Python Tutorial
(DRAFT)

Guido van Rossum
Dept. CST, CWI, Kruislaan 413
1098 SJ Amsterdam, The Netherlands
E-mail: `gu...@cwi.nl`

August 31, 2023

Abstract

Python is a simple, yet powerful programming language that bridges the gap between C and shell programming, and is thus ideally suited for rapid prototyping. Its syntax is put together from constructs borrowed from a variety of other languages; most prominent are influences from ABC, C, Modula-3 and Icon.

The Python interpreter is easily extended with new functions and data types implemented in C. Python is also suitable as an extension language for highly customizable C applications such as editors or window managers.

Python is available for various operating systems, amongst which several flavors of UNIX, Amoeba, and the Apple Macintosh O.S.

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It helps to have a Python interpreter handy for hands-on experience, but as the examples are self-contained, the tutorial can be read off-line as well.

For a description of standard objects and modules, see the Library Reference document. The Language Reference document (XXX not yet existing) gives a more formal reference to the language.

Contents

1 Whetting Your Appetite

If you ever wrote a large shell script, you probably know this feeling: you'd love to add yet another feature, but it's already so slow, and so big, and so complicated; or the feature involves a system call or other funcion that is only accessible from C ... Usually the problem at hand isn't serious enough to warrant rewriting the script in C; perhaps because the problem requires variable-length strings or other data types (like sorted lists of file names) that are easy in the shell but lots of work to implement in C; or perhaps just because you're not sufficiently familiar with C.

In all such cases, Python is just the language for you. Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than the shell has. On the other hand, it also offers much more error checking than C, and, being a *very-high-level language*, it has high-level data types built in, such as flexible arrays and dictionaries that would cost you days to implement efficiently in C. Because of its more general data types Python is applicable to a much larger problem domain than *Awk* or even *Perl*, yet most simple things are at least as easy in Python as in those languages.

Python allows you to split up your program in modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis for your programs — or as examples to start learning to program in Python. There are also built-in modules that provide things like file I/O, system calls, and even a generic interface to window systems (STDWIN).

Python is an interpreted language, which saves you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development. It is also a handy desk calculator.

Python allows writing very compact and readable programs. Programs written in Python are typically much shorter than equivalent C programs: No declarations are necessary (all type checking is dynamic); statement grouping is done by indentation instead of begin/end brackets; and the high-level data types allow you to express complex operations in a single statement.

Python is *extensible*: if you know how to program in C it is easy to add a new built-in module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may be only available in binary form (such as a vendor-specific graphics library). Once you are really hooked, you can link the Python interpreter into an application written in C and use it as an extension or command language.

1.1 Where From Here

Now that you are all excited about Python, you'll want to examine it in some more detail. Since the best introduction to a language is using it, you are invited here to do so.

In the next section, the mechanics of using the interpreter are explained. This is rather mundane information, but essential for trying out the examples shown later. The rest of the tutorial introduces various features of the Python language and system though examples, beginning with simple expressions, statements and data types, through functions and modules, and finally touching upon advanced concepts like exceptions and classes.

2 Using the Python Interpreter

The Python interpreter is usually installed as `/usr/local/python` on those machines where it is available; putting `/usr/local` in your UNIX shell's search path makes it possible to start it by typing the command

```
python
```

to the shell. Since the choice of the directory where the interpreter lives is an installation option, other places instead of `/usr/local` are possible; check with your local Python guru or system administrator.[1]

The interpreter operates somewhat like the UNIX shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a *script* from that file.[2] If available, the script name and additional arguments thereafter are passed to the script in the variable `sys.argv`, which is a list of strings.

When standard input is a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually three greater-than signs (`>>>`); for continuation lines it prompts with the *secondary prompt*, by default three dots (`...`). Typing an EOF (Control-D) at the primary prompt causes the interpreter to exit with a zero exit status.

When an error occurs in interactive mode, the interpreter prints a message and a stack trace and returns to the primary prompt; with input from a file, it exits with a nonzero exit status. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from the executed commands is written to standard output.

Typing an interrupt (normally Control-C or DEL) to the primary or secondary prompt cancels the input and returns to the primary prompt. Typing an interrupt while a command is being executed raises the `KeyboardInterrupt` exception, which may be handled by a `try` statement.

When a module named `foo` is imported, the interpreter searches for a file named `foo.py` in a list of directories specified by the environment variable `PYTHONPATH`. It has the same syntax as the UNIX shell variable `PATH`, i.e., a list of colon-separated directory names. When `PYTHONPATH` is not set, an installation-dependent default path is used, usually `.:/usr/local/lib/python`.[3]

On BSD'ish UNIX systems, Python scripts can be made directly executable, like shell scripts, by putting the line

```
#! /usr/local/python
```

(assuming that's the name of the interpreter) at the beginning of the script and giving the file an executable mode. (The `#!` must be the first two characters of the file.)

[1]At CWI, at the time of writing, the interpreter can be found in the following places: On the Amoeba Ultrix machines, use the standard path, `/usr/local/python`. On the Sun file servers, use `/ufs/guido/bin/`*arch*`/python`, where *arch* can be `sgi` or `sun4`. On piring, use `/userfs3/amoeba/bin/python`. (If you can't find a binary advertised here, get in touch with me.)

[2]There is a difference between "`python file`" and "`python <file`". In the latter case `input()` and `raw_input()` are satisfied from *file*, which has already been read until the end by the parser, so they will read EOF immediately. In the former case (which is usually what you want) they are satisfied from whatever file or device is connected to standard input of the Python interpreter.

[3]Modules are really searched in the list of directories given by the variable `sys.path` which is initialized from `PYTHONPATH` or from the installation-dependent default. See the section on Standard Modules later.

2.1 Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the *GNU Readline* library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won't duplicate here; however, the basics are easily explained.

If supported,[4] input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: C-A (Control-A) moves the cursor to the beginning of the line, C-E to the end, C-B moves it one position to the left, C-F to the right. Backspace erases the character to the left of the cursor, C-D the character to its right. C-K kills (erases) the rest of the line to the right of the cursor, C-Y yanks back the last killed string. C-underscore undoes the last change you made; it can be repeated for cumulative effect.

History substitution works as follows. All non-empty input lines issued are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. C-P moves one line up (back) in the history buffer, C-N moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the Return key passes the current line to the interpreter. C-R starts an incremental reverse search; C-S starts a forward search.

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called `$HOME/.initrc`. Key bindings have the form

```
key-name: function-name
```

and options can be set with

```
set option-name value
```

Example:

```
# I prefer vi-style editing:
set editing-mode vi
# Edit using a single line:
set horizontal-scroll-mode On
# Rebind some keys:
Meta-h: backward-kill-word
Control-u: universal-argument
```

Note that the default binding for TAB in Python is to insert a TAB instead of Readline's default filename completion function. If you insist, you can override this by putting

```
TAB: complete
```

in your `$HOME/.inputrc`. (Of course, this makes it hard to type indented continuation lines.)

This facility is an enormous step forward compared to previous versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on

[4]Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing. If not, you can skip the rest of this section.

continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter's symbol table. A function to check (or even suggest) matching parentheses, quotes etc. would also be useful.

3 An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (`>>>` and `...`): to repeat the example, you must type everything after the prompt, when the prompt appears; everything on lines that do not begin with a prompt is output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

3.1 Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work just as in most other languages (e.g., Pascal or C); parentheses can be used for grouping. For example:

```
>>> # This is a comment
>>> 2+2
4
>>>
>>> (50-5+5*6+25)/4
25
>>> # Division truncates towards zero:
>>> 7/3
2
>>>
```

As in C, the equal sign (`=`) is used to assign a value to a variable. The value of an assignment is not written:

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

There is some support for floating point, but you can't mix floating point and integral numbers in expression (yet):

```
>>> 10.0 / 3.3
3.0303030303
>>>
```

Besides numbers, Python can also manipulate strings, enclosed in single quotes:

```
>>> 'foo bar'
'foo bar'
>>> 'doesn\'t'
'doesn\'t'
>>>
```

Strings are written inside quotes and with quotes and other funny characters escaped by backslashes, to show the precise value. (There is also a way to write strings without quotes and escapes.) Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
>>>
```

Strings can be subscripted; as in C, the first character of a string has subscript 0. There is no separate character type; a character is simply a string of size one. As in Icon, substrings can be specified with the *slice* notation: two subscripts (indices) separated by a colon.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
>>> # Slice indices have useful defaults:
>>> word[:2]    # Take first two characters
'He'
>>> word[2:]    # Drop first two characters
'lpA'
>>> # A useful invariant: s[:i] + s[i:] = s
>>> word[:3] + word[3:]
'HelpA'
>>>
```

Degenerate cases are handled gracefully: an index that is too large is replaced by the string size, an upper bound smaller than the lower bound returns an empty string.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
>>>
```

Slice indices (but not simple subscripts) may be negative numbers, to start counting from the right. For example:

```
>>> word[-2:]    # Take last two characters
'pA'
>>> word[:-2]    # Drop last two characters
'Hel'
>>> # But -0 does not count from the right!
>>> word[-0:]    # (since -0 equals 0)
'HelpA'
>>>
```

The best way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of `n` characters has index `n`, for example:

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0...5 in the string; the second row gives the corresponding negative indices. For nonnegative indices, the length of a slice is the difference of the indices, if both are within bounds, e.g., the length of `word[1:3]` is 3–1 = 2.

Finally, the built-in function `len()` computes the length of a string:

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

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 between square brackets:

```
>>> a = ['foo', 'bar', 100, 1234]
>>> a
['foo', 'bar', 100, 1234]
>>>
```

As for strings, list subscripts start at 0:

```
>>> a[0]
'foo'
>>> a[3]
1234
>>>
```

Lists can be sliced and concatenated like strings:

```
>>> a[1:3]
['bar', 100]
>>> a[:2] + ['bletch', 2*2]
['foo', 'bar', 'bletch', 4]
>>>
```

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

```
>>> a
['foo', 'bar', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['foo', 'bar', 123, 1234]
>>>
```

Assignment to slices is also possible, and this may even change the size of the list:

```
>>> # 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]
>>>
```

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

```
>>> len(a)
4
>>>
```

3.2 Tuples and Sequences

XXX To Be Done.

3.3 First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial subsequence of the *Fibonacci* series as follows:

```
>>> # Fibonacci series:
>>> # the sum of two elements defines the next
>>> a, b = 0, 1
>>> while b < 100:
...         print b
...         a, b = b, a+b
...
1
1
2
3
5
8
13
21
34
55
89
>>>
```

This example introduces several new features.

- 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 `while` loop executes as long as the condition (here: $b < 100$) remains true. In Python, as in C, 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 as `<`, `>`, `=`, `<=`, `>=` and `<>`.[5]
- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. Python does not (yet!) provide an intelligent input line editing facility, so you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; most text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line).
- 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 written 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:

[5]The ambiguity of using `=` for both assignment and equality is resolved by disallowing unparenthesized conditions at the right hand side of assignments.

```
>>> 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
>>>
```

Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.

3.4 More Control Flow Tools

Besides the `while` statement just introduced, Python knows the usual control flow statements known from other languages, with some twists.

3.4.1 If Statements

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> 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...` sequence is a substitute for the *switch* or *case* statements found in other languages.

3.4.2 For Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (as Pascal), or leaving the user completely free in the iteration test and step (as C), Python's `for` statement iterates over the items of any sequence (e.g., a list or a string). For example (no pun intended):

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

3.4.3 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, e.g.:

```
>>> 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, exactly 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):

```
>>> 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, combine `range()` and `len()` as follows:

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

3.4.4 Break Statements and Else Clauses on Loops

The `break` statement breaks out of the smallest enclosing `for` or `while` 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 a list item of value 0:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x = 0:
...            print n, 'equals', x, '*', n/x
...            break
...     else:
...          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
>>>
```

3.4.5 Pass Statements

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

```
>>> while 1:
...       pass # Busy-wait for keyboard interrupt
...
```

3.4.6 Conditions Revisited

XXX To Be Done.

3.5 Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):    # write 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 starts at the next line, indented by a tab stop. The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; variable references first look in the local symbol table, then in the global symbol table, and then in the table of built-in names. Thus, the global symbol table is *read-only* within a function. The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value*.[6] When a function calls another function, a new local symbol table is created for that call.

A function definition introduces the function name in the global symbol table. The value has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

```
>>> fib
<function object at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
>>>
```

You might object that `fib` is not a function but a procedure. In Python, as in C, procedures are just functions that don't return a value. In fact, technically speaking, procedures do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to:

```
>>> print fib(0)
None
>>>
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n): # return 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]
>>>
```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument is used to return from the middle of a procedure (falling off the end also returns from a proceduce).

- The statement `ret.append(b)` calls a *method* of the list object `ret`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this

[6]Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (e.g., items inserted into a list).

may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. See the section on classes, later, to find out how you can define your own object types and methods. The method `append` shown in the example, is defined for list objects; it adds a new element at the end of the list. In this case it is equivalent to `ret = ret + [b]`, but more efficient.[7]

The list object type has two more methods:

`insert(i, x)` Inserts 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)`.

`sort()` Sorts the elements of the list.

For example:

```
>>> a = [10, 100, 1, 1000]
>>> a.insert(2, -1)
>>> a
[10, 100, -1, 1, 1000]
>>> a.sort()
>>> a
[-1, 1, 10, 100, 1000]
>>> # Strings are sorted according to ASCII:
>>> b = ['Mary', 'had', 'a', 'little', 'boy']
>>> b.sort()
>>> b
['Mary', 'a', 'boy', 'had', 'little']
>>>
```

3.6 Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and run it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program. To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. For instance, use your favorite text editor to create a file called `fibo.py` in the current directory with the following contents:

[7]There is a subtle semantic difference if the object is referenced from more than one place.

```
# 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
    ret = []
    a, b = 0, 1
    while b <= n:
          ret.append(b)
          a, b = b, a+b
    return ret
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
>>>
```

This does not enter the names of the functions defined in `fibo` directly in the symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

```
>>> 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]
>>>
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>>
```

3.6.1 More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module is imported somewhere.[8]

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

Modules can import other modules. It is customary but not required to place all `import`

[8]In fact function definitions are also 'statements' that are 'executed'; the execution enters the function name in the module's global symbol table.

statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the `import` statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>>
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>>
```

This imports all names except those beginning with an underscore (_).

3.6.2 Standard Modules

Python comes with a library of standard modules, described in a separate document (Python Library and Module Reference). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option; e.g., the `amoeba` module is only provided on systems that somehow support Amoeba primitives. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determine the interpreter's search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations, e.g.:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
>>>
```

3.7 Errors and Exceptions

Until now error messages haven't yet been mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

3.7.1 Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while 1 print 'Hello world'
Parsing error: file <stdin>, line 1:
while 1 print 'Hello world'
          ^

Unhandled exception: run-time error: syntax error
>>>
```

The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the keyword `print`, since a colon (`:`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

3.7.2 Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it:

```
>>> 10 * (1/0)
Unhandled exception: run-time error: integer division by zero
Stack backtrace (innermost last):
  File "<stdin>", line 1
>>> 4 + foo*3
Unhandled exception: undefined name: foo
Stack backtrace (innermost last):
  File "<stdin>", line 1
>>> '2' + 2
Unhandled exception: type error: illegal argument type for built-in operation
Stack backtrace (innermost last):
  File "<stdin>", line 1
>>>
```

Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here.

The first 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 `run-time error`, `undefined name` and `type error`. The rest of the line is a detail whose interpretation depends on the exception type.

The rest of the error message shows the context where the exception happened. In general it contains a stack backtrace listing source lines; however, it will not display lines read from standard input.

Here is a summary of the most common exceptions:

- *Run-time errors* are generally caused by wrong data used by the program; this can be the programmer's fault or caused by bad input. The detail states the cause of the error in more detail.
- *Undefined name* errors are more serious: these are usually caused by misspelled identifiers.[9] The detail is the offending identifier.
- *Type errors* are also pretty serious: this is another case of using wrong data (or better, using data the wrong way), but here the error can be glanced from the object type(s) alone. The detail shows in what context the error was detected.

3.7.3 Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which prints a table of inverses of some floating point numbers:

```
>>> numbers = [0.3333, 2.5, 0.0, 10.0]
>>> for x in numbers:
...     print x,
...     try:
...         print 1.0 / x
...     except RuntimeError:
...         print '*** has no inverse ***'
...
0.3333 3.00030003
2.5 0.4
0 *** has no inverse ***
10 0.1
>>>
```

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, and its type matches the exception named after the `except` keyword, the rest of the try clause is skipped, 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 list, e.g.:

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

[9]The parser does not check whether names used in a program are at all defined elsewhere in the program, so such checks are postponed until run-time. The same holds for type checking.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution!

When an exception occurs, it may have an associated value, also known as the exceptions's *argument*. The presence and type of the argument depend on the exception type. For exception types which have an argument, the except clause may specify a variable after the exception name (or list) to receive the argument's value, as follows:

```
>>> try:
...     foo()
... except NameError, x:
...     print 'name', x, 'undefined'
...
name foo undefined
>>>
```

If an exception has an argument, it is printed as the third part ('detail') of the message for unhandled exceptions.

Standard exception names are built-in identifiers (not reserved keywords). These are in fact string objects whose *object identity* (not their value!) identifies the exceptions.[10] The string is printed as the second part of the message for unhandled exceptions. Their names and values are:

`EOFError`	`'end-of-file read'`	
`KeyboardInterrupt`	`'keyboard interrupt'`	
`MemoryError`	`'out of memory'`	*
`NameError`	`'undefined name'`	*
`RuntimeError`	`'run-time error'`	*
`SystemError`	`'system error'`	*
`TypeError`	`'type error'`	*

The meanings should be clear enough. Those exceptions with a `*` in the third column have an argument.

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except RuntimeError, detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: domain error or zero division
>>>
```

[10]There should really be a separate exception type; it is pure laziness that exceptions are identified by strings, and this may be fixed in the future.

3.7.4 Raising Exceptions

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

```
>>> raise NameError, 'Hi There!'
Unhandled exception: undefined name: Hi There!
Stack backtrace (innermost last):
  File "<stdin>", line 1
>>>
```

The first argument to `raise` names the exception to be raised. The optional second argument specifies the exception's argument.

3.7.5 User-defined Exceptions

Programs may name their own exceptions by assigning a string to a variable. For example:

```
>>> my_exc = 'nobody likes me!'
>>> try:
...     raise my_exc, 2*2
... except my_exc, val:
...     print 'My exception occured, value:', val
...
My exception occured, value: 4
>>> raise my_exc, 1
Unhandled exception: nobody likes me!: 1
Stack backtrace (innermost last):
  File "<stdin>", line 7
>>>
```

Many standard modules use this to report errors that may occur in functions they define.

3.7.6 Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
Unhandled exception: keyboard interrupt
Stack backtrace (innermost last):
  File "<stdin>", line 2
>>>
```

The *finally clause* must follow the except clauses(s), if any. It is executed whether or not an exception occurred. If the exception is handled, the finally clause is executed after the handler (and even if another exception occurred in the handler). It is also executed when the `try` statement is left via a `break` or `return` statement.

3.8 Classes

Classes in Python make it possible to play the game of encapsulation in a somewhat different way than it is played with modules. Classes are an advanced topic and are probably best skipped on the first encounter with Python.

3.8.1 Prologue

Python's class mechanism is not particularly elegant, but quite powerful. It is a mixture of the class mechanisms found in C++ and Modula-3. As is true for modules, classes in Python do not put an absolute barrier between definition and user, but rather rely on the politeness of the user not to "break into the definition." The most important features of classes are retained with full power, however: the class inheritance mechanism allows multiple base classes, a derived class can override any method of its base class(es), a method can call the method of a base class with the same name. Objects can contain an arbitrary amount of private data.

In C++ terminology, all class members (including data members) are *public*, and all member functions (methods) are *virtual*. There are no special constructors or destructors. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects, albeit in the wider sense of the word: in Python, all data types are objects. This provides semantics for renaming or aliasing. But, just like in C++ or Modula-3, the built-in types cannot be used as base classes for extension by the user. Also, like Modula-3 but unlike C++, the built-in operators with special syntax (arithmetic operators, subscripting etc.) cannot be redefined for class members.[11]

3.8.2 A Simple Example

Consider the following example, which defines a class `Set` representing a (finite) mathematical set with operations to add and remove elements, a membership test, and a request for the size of the set.

```
class Set():
    def new(self):
        self.elements = []
        return self
    def add(self, e):
        if e not in self.elements:
            self.elements.append(e)
    def remove(self, e):
        if e in self.elements:
            for i in range(len(self.elements)):
                if self.elements[i] = e:
                    del self.elements[i]
                    break
    def is_element(self, e):
        return e in self.elements
    def size(self):
        return len(self.elements)
```

[11]They can be redefined for new object types implemented in C in extensions to the interpreter, however. It would require only a naming convention and a relatively small change to the interpreter to allow operator overloading for classes, so perhaps someday...

Note that the class definition looks like a big compound statement, with all the function definitons indented repective to the `class` keyword.

Let's assume that this *class definition* is the only contents of the module file `SetClass.py`. We can then use it in a Python program as follows:

```
>>> from SetClass import Set
>>> a = Set().new() # create a Set object
>>> a.add(2)
>>> a.add(3)
>>> a.add(1)
>>> a.add(1)
>>> if a.is_element(3): print '3 is in the set'
...
3 is in the set
>>> if not a.is_element(4): print '4 is not in the set'
...
4 is not in the set
>>> print 'a has', a.size(), 'elements'
a has 3 elements
>>> a.remove(1)
>>> print 'now a has', a.size(), 'elements'
>>>
now a has 2 elements
>>>
```

From the example we learn in the first place that the functions defined in the class (e.g., `add`) can be called using the *member* notation for the object `a`. The member function is called with one less argument than it is defined: the object is implicitly passed as the first argument. Thus, the call `a.add(2)` is equivalent to `Set.add(a, 2)`.

XXX This section is not complete yet!

4 XXX P.M.

- The `del` statement.
- The `dir()` function.
- Tuples.
- Dictionaries.
- Objects and types in general.
- Backquotes.
- And/Or/Not.