

Python

What is it?

What is it good for?

Brian Mays

July 2002

Topics

- Motivation
- Language Features
- History
- The Python Language
- Examples
- Summary

Motivation

Advantages:

- Object-oriented
- Free (mostly GPL-compatible)
- Portable
- Powerful
- Extensible
- Easy

Applications:

- System utilities
- Graphic user interface
- Component integration (glue)
- Rapid prototyping
- Internet (CGI, HTML, FTP, XML, etc.)
- Numerical programming
- Database programming

Language Features

- Dynamic typing
- Built-in objects
 - lists, dictionaries, strings
- Built-in tools
 - concatenation, slicing, sorting, mapping
- Library utilities
 - regular-expression matching
 - networking
- Third-party utilities
 - COM, imaging, XML
- Automatic memory management (“garbage collection”)
- Modularity
 - modules, classes, exceptions

History

- Created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI) in the Netherlands (until version 1.2)
- Successor of a language called ABC
- In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI) in Reston, Virginia (until 1.6)
- After 1.6, when Guido left CNRI to work with commercial software developers, CNRI and the Free Software Foundation (FSF) interacted to develop enabling wording changes to the Python license to be GPL-compatible

- In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team (Python 2.0)
- In October, the PythonLabs team moved to Digital Creations
- In 2001, the Python Software Foundation was formed, a non-profit modeled after the Apache Software Foundation
- All releases are “open source”; however, 1.6 to 2.1 are not GPL-compatible

The Python Language

Types:

Integers (C longs)	1234, -24, 0
Long integers (unlimited)	99999999999999L
Floating-point (C doubles)	1.23, 3.14e-10, 4E210, 4.0e+210
Octal and hex	0177, 0x9ff
Complex numbers	3+4j, 3.0+4.0j, 3J
<hr/>	
Strings	' ', "spam's" """ ... """
Raw strings	r"a\b\c"

Lists

```
[], [0, 1, 2, 3]
```

```
['abc', ['def', 'ghi']]
```

Tuples

```
(), (0,), (0, 1, 2, 3)
```

```
('abc', ('def', 'ghi'))
```

Dictionaries

```
{}, {'spam': 2, 'eggs': 3}
```

```
{'food': {'ham': 1, 'egg': 2}}
```

Files

```
input = open('data', 'r')
```

```
S = input.read(N)
```

```
input.close()
```

Operations:

Logical

`x or y, x and y, not x`

Comparison

`x < y, x <= y, x > y, x >= y`

`x == y, x <> y, x != y`

`x is y, x is not y`

`x in y, x not in y`

Bitwise

`x | y, x ^ y, x & y`

`x << y, x >> y, ~x`

Addition/Concatenation

`x + y, x - y`

Multiplication/Repetition

`x * y, x / y, x % y`

Indexing/Slicing

`x[i], x[i:j]`

Qualification, Function calls

`x.y, x(...)`

Truth Tests:

- True = nonzero number or nonempty object
- False = zero number, empty object, or None
- Comparisons return 1 (true) or 0 (false)
- The and and or operators return an operand object

Indexing:

Counts from 0 from the start or the end

```
>>> S = 'spam'           # S is |s|p|a|m|
>>> S[0], S[-2]         # indexing
('s', 'a')
>>> S[1:3], S[1:], S[: -1] # slicing
('pa', 'pam', 'spa')
```

Lists:

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'; L          # index assignment
['spam', 'eggs', 'SPAM!']
>>> L[0:2] = ['eat', 'more']; L # slice assignment
['eat', 'more', 'SPAM!']

>>> L.append('please'); L     # append method
['eat', 'more', 'SPAM!', 'please']
>>> L.sort(); L              # sort method
['SPAM!', 'eat', 'more', 'please']

>>> del L[0]; L              # delete one item
['eat', 'more', 'please']
>>> del L[1:]; L            # delete section
['eat']
```

Dictionaries:

```
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> d2['spam']          # fetch value for key
2
>>> len(d2)            # number of entries
3

>>> d2.has_key('ham') # membership test
1
>>> d2.keys()          # list keys
['eggs', 'spam', 'ham']

>>> d2['ham'] = ['bake', 'fry']; d2 # change entry
{'eggs': 3, 'spam': 2, 'ham': ['bake', 'fry']}
>>> del d2['eggs']; d2      # delete entry
{'spam': 2, 'ham': ['bake', 'fry']}
>>> d2['brunch'] = 'Bacon'; d2 # add new entry
{'brunch': 'Bacon', 'spam': 2, 'ham': ['bake', 'fry']}
```

Statements:

- Line based input
- Indentation matters!
- No brackets
- Colons introduce compound (indented) statements
- Functions/methods don't always return a result (the default is None)
- Assignment is by reference: $a = b \Rightarrow a \text{ is } b$
- Special techniques are needed to copy objects
e.g., `a = b[:]` or `a = b.copy()`

If:

```
if <test1>:                # if test
    <statements1>         # associated block
elif <test2>:              # optional elif's
    <statements2>
else:                      # optional else
    <statements3>
```

While:

```
while <test>:              # loop test
    <statements>          # loop body
    if <test>: break      # exit loop now, skip else
    if <test>: continue  # restart loop
else:                     # optional else
    <statements>        # if we didn't hit a break
```

For:

```
for <target> in <object>:    # assign object items to target
    <statements>            # loop body; use target
    if <test>: break        # exit loop now, skip else
    if <test>: continue    # restart loop
else:
    <statements>           # if we didn't hit a break
```

```
>>> X = 'spam'
>>> for item in X: print item,          # simple iteration
...
s p a m
>>> i = 0
>>> while i < len(X):                  # while iteration
...     print X[i],; i = i + 1
...
s p a m
>>> for i in range(len(X)): print X[i], # manual indexing
...
s p a m
```

Functions:

```
def <name>(arg1, arg2, ..., argN):  
    <statements>  
    return <value>
```

```
>>> def times(x, y): # create and assign function  
...     return x * y # body executed when called  
...  
>>> times(2, 4)      # arguments in parentheses  
8  
>>> times('Ni', 4)  # functions are typeless  
'NiNiNiNi'
```

Scope Rules:

Search **Local** then **Global** then **Built-in**
The **LGB** rule

- Functions are objects
 - Scope rules are evaluated at *compile time* not run time
 - Arguments are passed by assigning objects to local names
 - Variables inside a function can be declared global with the `global` keyword
-

<code>func(value)</code>	Normal argument: matched by position
<code>func(name=value)</code>	Keyword argument: matched by name
<code>def func(name)</code>	Matches by position or name
<code>def func(name=value)</code>	Default argument value
<code>def func(*name)</code>	Remaining positional args in a tuple
<code>def func(**name)</code>	Remaining keyword args in a dictionary

Lambda Expressions:

```
lambda arg1, arg2, ..., argN: <expression>
```

```
>>> def f(x, y, z): return x + y + z
```

```
...
```

```
>>> f = lambda x, y, z: x + y + z    # equivalent declaration
```

Apply and Map:

```
>>> apply(f, (2, 3, 4))
```

```
9
```

```
>>> counters = [1, 2, 3, 4]
```

```
>>> map((lambda x: x + 3), counters) # uses a function expression
```

```
[4, 5, 6, 7]
```

Modules:

- Modules are Python files (.py) or C extensions (.so)
- Module files are namespaces
- Modules are objects
- Modules are byte compiled for efficiency (.pyc)
- Data hiding is a convention

```
>>> import module1                # get module
>>> module1.printer('Hello world!') # qualify to get names
Hello world!
```

```
>>> from module1 import printer    # get an export
>>> printer('Hello world!')       # no need to qualify name
Hello world!
```

```
>>> from module1 import *         # get all exports
```

Classes:

- Classes are objects
- Classes generate multiple instance objects
- Each instance object inherits class attributes and gets its own namespace
- Operators can be overloaded for class objects

```
>>> class FirstClass:                # define a class object
...     def setdata(self, value):     # define class methods
...         self.data = value        # self is an instance
...     def display(self):
...         print self.data          # self.data: per instance
...
>>> x = FirstClass(); y = FirstClass() # each is a new namespace
```

```
>>> x.setdata("King Arthur") # call methods: self is x or y
>>> y.setdata(3.14159)        # = FirstClass.setdata(y, 3.14159)
>>> x.display()              # self.data differs in each
King Arthur
>>> y.display()
3.14159
```

Inheritance:

```
>>> class SecondClass(FirstClass): # inherits setdata
...     def display(self):         # changes display
...         print 'Current value = "%s"' % self.data
...
>>> z = SecondClass()
>>> z.setdata(42)                # setdata found in FirstClass
>>> z.display()                  # finds overridden method in SecondClass
Current value = "42"
```

Operator Overloading:

```
>>> class ThirdClass(SecondClass):
...     def __init__(self, value):    # on "ThirdClass(value)"
...         self.data = value
...     def __add__(self, other):    # on "self + other"
...         return ThirdClass(self.data + other):
...     def __mul__(self, other):    # on "self * other"
...         self.data = self.data * other
...
>>> a = ThirdClass("abc")           # new __init__ called
>>> a.display()                     # inherited method
Current value = "abc"

>>> b = a + 'xyz'                   # makes new instance
>>> b.display()
Current value = "abcxyz"

>>> a * 3                           # changes instance in-place
>>> a.display()
Current value = "abcabcabc"
```

Exceptions:

Provide *error handling, event notification, special-case handling*, and form the basis of *unusual control flows*

```
try:
    <statements>                # run/call actions
except <name>:
    <statements>                # if name raised
except <name>, <data>:
    <statements>                # if name raised, get extra data
else:
    <statements>                # no exception raised
finally:
    <statements>                # always run "on the way out"
```

Exceptions are raised by

```
raise <name>                    # manually trigger an exception
raise <name>, <data>            # pass extra data
```

- Uncaught exceptions are handled by Python; it kills the program and prints an error message showing where the error occurred
- Exceptions aren't always a bad thing
- Searches sometimes signal success by an exception
- Try statements can be used to debug code

Python Standard Library:

`string` string manipulation

`re` regular expressions

`os` generic operating system interface

`shutil` copying files and directories

`cgi` common gateway interface

`struct` binary data manipulation

Examples

CGI script: Process HTML form data, store it, email it, and display a response.

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

import cgi, os, sys, string
from printform import printform, form_keys
from time import ctime, time

FROMADDR="bemays@virginia.edu"
TOADDRES=["bemays@virginia.edu",
          "jmalone@tovaris.com",
          "cr8s@virginia.edu"]

def print_init():
    print """Content-type: text/html
    ...
    """
```

```
def print_end():
    print """
    <p>Please save or print this page for your records.</p>
    ...
</html>
    """

def mail_init(out):
    out.write("""From: %s
To: %s
Subject: New installfest registration
MIME-Version: 1.0
Content-Type: text/html; charset="us-ascii"

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    ...
    <p><b>New Participant:</b></p>
    """ % (FROMADDR, string.join(TOADDRES, ", ")))
```

```
def mail_end(out):
    out.write("</body>\n</html>\n")

if __name__ == "__main__":
    os.chdir("/home/bem/public_html")
    sys.stderr = sys.stdout
    form = cgi.FieldStorage()
    xmlf = open("data/if_data.xml", "a")
    xmlf.write("<participant>\n")
    dict = {}
    dict['regtime'] = cgi.escape(ctime(time()))
    xmlf.write("  <regtime>%s</regtime>\n" % dict['regtime'])
    for key in form_keys:
        if (form.has_key(key) and
            type(form[key].value) != type([]) and
            form[key].value != ""):
            val = string.replace(cgi.escape(form[key].value),
                                "\r\n", "&nl;")
            xmlf.write("  <%s>%s</%s>\n" % (key, val, key))
            dict[key] = string.replace(val, "&nl;", "<br>")
```

```
xmlf.write("</participant>\n")
mail = os.popen("/usr/lib/sendmail -bm " +
                string.join(TOADDRESSES), "w")
mail_init(mail)
printform(mail, dict)
mail_end(mail)
mail.close()
print_init()
printform(sys.stdout, dict)
print_end()
```

CGI script: Parse XML file and print data in HTML format.

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

import os, sys
from string import join, replace
from printform import printform
import xmllib
```

```
class Parser(xmllib.XMLParser):
    def __init__(self):
        xmllib.XMLParser.__init__(self)
        self.entitydefs["nl"] = "&lt;br&gt;"
        self.dict = {}
        self._datain = []
    def load(self, file):
        self.feed("<top>")
        while 1:
            s = file.read(512)
            if not s: break
            self.feed(s)
        self.feed("</top>")
        self.close()
    def handle_data(self, data):
        self._datain.append(data)
    def start_top(self, attrs): pass
    def end_top(self): pass
```

```
def start_participant(self, attrs):
    self.dict = {}
def end_participant(self):
    printform(sys.stdout, self.dict)
def unknown_starttag(self, tag, attrs):
    self._datain = []
def unknown_endtag(self, tag):
    self.dict[tag] = join(self._datain, "")

def print_init():
    ...
def print_end():
    ...
if __name__ == "__main__":
    os.chdir("/home/bem/public_html")
    sys.stderr = sys.stdout
    print_init()
    p = Parser()
    p.load(open("data/if_data.xml"))
    print_end()
```

Numerical Analysis: Postprocess a data file and calculate errors.

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

import sys, re, Scientific.IO.NetCDF
from math import cos, sin, sqrt, atan2, pi
import havg          # C module for calculating h
degs = pi / 180.0

def true_coords(alpha, dellam):
    x = cos(alpha) * sin(dellam)
    y = -cos(dellam)
    z = sin(alpha) * sin(dellam)
    l = sqrt(x*x + y*y)
    return (atan2(y, x), atan2(z, l))

if __name__ == "__main__":
    rec = int(sys.argv[1])
    files = sys.argv[2:]
    for file in files:
        nc = Scientific.IO.NetCDF.NetCDFFile(file)
```


Python module in C: Calculate average h for each cell.

To build:

1. Create a file named `Setup.in` containing the following:

```
*shared*  
havg havgmodule.c
```

2. Copy `Makefile.pre.in` into current directory
3. Run `“make -f Makefile.pre.in boot”`
4. Fix generated makefile
5. Run `“make”`
6. Copy `havgmodule.so` to destination

```
#include <Python.h>

typedef struct {
    double lam, phi;
    double rad;
} Bell;

double havg(Bell *bell, float lam, float phi,
            float dellam, float delphi);

static PyObject *havg_h(PyObject * self, PyObject * args);

static PyMethodDef HavgMethods[] = {
    {"h", havg_h, METH_VARARGS},
    {NULL, NULL}
};

void inithavg(void)
{
    (void) Py_InitModule("havg", HavgMethods);
}
```

```
static PyObject *havg_h(PyObject * self, PyObject * args)
{
    float h, lam, phi, dellam, delphi, lamc, phic, r;
    Bell b;

    if (!PyArg_ParseTuple(args, "fffffff",
        &lam, &phi, &dellam, &delphi, &lamc, &phic, &r))
        return NULL;
    b.lam = lamc;
    b.phi = phic;
    b.rad = r;
    h = havg(&b, lam, phi, dellam, delphi);
    return Py_BuildValue("d", h);
}
```

Summary

- Python is easy to use
 - Dynamic typing
 - Automatic memory management
 - “Executable pseudocode”
- Python is powerful
 - Built-in object types and tools
 - Extensible with new types (OOP)
 - Expandable with C (or other languages)
- Python is useful
 - CGI programming
 - Data handling (XML)
 - Scientific computation