An introduction to Python

Source:

http://www.python.org

PDF:

http://www.monard.info

Introduction: why Python

Python in a few words

Python is a **scripting** language which can replace most task scripts and compiled programs written in languages like C/C++/Java (no compilation is necessary).

Python is available in many platforms: Windows, Mac, Linux, etc.

It is a very high level language that provides

- high level data types built-in (lists, dictionnaries, tuples, sets, ...),
- object-oriented programming (but this is not mandatory 🙂),
- numerous modules (I/O, system calls, threads, GUIs, ...),
- and dynamical prototyping (no need to declare variable type).

Python is extensible: one can write C, Fortran or Java libraries which can be called directly in a Python script.

The language is named after the BBC show “Monty Python’s Flying Circus” and has nothing to do with reptiles.

History

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI) in the Netherlands as a successor of a language called ABC.

Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, Benevolent Dictator for Life (BDFL).

In 2001, the Python Software Foundation (PSF) was formed, a non-profit organization created specifically to own Python-related Intellectual Property.

From 2005 to 2012, Van Rossum worked for Google.

In January 2013, Van Rossum started working with Dropbox

All Python releases are Open Source.

- Python 1.0 - January 1994
    - Python 1.5 - December 31, 1997
    - Python 1.6 - September 5, 2000
- Python 2.0 - October 16, 2000
    - Python 2.1 - April 17, 2001
    - Python 2.2 - December 21, 2001
    - Python 2.3 - July 29, 2003
    - Python 2.4 - November 30, 2004
    - Python 2.5 - September 19, 2006
    - Python 2.6 - October 1, 2008
    - Python 2.7 - July 3, 2010

- Python 3.0 - December 3, 2008
    - Python 3.1 - June 27, 2009
    - Python 3.2 - February 20, 2011
    - Python 3.3 - September 29, 2012
    - Python 3.4 - March 16, 2014

Python 2.x vs. Python 3.x

Python 3.0 (also called Python 3000 or py3k), a major, backwards-incompatible release, was released on 3 December 2008

Python 2.x and Python 3.x branches have been planned to coexist in parallel release.

Python 2.6 was released to coincide with Python 3.0, and included some features from that release.

Similarly, Python 2.7 coincided with and included features from Python 3.1.

Parallel releases ceased as of Python 3.2.

Stables releases: 3.4.1 (18 May 2014), 2.7.8 (31 May 2014)

The tutorial presented here will only concern the 2.x branch.

First steps in python

The Python interpreter

To start using Python, just ask `python` !

```
Python 2.7.3 (default, Aug  1 2012, 05:14:39)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can impose the python version you want to use:

```
 $ python2.7
Python 2.7.3 (default, Aug  1 2012, 05:14:39)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To quit the python interpreter, type `Ctrl-D`.

Playing with the python interpreter:

```
>>> 2+3
5
>>> 3/5.3
0.5660377358490566
>>> (2+3)*5+4**3
89
>>> print 'hello'
hello
>>> print 'another hello'
another hello
>>> print 'you can use \' or " in a string definition'
you can use ' or " in a string definition
>>> print "don't you understand?"
don't you understand?
```

Python scripts

On Unix/Linux systems, Python scripts can be made directly executable, like shell scripts, by putting the line

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

The script can be given an executable mode, or permission, using the chmod command:

```
chmod +x myscript.py
```

On Windows systems, there is no notion of an “executable mode”. The Python installer automatically associates .py files with python.exe so that a double-click on a Python file will run it as a script. The extension can also be .pyw, in that case, the console window that normally appears is suppressed.

Source code encoding

It is possible to use encodings different than ASCII in Python source files. The best way to do it is to put one more special comment line right after the #! line to define the source file encoding:

```
#!/usr/bin/env python
# -*- coding: iso8859-1 -*-

print "un pétit encodéûr"
```

```
$ ./script1.py
un pétit encodéûr
```

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

print "un pétit encodéûr"
```

```
$ ./script1.py
File "script.py", line 3
SyntaxError: Non-ASCII character '3' in file script.py on line 3, but n
```

Python as a calculator

When used directly as an interpreter, python is first a calculator.

Prompts:

- >>>: enter a command
- (no prompt): the result of the command
- `...`: wait for the end of the command (continuation of the previous line or block of commands)

Comments:

- all comments in python start with a # and finish at the end of the current line

```
# this is the first comment
SPAM = 1
# and this is the second comment
# ... and now a third!
STRING = "# This is not a comment."
```

Numbers

```
>>> 2+2
4
>>> # This is a comment
... 4+4-2*3
2
>>> (50+6)/3
18
>>> # integer division returns the floor number
... # if you want a real, you must perform the division
... # using real number
... (50.+6)/3
18.666666666666668
>>>
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

A value can be assigned to several variables simultaneously:

```
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

Variables must be “defined” (assigned a value) before they can be used, or an error will occur:

```
>>> n
Traceback (most recent call last):
  File "`<stdin>`", line 1, in `<module>`
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Complex numbers are also supported; imaginary numbers are written with a suffix of j or J. Complex numbers with a nonzero real component are written as (real+imagj), or can be created with the complex(real, imag) function.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Complex numbers are always represented as two floating point numbers, the real and imaginary part. To extract these parts from a complex number z, use z.real and z.imag.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>> price + _
113.0625
>>> round(_, 2)
113.06
```

Strings

```
a = 'This is a string'
b = "This is another string"
c = """And another way of defining a string"""
d = "The difference between \', \", and \"\"\" is \n the way \
a newline is handled"
e = """ using three double quotes " is better when
multiple lines are involved"""

print a
print b
print c
print d
print e
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
text = "This is a "+"dead parrot !"
print text
print (text+" ")*3
```

Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) 0. There is no separate character type; a character is simply a string of size one. Substrings can be specified with the slice notation: two indices separated by a colon.

```
text = "This is a "+"dead parrot !"
print text           # all characters
print text[0]        # character at index 0
print text[10]       # character at index 10
print text[10:14]    # characters from index 10 to index 14 (excluded)
print text[15:]      # characters from index 15 to the end
                     #   of the string
print text[:4]       # first four characters
print text[10:-9]    # from index 10 to the end of the string
                     #   minus 9 characters
```

The built-in function len() returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Unlike strings, which are immutable, it is possible to change individual elements of a list:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
```

```
>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Clear the list: replace all items with an empty list
>>> a[:] = []
>>> a
[]
```

The built-in function `len()` also applies to lists:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Note that in the last example, p[1] and q really refer to the same object!

Programming in Python

A first example

Python can be used for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...   print b
...   a, b = b, a+b
...
1
1
2
3
5
8
```

This example introduces several new features.

multiple variable assignment

The first line contains a multiple assignment: the variables a and b simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

conditional loop

The while loop executes as long as the condition (here: b < 10) remains true. In Python, like in C/Java, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison.

The standard comparison operators are written the same as in C:

- `<` (less than),
- `>` (greater than),
- `==` (equal to),
- `<=` (less than or equal to),
- `>=` (greater than or equal to)
- `!=` (not equal to).

blocks are made by indenting the code

The body of the loop is indented: indentation is Python’s way of grouping statements. Each line within a basic block must be indented by the same amount.

print statement

The print statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple expressions and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

A trailing comma avoids the newline after the output:

```
>>> 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
```

if statements

```
x = int(raw_input("Please enter an integer: "))
if x < 0:
x = 0
     print 'Negative changed to zero'
elif x == 0:
     print 'Zero'
elif x == 1:
     print 'Single'
else:
     print 'More'
```

There can be zero or more elif parts, and the else part is optional. The keyword `elif` is short for `else if`, and is useful to avoid excessive indentation. An `if ... elif ... elif ... else ...` sequence is a substitute for the switch or case statements found in other languages.

for statement

Python’s `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

the range() function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates lists containing arithmetic progressions:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; `range(10)` generates a list of 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the *step*):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

break, continue, and else (again)

The `break` statement breaks out of the smallest enclosing `for` or `while` loop. The `continue` statement continues with the next iteration of the loop.

Loop statements may have an `else` clause; it is executed when the loop terminates through exhaustion of the list (with `for`) or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement. This is exemplified by the following loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
```

```
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

pass

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
...
```

Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

defining functions

a first example

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring.

Instead of printing the results, one can write a function that returns a list of the numbers of the Fibonacci series.

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

The `return` statement returns with a value from a function. `return` without an expression argument returns None. Falling off the end of a function also returns `None`.

A function can return more than one argument using a tuple:

```python
def euclidean(n,m):
  """return the integer division and the modulo of n/m"""
  return n/m, n%m

a, b = euclidean(18,7)
print "Euclidean division"
print "%d = %d * %d + %d" % (18, a, 7, b)
```

default argument values

Python allows the use of default argument values. This creates a function that can be called with fewer arguments than it is defined to allow.

```
def f(x = 2, y = 3):
  return x+y
print f(1, 2), f(1), f(x=2), f(y=3), f(y=5,x=-2)
```

Documentation strings

Python allows the code documentation (use it ! 🙂)

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

More on lists

The list data type has some more methods. Here are all of the methods of list objects:

list.append(x):

Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

list.extend(L):

Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

list.insert(i, x):

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

list.remove(x):

Remove the first item from the list whose value is `x`. It is an error if there is no such item.

list.pop([i]):

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

list.index(x):

Return the index in the list of the first item whose value is `x`. It is an error if there is no such item.

list.count(x):

Return the number of times `x` appears in the list.

list.sort():

Sort the items of the list, in place.

list.reverse():

Reverse the elements of the list, in place.

Functional programming tools

There are three built-in functions that are very useful when used with lists: `filter()`, `map()`, and `reduce()`.

filter(function, sequence)

returns a sequence consisting of those items from the sequence for which `function(item)` is true. For example, to compute some primes:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

map(function, sequence)

calls `function(item)` for each of the sequence's items and returns a list of the return values. For example, to compute some cubes:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

reduce(function, sequence)

returns a single value constructed by calling the binary function `function` on the first two items of `sequence`, then on the result and the next item, and so on. For example, to compute the sum of the numbers 1 through 10:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

List Comprehension

List comprehensions provide a concise way to create lists without resorting to use of `map()`, or `filter()`. The resulting list definition tends often to be clearer than lists built using those constructs.

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Tuples

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

It is not possible to assign to the individual items of a tuple. However, it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain an heterogeneous sequence of elements that are accessed via unpacking or indexing (or even by attribute in the case of namedtuples). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

The statement:

```
t = 12345, 54321, ’hello!’
```

is an example of tuple packing: the values 12345, 54321 and ’hello!’ are packed together in a tuple. The reverse operation is also possible:

```
>>> x, y, z = t
```

Dictionnaries

It is best to think of a dictionary as an unordered set of `key: value` pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of `key:value` pairs within the braces adds initial `key:value` pairs to the dictionary.

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
```

```
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

Looping techniques:

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for (key, value) in knights.iteritems():
...     print key, value
...
gallahad the pure
robin the brave
```

Modules

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended.

```
# Fibonacci numbers module
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b
def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Using the module (saved as `fibo.py`): .. code-block:: python

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Python comes with a library of standard modules, described in a separate document, the Python Library Reference.

Input/Output

old string formatting

The `%` operator can also be used for string formatting.

```
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

reading and writing files

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

To read a file’s contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. `size` is an optional numeric argument. When size is omitted or negative, the entire contents of the file will be read and returned.

```
>>> f = open('/tmp/workfile', 'w')
>>> print f
`<open file '/tmp/workfile', mode 'w' at 80a0960>`
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
>>> f.close()
```

`f.readline()` reads a single line from the file.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

An alternative approach to reading lines is to loop over the file object. This is memory efficient, fast, and leads to simpler code:

```
>>> for line in f:
        print line,

This is the first line of the file.
Second line of the file
```

`f.write(string)` writes the contents of string to the file, returning `None`.

```
>>> f.write('This is a test\n')
```

To write something other than a string, it needs to be converted to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

Standard input is available with `sys.stdin` file object. A simple equivalent to the `cat` UNIX command is:

```
import sys

for line in sys.stdin:
  print line,             # don't forget the comma!
```

Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "`<stdin>`", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "`<stdin>`", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "`<stdin>`", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

The last line of the error message indicates what happened.

Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`.

The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions.

Handling exceptions

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

The try statement works as follows.

- First, the `try` clause (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the `except` clause is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the `except` clause is executed, and then execution continues after the `try` statement.
- If an exception occurs which does not match the exception named in the `except` clause, it is passed on to outer `try` statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

A `try` statement may have more than one `except` clause, to specify handlers for different exceptions.

At most one handler will be executed.

Handlers only handle exceptions that occur in the corresponding `try` clause, not in other handlers of the same `try` statement.

An `except` clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError("This is my error message")
Traceback (most recent call last):
  File "`<stdin>`", line 1, in `<module>`
NameError: This is my error message
```

Programs may name their own exceptions by creating a new exception class. Exceptions should typically be derived from the Exception class, either directly or indirectly.

Classes

Class Definition

The simplest form of class definition looks like this:

```python
class MyFirstClass(object):
   # list of statements
   pass
```

The `object` statement is not mandatory but highly recommended since it allows the use of "modern" python classes.

Class Instantiation

The creation (instantiation) of python objects is performed by calling the class:

```
x = MyFirstClass()
```

Object Attributes

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`.

Valid attribute names are all the names that were in the class’s namespace when the class object was created.

Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to.

```
class MyFirstClass(object):
  pass

x = MyFirstClass()
x.a = 1
x.b = 'toto'
print x.a
print x.b
```

```python
class ClassWithAttributes(object):
  i = 12345

x = ClassWithAttributes()
print x.i
```

Object Methods

A method is a function that “belongs to” an object.

Usually, a method is called right after it is bound:

```
x.f()
```

When an object method is called by python, a reference to the object is given as the **first** parameter of the function. Therefore, a method defined in a class body has always a mandatory first parameter, conventionally named `self`.

```
class MyClass(object):
  def addition(self, a, b):  # <-- three parameters !
    """return the addition of two parameters"""
    return a+b

a = MyClass()
print a.addition(2,3) # <-- Two parameters !
         #(the parameter 'self' is automatically include by python)
```

Constructors

Class constructor is a special function as defined in the class: `__init__()`.

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance.

```
class MyClass(object):
  def __init__(self):
    print "MyClass instance is generated"

x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility.

In that case, arguments given to the class instantiation operator are passed on to `__init__()`.

```
class MyClass(object):
  def __init__(self, value):
    print "MyClass instance is generated with value:"+repr(value)

x = MyClass('toto')
```

Object Attributes (bis)

Using the `self` reference, it is possible to distinguish in a method definition between local variables and object attributes:

```
>>> class MyClass(object):
...   a = 1          # object attribute
...   def f1(self):
...     a = 2        # local variable
...   def f2(self):
...     self.a = -3  # self attribute
...
>>> x = MyClass()
>>> x.a
1
>>> x.f1()
>>> x.a
1
```

```
>>> x.f2()
>>> x.a
-3
```

Static Methods

A static method is a method that you can call on a class, or on any instance of the class, without the special behavior and constraints of ordinary methods (which are bound on the first argument).

In python, a static method can be built using the `staticmethod` built-in type:

```python
class Complex(object):
  def __init__(self, r, i):
    self.real = r
    self.img = i

  def conjugate(self):
    """return the conjugate of the complex number"""
    return Complex(self.real, -self.img)
  def addition(a, b):    # <-- no 'self' parameter
    """return the addition of two complex numbers"""
    return Complex(a.real+b.real, a.img+b.img)
  addition = staticmethod(addition)

  def __repr__(self):
    """return a representation of the complex number"""
    return "(%f, %f)" % (self.real, self.img)

z1 = Complex(1.4, 2.3)
z2 = z1.conjugate()
zsomme = Complex.addition(z1, z2)
print repr(z1)
print repr(z2)
print repr(zsomme)
```

Properties

A property is an instance attribute with special functionality. You reference the attribute with the normal syntax (`obj.x`). However, rather than following the usual semantics for attribute reference, specific access call methods are used:

```
class C(object):
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

An example:

```
class PositiveInteger(object):
  def __init__(self):
    self.__n = 0  # the two underscores mean 'private' attribute
                  # while one underscore means 'protected' attribute
                  # (self._x is accessible within the class
                  #  and the subclasses,
                  #  self.__n is accessible only within the class)

  def setN(self, n):
    if (n < 0):
      raise TypeError, "%d is not a positive integer" % n
    self.__n = n

  def getN(self):
    return self.__n
  n = property(getN, setN)
```

```python
a = PositiveInteger()
a.n = 2
print a.n
a.n = -2
```

Set, delete and doc attributes are not mandatory. This enables the use of properties when an attribute is only allowed to be read and not set.

```python
class Rectangle(object):
  def __init__(self, width, height):
    self.width = width
    self.height = height

  def getArea(self):
    return self.width*self.height
  area = property(getArea)

r = Rectangle(10, 22)
print r.area
```

Inheritance

The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    `<statement-1>`
    .
    .
    .
    `<statement-N>`
```

Python has two built-in functions that work with inheritance:

```
* Use isinstance() to check an object’s type: ``isinstance(obj, int)`` w
* Use issubclass() to check class inheritance: ``issubclass(subclass, c

>>> a = 3
>>> isinstance(a, int)       # a is an instance of the 'int' class
True
>>> isinstance(a, float)     # but not of the 'float' class
False
>>> issubclass(bool, int)    # the 'bool' class is derived from 'int'
True
>>> issubclass(bool, float)  # but not from 'float'
False
```

```
class Rectangle(object):
  def __init__(self, width, height):
    print "Rectangle __init__"
    self.width = width
    self.height = height
  def getArea(self):
    print "Rectangle getArea"
    return self.width*self.height
  area = property(getArea)
class Trapeze(Rectangle):
  def __init__(self, base, height):
    self.width = base
    self.height = height
class Square(Rectangle):
  def __init__(self, size):
    print "Square __init__"
    Rectangle.__init__(self, size, size) # call the super constructor
```

```python
print "issubclass(Square, Rectangle) ", issubclass(Square, Rectangle)
print "issubclass(Rectangle, Square) ", issubclass(Rectangle, Square)
print "issubclass(Rectangle, object) ", issubclass(Rectangle, object)
print "issubclass(Square, object) ",    issubclass(Square, object)
print "issubclass(Square, Trapeze) ",   issubclass(Square, Trapeze)

t = Trapeze(10, 20)
print "isinstance(t, Trapeze) ", isinstance(t, Trapeze)
print "isinstance(t, Rectangle) ", isinstance(t, Rectangle)
print "isinstance(t, Square) ", isinstance(t, Square)
print t.area

a = Square(20)
print "isinstance(a, Trapeze) ", isinstance(a, Trapeze)
print "isinstance(a, Rectangle) ", isinstance(a, Rectangle)
print "isinstance(a, Square) ", isinstance(a, Square)
print a.area
```

General-Purpose Special Methods

Some special methods relate to general-purpose operations. A class that defines or inherits these methods allows its instances to control such operations.

```python
class Complexe(object):
  def __init__(self, r, i):
    self.real = r
    self.img = i

  def __add__(self, other):
    return Complexe(self.real + other.real, self.img + other.img)

  def __sub__(self, other):
    return Complexe(self.real - other.real, self.img - other.img)
  def __mul__(self, other):
    return Complexe(self.real*other.real-self.img*other.img,
                    self.real*other.img + self.img*other.real)

  def __div__(self, other):
    denominator = other.real**2 + other.img**2
    return Complexe( (self.real*other.real +
                      self.img*other.img)/denominator,
                     (self.img*other.real -
                      self.real*other.img)/denominator)

  def __repr__(self):
    return "%f + %fi" % (self.real, self.img)

z1 = Complexe(1, 2.4)
z2 = Complexe(-3.2, 5.1)
print z1+z2, z1-z2, z1*z2, z1/z2
```

Brief Tour of the Standard Library

See the Python documentation.

Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system

```
>>> import os
>>> os.getcwd()
'/home/gmonard/Ecrit/Conferences/2013/Ecole-CORREL-Paris/python-wiki/ex
>>> os.chdir('/tmp/')
>>> os.mkdir('test')
>>> os.chdir('test')
>>> os.getcwd()
'/tmp/test'
>>>
```

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`.

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('my_source_file', 'my_target_file')
>>> shutil.move('/tmp/filename', 'newfilename')
```

Useful functions on pathnames are implemented in the `os.path` module.

```
>>> os.path.basename('/tmp/toto1.txt')
'toto1.txt'
>>> os.path.dirname('/tmp/toto1.txt')
'/tmp'
>>> os.path.isfile('/tmp/toto1.txt')
False
>>> os.path.isfile('script.py')
True
```

File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list.

```
>>> import sys
>>> print sys.argv
```

The `getopt` module processes `sys.argv` using the conventions of the Unix `getopt()` function. More powerful and flexible command line processing is provided by the `argparse` module.

Error Output Redirection and Program Termination

The `sys` module also has attributes for *stdin*, *stdout*, and *stderr*. The latter is useful for emitting warnings and error messages to make them visible even when stdout has been redirected:

```
>>> sys.stderr.write('Warning, log file not found\n')
Warning, log file not found
```

The most direct way to terminate a script is to use `sys.exit()`.

Mathematics

The `math` module gives access to the underlying C library functions for floating point math:

```
>>> import math
>>> math.cos(math.pi/4)
0.7071067811865476
>>> math.log(1024, 2)
10.0
>>>
```

The `random` module provides tools for making random selections

```
>>> import random
>>> random.choice(['apple', 'orange', 'banana'])
'apple'
>>> random.choice(['apple', 'orange', 'banana'])
'orange'
>>> random.choice(['apple', 'orange', 'banana'])
'banana'
>>> random.choice(['apple', 'orange', 'banana'])
'banana'
>>> random.choice(['apple', 'orange', 'banana'])
'apple'
```

```
>>> random.sample(range(100), 10)    # sampling without replacement
[6, 16, 39, 0, 4, 11, 55, 54, 13, 12]
>>> random.random()                  # random float
0.28630038627790344
>>> random.random()
0.130898082399855
```

```
>>> random.randrange(6)   # random integer chosen from range(6)
5
>>> random.randrange(6)
4
>>> random.randrange(6)
5
>>> random.randrange(6)
4
>>> random.randrange(6)
1
>>> random.randrange(6)
3
>>>
```

Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are `urllib2` for retrieving data from URLs and `smtplib` for sending mail.

Data Compression

Common data archiving and compression formats are directly supported by modules including: `zlib`, `gzip`, `bz2`, `zipfile` and `tarfile`.

Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The `doctest` module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the `doctest` module to make sure the code remains true to the documentation:

```python
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The `unittest` module is not as effortless as the `doctest` module, but it allows a more comprehensive set of tests to be maintained in a separate file.