foundation (PSF) - a non-profit organization
- Python is distributed under an OSI-approved open source license that makes it free to use, even for commercial products
- You can sell a product written in Python or a product that embeds the Python interpreter. No licensing fees need to be paid for such usage.
- Distributing binary-only versions of Python, modified or not, is allowed. There is no requirement to release any of your source code (as in GPL).



Python Interpreter

```
Python 2.5.1 (r251:54863, May 18 2007, 16:56:43)
[GCC 3.4.4 (cygming special, gdc 0.12, using dmd 0.125)] on cygwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> 2+2
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
>>> x = y = z = 0 # Zero x, y and z
>>> x
0
>>> 7.0 / 2
3.5
>>> (1+2j)/(1+1j)
(1.5+0.5j)
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```



Basics of Syntax

- Command are separated by ";" or newline
- A comment starts with a hash character "#"
- !!! Leading whitespace (spaces and tabs) indicate indentation level of the line
- !!! Indentation level of the line is used to determine the grouping of statements
- Except at the beginning line or in string literals, the whitespace characters (space, tab) are used to separate tokens



Python Statements

- `x = 10
print x`
- `x = 10;
print x;`
- `x = 10; print x;`
- `x = 10; print x`
- `x = 10; print \
x`
- `x = \
10; print x`



Indentation Example

- Correct

```
def perm(l):  
    # Compute the list of all permutations of l  
    if len(l) <= 1:  
        return [l]  
  
    r = []  
    for i in range(len(l)):  
        s = l[:i] + l[i+1:]  
        p = perm(s)  
        for x in p:  
            r.append(l[i:i+1] + x)  
  
    return r
```

- Wrong !

```
def perm(l):  
    for i in range(len(l)):  
        s = l[:i] + l[i+1:]  
        p = perm(l[:i] + l[i+1:])  
        for x in p:  
            r.append(l[i:i+1] + x)  
    return r
```

error: first line indented
error: not indented
error: unexpected indent
error: inconsistent indent



Strings

- A sequence within single quotes ('...'), may include double quotes
- A sequence within double quotes ("..."), may include single quotes
- A sequence within triple quotes ('''...''' or ''''...''''), may include single&double quotes and newlines
- Escape sequence possible
- Concatenation with '+' or side-by-side



Strings Example

```
x = "What's your name?"

x = 'Say "Hello"'

x = '''This is
a multiline
string'''

x = 'This is\
a single line\
string'

x = 'What\'s your name?'
x = "Hello World \n"

x = "To jest"+' jeden tekst'
x = "To jest" ' jeden tekst'
x = "To jest"' jeden tekst'
```



Strings Properties

- Strings are immutable - ones created cannot be modified - performance implications

```
x = "Hello"      # let's create a string
y = " world"    # and another

x = x + y       # new x-string is created
```

- Strings are objects of str() class
- Find all string methods by `>>>help(str)`
- Method str() return nice string representation of any object



String Formatting

- Strings can be created with format pattern
- Type spec. %s, %i, %f, %g etc.

```
"format pattern" % (sequence of objects)
```

```
>>> "%s = %s" % ("string", 123)
'string = 123'
>>> "%s = %10s" % ("string", 123)
'string =          123'
>>> "%s = %i" % ("integer", 123)
'integer = 123'
>>> "%s = %e" % ("real", 123)
'real = 1.230000e+02'
>>> "%s = %x" % ("hex", 123)
'hex = 7b'
```



String Slicing

- Any part of a strings can be accessed with index-like ranges
- Negative indexes refer to the end of string
- First element has index 0, the last one -1

```
>>> x = "Hello world"  
>>> x[0]  
'H'  
>>> x[0:4]  
'Hell'  
>>> x[:4]  
'Hell'  
>>> x[:]  
'Hello world'
```

```
>>> x[-1]  
'd'  
>>> x[3:-1]  
'lo worl'  
>>> x[-4:-1]  
'orl'  
>>> x[-4:]  
'orld'
```



Numbers

- Four types: Integer, Long, Float, Complex
- Conversions are done automatically, if possible
- Use `int()`, `float()`, `complex()` for type conversion

```
>>> x=123
>>> type(x)
<type 'int'>
>>> type(12345678901234567)
<type 'long'>
>>> type(123.123)
<type 'float'>
>>> x= 1+2j
>>> type(x)
<type 'complex'>
>>> int(x)
Traceback (most recent call last):
TypeError: can't convert complex to int; use int(abs(z))
>>> x=123
>>> complex(x)
(123+0j)
```



Numbers are objects

- Class methods can be applied to numbers

```
>>>help(int)
class int(object)
| int(x[, base]) -> integer
|
| Convert a string or number to an integer, if possible. A floating point
| argument will be truncated towards zero (this does not include a string
| representation of a floating point number!) When converting a string, use
| the optional base. It is an error to supply a base when converting a
| non-string. If the argument is outside the integer range a long object
| will be returned instead.
|
| Methods defined here:
|
| __abs__(...)
|     x.__abs__() <==> abs(x)
|
| __add__(...)
|     x.__add__(y) <==> x+y
|
| __and__(...)
|     x.__and__(y) <==> x&y
|
| __cmp__(...)
```



Special Types

- None - returned by functions defined not to return anything

```
>>> type(None)
<type 'NoneType'>
```

- True and False - representation of 1 and 0

```
>>> True == 1
True
>>> False == 0
True
>>> True == 2
False
>>> True == -1
False
```



Assignments, Expressions

- Multiple assignments possible
- Standard set of operators and:
 - `**` - power
 - `//` - floor division
 - `is` - object identity check
 - `in` - membership test
- Order of precedence
 - no surprises, but check in the language reference

```
x = y = z = 10
```

```
x, y, z = 1, 2, 3
```

```
a, b = b, a+b
```



Flow Control

- Remember about ":" and body indentation
 - if-elif-else (elif, else - optional)
 - while-else (else - optional)
 - for-in-else (else - optional)
 - break and continue (for loops only)

```
if x!=0:  
    print y/x  
elif y!=0:  
    print x/y  
else:  
    print "Cannot divide"
```

```
while s!="stop":  
    s = raw_input()  
    if s == 'exit':  
        break  
    print s  
else:  
    print "Done with stop"
```



Flow Control

- For-loop iterate over any sequence of objects
- Create numerical sequences with range()
 - `range(i, j)` returns `i..j-1`
 - `range(1, 4) → [1, 2, 3]`
 - `range(4) → [0, 1, 2, 3]` - default start is 0

```
>>> help(range)
```

```
x = 1
for i in range(2, 1001):
    x = x * i
print x
```

```
x = "Hello world"
for i in x:
    print i
```



Data Structures

- Lists - ordered collection of items, mutable
 - [item, item, ...]
- Tuple (*pol. krotka*) - immutable list
 - (item, item, ...)
- Dictionaries - collections of key-value items, not ordered, mutable (keys immutable)
 - {key:value, key:value, ...}
- All structure can have complex (and mutable) elements

```
>>> help(list)
>>> help(tuple)
>>> help(dict)
```



Lists

- Order is always maintained
- Elements can be appended, popped, deleted, inserted, reversed and sorted
- Lists can be sliced, element count starts with 0

```
x = ['apple', 'glum', 'carrot', "orange"]
x.append('banana')
x.remove('orange')
x[1]="plum"
del x[2]
x.append(x.pop(0))
y = x[1:-1]
for i in range(len(x)):
    print x[i][0:]
```



Tuples

- Use tuples to store data that need not to be modified – faster than list
- Items can be anything
- Order is preserved and slicing is possible

```
fruit=["pineapple"]
color=["yellow"]
taste=["sweet"]
x = (fruit,color,taste)
new = ("lime","green","sour")
for i in range(len(new)):
    x[i].append(new[i])
print x
```

```
(['pineapple','lime'], ['yellow','green'], ['sweet','sour'])
```



Dictionaries

- Dictionaries are mutable
- Keys must be immutable (strings, numbers)
- Values can be anything
- Dictionaries are not ordered - no slicing

```
x = {"tom": "tom@bla.bla.bla", "jerry": "jerry@bla.bla.bla"}

for i in x.keys():
    print "Nick: %s, email: %s" % (i, x[i])

for i in x.values():
    print "Email: %s" % (i)
```



Variables are References

- Assignments of variables create references to objects, that are created or already exists
- More than one variable can refer to the same object - modification of one, affects all
- Operations that create objects, set the variables to new objects

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y = x
>>> x.append(0)
>>> print y
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
```

```
>>> x = 10
>>> y = x
>>> x = x + 10
>>> print y
10
>>> print x
20
```



Functions

- def-keyword, identifier, parameter list, colon ":"
- Function body must be indented
- return-keyword is optional (may return a value)
- Parameters can be of any type (not declared)

```
def function(parameter):  
    body
```

```
def max(x,y):  
    if x>y:  
        return x  
    else:  
        return y  
...  
print max(1,7)
```



Default Parameter Values

- Parameters can be default values assigned
- Functions can be called without def.parameters

```
def scale(x, s = 0.01):  
    a = x*s  
    return a  
  
...  
print scale(3)  
print scale(3,1)
```



Documentation String

- Functions (modules and classes) can be assigned a documentation string used by `help(function)`
- Documentation string can be accessed by `__doc__` attribute of the function

```
def max(x,y):  
    """Maximum of two numbers"""  
    if x>y:  
        return x  
    else:  
        return y  
...  
print max.__doc__
```



Scope of Variables

- Variable is visible within the block it was defined in (main or procedure)
- Parameters (if any) are passed by value
- `global`-keyword indicate external variable

```
def f(a):  
    a = 1  
    return
```

```
a = 0  
f(a)  
print a
```

```
def f():  
    global a  
    a = 1  
    return
```

```
a = 0  
f()  
print a
```



Modules

- Modules (`.py`) contain functions and variables
- Modules are included by `import`-keyword
- There exists a large collection of standard modules (too many to mention) - see Python Library Reference
- When a module is imported, its byte-compiled version is created (`.pyc`) to speedup the subsequent references



Using Modules

- After importing the module, all its functions and variables are accessed within its name space

```
import sys

print 'The command line arguments are:'
for i in sys.argv:
    print i
```

- Using `from ... import` allows to use module's functions and variables without name space

```
from sys import argv

print 'The command line arguments are:'
for i in argv:
    print i
```



Detecting Operation Mode

- Modules can be complete programs, used to run on their own or being imported
- Attribute `__name__` is set at run-time to identify the operation mode for the module

```
if __name__ == '__main__':  
    print 'This program is being run by itself.'  
    ... some demonstration of use code ...  
else:  
    print 'The module is being imported'  
    ... appropriate code ....
```



Inspecting Modules

- Built-in `dir()` function displays all module's identifiers: functions, classes, variables

```
>>> import sys
>>> dir(sys)
['_displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_current_frames', '_getframe', 'api_version',
 'argv', 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
 'copyright', 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getcheckinterval', 'getdefaultencoding',
 'getdlopenflags', 'getfilesystemencoding', 'getrecursionlimit', 'getrefcount',
 'hexversion', 'last_type', 'last_value', 'maxint', 'maxunicode', 'meta_path',
 'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix',
 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags', 'setprofile',
 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
 'version', 'version_info', 'warnoptions']
```



Input/Output

- Simplest: print and raw_input
- Object: file
 - read, readline, write, close, ...

```
>>> help(file)
```

```
f = file('file.txt', 'w')
f.write("any data")
f.close()
```

```
f = file('file.txt') # read mode is default
while True:
    line = f.readline()
    if len(line) == 0:
        break
    print line, # comma to avoid automatic line breaks
f.close()
```



Objects Persistence

- Objects can be stored to/restored from files (pickling/unpickling)

```
import pickle as p

file = 'mylist.pyobj' # file to store the object
mylist = ['apple', 'mango', 'carrot']

f = file(file, 'w')
p.dump(mylist, f) # dump the object to a file
f.close()

...
del mylist # remove the mylist from memory

...
f = file(file)
mynewlist = p.load(f) # read into a new Python object
```



GUI Programming

- No default graphical library
- Lots of GUI-libraries with Python interface:
 - PyGtk - Gnome Gtk library
 - PyQt - KDE Qt library
 - WxPython - WxWidgets library
 - TkInter - Tk library
 - (<http://wiki.python.org/moin/GuiProgramming>)



Tkinter

- Simplest approach

```
#!/usr/bin/python
from Tkinter import *

root = Tk()

w = Label(root, text="Hello, world!")
w.pack()

root.mainloop()
```



Tkinter

- OO approach

```
#!/usr/bin/python
from Tkinter import *

class App:
    def __init__(self, master):
        frame = Frame(master)
        frame.pack()
        self.button = Button(frame, text="QUIT", command=frame.quit)
        self.button.pack(side=LEFT)
        self.hi_there = Button(frame, text="Hello", command=self.say_hi)
        self.hi_there.pack(side=LEFT)
    def say_hi(self):
        print "hi there, everyone!"

root = Tk()
app = App(root)
root.mainloop()
```