Python Tutorial

Guido van Rossum
Dept. AA, CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
E-mail: guido@cwi.nl

10 April 1995
Release 1.2

Abstract

Python is a simple, yet powerful programming language that bridges the gap between C and shell programming, and is thus ideally suited for "throw-away programming" and 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, the Apple Macintosh O.S., and MS-DOS.

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 *Python Library Reference* document. The *Python Reference Manual* gives a more formal definition of the language.

Contents

Chapter 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 function 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 such cases, Python may be 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 many 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 of 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, sockets, and even a generic interface to window systems (STDWIN).

Python is an interpreted language, which can save 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, for several reasons:

- the high-level data types allow you to express complex operations in a single statement;
- statement grouping is done by indentation instead of begin/end brackets;
- no variable or argument declarations are necessary.

Python is *extensible*: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be 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 for that application.

By the way, the language is named after the BBC show "Monty Python's Flying Circus" and has nothing to do with nasty reptiles...

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 way to learn a language is using it, you are invited here to do so.

In the next chapter, 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 user-defined classes.

When you're through with the tutorial (or just getting bored), you should read the Library Reference, which gives complete (though terse) reference material about built-in and standard types, functions and modules that can save you a lot of time when writing Python programs.

Chapter 2

Using the Python Interpreter

2.1 Invoking the Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python` on those machines where it is available; putting `/usr/local/bin` 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 are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

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.

A third way of starting the interpreter is "`python -c command [arg] ...`", which executes the statement(s) in `command`, analogous to the shell's `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is best to quote `command` in its entirety with double quotes.

Note that there is a difference between "`python file`" and "`python <file`". In the latter case, input requests from the program, such as calls to `input()` and `raw_input()`, are satisfied from *file*. Since this file has already been read until the end by the parser before the program starts executing, the program will encounter 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.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing `-i` before the script. (This does not work if the script is read from standard input, for the same reason as explained in the previous paragraph.)

2.1.1 Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are passed to the script in the variable `sys.argv`, which is a list of strings. Its length is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-'` (meaning standard input), `sys.argv[0]` is set to `'-'`. When `-c command` is used, `sys.argv[0]` is set to `'-c'`. Options found after `-c command` are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command to handle.

2.1.2 Interactive Mode

When commands are read from 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.

The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt, e.g.:

```
python
Python 1.2 (Mar 14 1995)
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

2.2 The Interpreter and its Environment

2.2.1 Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (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 the interrupt character (usually Control-C or DEL) to the primary or secondary prompt cancels the input and returns to the primary prompt.[1] Typing an interrupt while a command is executing raises the `KeyboardInterrupt` exception, which may be handled by a `try` statement.

[1] A problem with the GNU Readline package may prevent this.

2.2.2 The Module Search Path

When a module named `spam` is imported, the interpreter searches for a file named `spam.py` in the 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, or when the file is not found there, the search continues in an installation-dependent default path, usually `.:/usr/local/lib/python`.

Actually, modules are searched in the list of directories given by the variable `sys.path` which is initialized from `PYTHONPATH` and the installation-dependent default. This allows Python programs that know what they're doing to modify or replace the module search path. See the section on Standard Modules later.

2.2.3 "Compiled" Python files

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called `spam.pyc` exists in the directory where `spam.py` is found, this is assumed to contain an already-"compiled" version of the module `spam`. The modification time of the version of `spam.py` used to create `spam.pyc` is recorded in `spam.pyc`, and the file is ignored if these don't match.

Whenever `spam.py` is successfully compiled, an attempt is made to write the compiled version to `spam.pyc`. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting `spam.pyc` file will be recognized as invalid and thus ignored later.

2.2.4 Executable Python scripts

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

```
#! /usr/local/bin/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.

2.2.5 The Interactive Startup File

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named `PYTHONSTARTUP` to the name of a file containing your start-up commands. This is similar to the `.profile` feature of the UNIX shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when `/dev/tty` is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same name space where interactive commands are executed, so that objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts `sys.ps1` and `sys.ps2` in this file.

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file, e.g. `execfile('.pythonrc')`. If you want to use the startup file in a script, you must write this explicitly in the script, e.g. `import os; execfile(os.environ['PYTHONSTARTUP'])`.

2.3 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.

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 nothing appears to happen, or if `^P` is echoed, you can skip the rest of this section.

2.3.1 Line Editing

If supported, 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.

2.3.2 History Substitution

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.

2.3.3 Key Bindings

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

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

or

```
"string": function-name
```

and options can be set with

```
set option-name value
```

For 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
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

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...)

2.3.4 Commentary

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

Chapter 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; lines that do not begin with a prompt are output from the interpreter.[1] 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, >>>. (It shouldn't take long.)

3.1.1 Numbers

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 like in most other languages (e.g., Pascal or C); parentheses can be used for grouping. For example:

[1] I'd prefer to use different fonts to distinguish input from output, but the amount of LaTeX hacking that would require is currently beyond my ability.

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
>>>
```

Like 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
>>>
```

A value can be assigned to several variables simultaneously:

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

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

```
>>> 4 * 2.5 / 3.3
3.0303030303
>>> 7.0 / 2
3.5
>>>
```

3.1.2 Strings

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

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>>
```

Strings are written the same way as they are typed for input: inside quotes and with quotes and other funny characters escaped by backslashes, to show the precise value. The string is enclosed in double quotes if the string contains a single quote and no double quotes, else it's enclosed in single quotes. (The `print` statement, described later, can be used to write strings without quotes or 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 (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. Like in Icon, substrings can be specified with the *slice* notation: two indices separated by a colon.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
>>>
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2]    # The first two characters
'He'
>>> word[2:]    # All but the first two characters
'lpA'
>>>
```

Here's a useful invariant of slice operations: `s[:i] + s[i:]` equals `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
>>>
```

Degenerate slice indices 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]
''
>>>
```

Indices may be negative numbers, to start counting from the right. For example:

```
>>> word[-1]     # The last character
'A'
>>> word[-2]     # The last-but-one character
'p'
>>> word[-2:]    # The last two characters
'pA'
>>> word[:-2]    # All but the last two characters
'Hel'
>>>
```

But note that -0 is really the same as 0, so it does not count from the right!

```
>>> word[-0]     # (since -0 equals 0)
'H'
>>>
```

Out-of-range negative slice indices are truncated, but don't try this for single-element (non-slice) indices:

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # error
Traceback (innermost last):
  File "<stdin>", line 1
IndexError: string index out of range
>>>
```

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. The slice from `i` to `j` consists of all characters between the edges labeled `i` and `j`, respectively.

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 2.

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

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

3.1.3 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] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']
>>>
```

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:

```
>>> # 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]
>>> a[:0] = a     # Insert (a copy of) itself at the beginning
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>>
```

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

```
>>> len(a)
8
>>>
```

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! We'll come back to *object semantics* later.

3.2 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 < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
>>>
```

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 < 10`) remains true. In Python, like 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 the same as in C: <, >, ==, <=, >= and `!=`.

- 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 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
  >>>
  ```

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

Chapter 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.

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.

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 (like in 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), in the order that they appear in the sequence. 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
>>>
```

It is not safe to modify the sequence being iterated over in the loop (this can only happen for mutable sequence types, i.e., lists). If you need to modify the list you are iterating over, e.g., duplicate selected items, you must iterate over a copy. The slice notation makes this particularly convenient:

```
>>> for x in a[:]: # make a slice copy of the entire list
...    if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
>>>
```

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

4.4 Break and Continue Statements, and Else Clauses on Loops

The `break` statement, like in C, breaks out of the smallest enclosing `for` or `while` loop.

The `continue` statement, also borrowed from C, 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:
...          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
>>>
```

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
...
```

4.6 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; whereas 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, global variables cannot be directly assigned a value within a function (unless named in a `global` statement), although they may be referenced.

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*.[1] 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 current symbol table. The value of the function name 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
>>>
```

[1] 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).

You might object that `fib` is not a function but a procedure. In Python, like 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 procedure), in which case the `None` value is returned.
- The statement `result.append(b)` calls a *method* of the list object `result`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this 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. (It is possible to define your own object types and methods, using *classes*, as discussed later in this tutorial.) 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 example it is equivalent to `result = result + [b]`, but more efficient.

Chapter 5

Odds and Ends

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

5.1 More on Lists

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

`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)`.

`append(x)` Equivalent to `a.insert(len(a), x)`.

`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.

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

`sort()` Sort the items of the list, in place.

`reverse()` Reverse the elements of the list, in place.

`count(x)` Return the number of times `x` appears in the list.

An example that uses all list methods:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>>
```

5.2 The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This can also be used to remove slices from a list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
>>>
```

`del` can also be used to delete entire variables:

```
>>> del a
>>>
```

Referencing the name `a` hereafter is an error (at least until another value is assigned to it). We'll find other uses for `del` later.

5.3 Tuples and Sequences

We saw that lists and strings have many common properties, e.g., indexing and slicing operations. They are two examples of *sequence* data types. Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

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))
>>>
```

As you see, on output tuples are alway enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression).

Tuples have many uses, e.g., (x, y) coordinate pairs, employee records from a database, etc. Tuples, like strings, are immutable: it is not possible to assign to the individual items of a tuple (you can simulate much of the same effect with slicing and concatenation, though).

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
>>>
```

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, e.g.:

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

This is called, appropriately enough, *tuple unpacking*. Tuple unpacking requires that the list of variables on the left has the same number of elements as the length of the tuple. Note that multiple assignment is really just a combination of tuple packing and tuple unpacking!

Occasionally, the corresponding operation on lists is useful: *list unpacking*. This is supported by enclosing the list of variables in square brackets:

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> [a1, a2, a3, a4] = a
>>>
```

5.4 Dictionaries

Another useful data type built into Python is the *dictionary*. Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which are strings (the use of non-string values as keys is supported, but beyond the scope of this tutorial). 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; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in random order (if you want it sorted, just apply the `sort()` method to the list of keys). To check whether a single key is in the dictionary, use the `has_key()` method of the dictionary.

Here is a small example using a 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']
>>> tel.has_key('guido')
1
>>>
```

5.5 More on Conditions

The conditions used in `while` and `if` statements above can contain other operators besides comparisons.

The comparison operators `in` and `not in` check whether a value occurs (does not occur) in a sequence. The operators `is` and `is not` compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained: e.g., `a < b == c` tests whether `a` is less than `b` and moreover `b` equals `c`.

Comparisons may be combined by the Boolean operators `and` and `or`, and the outcome of a comparison (or of any other Boolean expression) may be negated with `not`. These all have lower priorities than comparison operators again; between them, `not` has the highest priority, and `or` the lowest, so that `A and not B or C` is equivalent to `(A and (not B)) or C`. Of course, parentheses can be used to express the desired composition.

The Boolean operators `and` and `or` are so-called *shortcut* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. E.g., if `A` and `C` are true but `B` is false, `A and B and C` does not evaluate the expression C. In general, the return value of a shortcut operator, when used as a general value and not as a Boolean, is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
>>>
```

Note that in Python, unlike C, assignment cannot occur inside expressions.

5.6 Comparing Sequences and Other Types

Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial subsequence of the other, the shorted sequence is the smaller one. Lexicographical ordering for strings uses the ASCII ordering for individual characters. Some examples of comparisons between sequences with the same types:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)              = (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types is legal. The outcome is deterministic but arbitrary: the types are ordered by their name. Thus, a list is always smaller than a string, a string is always smaller than a tuple, etc. Mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc.[1]

[1] The rules for comparing objects of different types should not be relied upon; they may change in a future version of the language.

Chapter 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 running 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 executed at the top level 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. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibo.py` in the current directory with the following contents:

```
# 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
```

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 current 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]
>>> fibo.__name__
'fibo'
>>>
```

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
>>>
```

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.[1]

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` 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:

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

```
>>> 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 (_).

6.2 Standard Modules

Python comes with a library of standard modules, described in a separate document (Python Library 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')
>>>
```

6.3 The `dir()` function

The built-in function `dir` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__name__', 'argv', 'builtin_module_names', 'copyright', 'exit',
'maxint', 'modules', 'path', 'ps1', 'ps2', 'setprofile', 'settrace',
'stderr', 'stdin', 'stdout', 'version']
>>>
```

Without arguments, `dir()` lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
>>>
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['AccessError', 'AttributeError', 'ConflictError', 'EOFError', 'IOError',
'ImportError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'MemoryError', 'NameError', 'None', 'OverflowError', 'RuntimeError',
'SyntaxError', 'SystemError', 'SystemExit', 'TypeError', 'ValueError',
'ZeroDivisionError', '__name__', 'abs', 'apply', 'chr', 'cmp', 'coerce',
'compile', 'dir', 'divmod', 'eval', 'execfile', 'filter', 'float',
'getattr', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'len', 'long',
'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'range', 'raw_input',
'reduce', 'reload', 'repr', 'round', 'setattr', 'str', 'type', 'xrange']
>>>
```

Chapter 7

Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the `print` statement. (A third way is using the `write` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. The key to nice formatting in Python is to do all the string handling yourself; using string slicing and concatenation operations you can create any lay-out you can imagine. The standard module `string` contains some useful operations for padding strings to a given column width; these will be discussed shortly. Finally, the `%` operator (modulo) with a string left argument interprets this string as a C sprintf format string to be applied to the right argument, and returns the string resulting from this formatting operation.

One question remains, of course: how do you convert values to strings? Luckily, Python has a way to convert any value to a string: just write the value between reverse quotes (``). Some examples:

```
>>> x = 10 * 3.14
>>> y = 200*200
>>> s = 'The value of x is ' + `x` + ', and y is ' + `y` + '...'
>>> print s
The value of x is 31.4, and y is 40000...
>>> # Reverse quotes work on other types besides numbers:
... p = [x, y]
>>> ps = `p`
>>> ps
'[31.4, 40000]'
>>> # Converting a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = `hello`
>>> print hellos
'hello, world\012'
>>> # The argument of reverse quotes may be a tuple:
... `x, y, ('spam', 'eggs')`
"(31.4, 40000, ('spam', 'eggs'))"
>>>
```

Here are two ways to write a table of squares and cubes:

```
>>> import string
>>> for x in range(1, 11):
...     print string.rjust(`x`, 2), string.rjust(`x*x`, 3),
...     # Note trailing comma on previous line
...     print string.rjust(`x*x*x`, 4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>>
```

(Note that one space between each column was added by the way `print` works: it always adds spaces between its arguments.)

This example demonstrates the function `string.rjust()`, which right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar functions `string.ljust()` and `string.center()`. These functions do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `string.ljust(x, n)[0:n]`.)

There is another function, `string.zfill`, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

```
>>> string.zfill('12', 5)
'00012'
>>> string.zfill('-3.14', 7)
'-003.14'
>>> string.zfill('3.14159265359', 5)
'3.14159265359'
>>>
```

Chapter 8

Errors and Exceptions

Until now error messages haven't been more than 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*.

8.1 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'
  File "<stdin>", line 1
    while 1 print 'Hello world'
                ^
SyntaxError: invalid syntax
>>>
```

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.

8.2 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* 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:

```
>>> 10 * (1/0)
Traceback (innermost last):
  File "<stdin>", line 1
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (innermost last):
  File "<stdin>", line 1
NameError: spam
>>> '2' + 2
Traceback (innermost last):
  File "<stdin>", line 1
TypeError: illegal argument type for built-in operation
>>>
```

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 name for the exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line is a detail whose interpretation depends on the exception type; its meaning is dependent on the exception type.

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

The Python library reference manual lists the built-in exceptions and their meanings.

8.3 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, 10]
>>> for x in numbers:
...     print x,
...     try:
...         print 1.0 / x
...     except ZeroDivisionError:
...         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, the rest of the clause is skipped. Then if 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
```

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way!

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

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

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 ZeroDivisionError, detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo
>>>
```

8.4 Raising Exceptions

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

```
>>> raise NameError, 'HiThere'
Traceback (innermost last):
  File "<stdin>", line 1
NameError: HiThere
>>>
```

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

8.5 User-defined Exceptions

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

```
>>> my_exc = 'my_exc'
>>> try:
...     raise my_exc, 2*2
... except my_exc, val:
...     print 'My exception occurred, value:', val
...
My exception occurred, value: 4
>>> raise my_exc, 1
Traceback (innermost last):
  File "<stdin>", line 1
my_exc: 1
>>>
```

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

8.6 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!
Traceback (innermost last):
  File "<stdin>", line 2
KeyboardInterrupt
>>>
```

A `finally` clause is executed whether or not an exception has occurred in the `try` clause. When an exception has occurred, it is re-raised after the `finally` clause is executed. The `finally` clause is also executed "on the way out" when the `try` statement is left via a `break` or `return` statement.

A `try` statement must either have one or more `except` clauses or one `finally` clause, but not both.

Chapter 9

Classes

Python's class mechanism adds classes to the language with a minimum of new syntax and semantics. 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 methods 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 the data members) are *public*, and all member functions 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 importing and renaming. But, just like in C++ or Modula-3, built-in types cannot be used as base classes for extension by the user. Also, like in C++ but unlike in Modula-3, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class members.

9.1 A word about terminology

Lacking universally accepted terminology to talk about classes, I'll make occasional use of Smalltalk and C++ terms. (I'd use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it...)

I also have to warn you that there's a terminological pitfall for object-oriented readers: the word "object" in Python does not necessarily mean a class instance. Like C++ and Modula-3, and unlike Smalltalk, not all types in Python are classes: the basic built-in types like integers and lists aren't, and even somewhat more exotic types like files aren't. However, *all* Python types share a little bit of common semantics that is best described by using the word object.

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has an (intended!) effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most types representing entities outside the program (files, windows, etc.). This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this obviates the need for two different argument passing mechanisms as in Pascal.

9.2 Python scopes and name spaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with name spaces, and you need to know how scopes and name spaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *name space* is a mapping from names to objects. Most name spaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of name spaces are: the set of built-in names (functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a name space. The important thing to know about name spaces is that there is absolutely no relation between names in different name spaces; for instance, two different modules may both define a function "maximize" without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same name space![1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the del statement, e.g. `del modname.the_answer`.

Name spaces are created at different moments and have different lifetimes. The name space containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global name space for a module is created when the module definition is read in; normally, module name spaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global name space. (The built-in names actually also live in a module; this is called `__builtin__`.)

The local name space for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have

[1] Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module's name space; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of name space implementation, and should be restricted to things like post-mortem debuggers...

their own local name space.

A *scope* is a textual region of a Python program where a name space is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the name space.

Although scopes are determined statically, they are used dynamically. At any time during execution, exactly three nested scopes are in use (i.e., exactly three name spaces are directly accessible): the innermost scope, which is searched first, contains the local names, the middle scope, searched next, contains the current module's global names, and the outermost scope (searched last) is the name space containing built-in names.

Usually, the local scope references the local names of the (textually) current function. Outside of functions, the local scope references the same name space as the global scope: the module's name space. Class definitions place yet another name space in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's name space, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that assignments always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of x from the name space referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope. (The `global` statement can be used to indicate that particular variables live in the global scope.)

9.3 A first look at classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

9.3.1 Class definition syntax

The simplest form of class definition looks like this:

```
class ClassName:
        <statement-1>
        .
        .
        .
        <statement-N>
```

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new name space is created, and used as the local scope — thus, all assignments to local variables go into this new name space. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the name space created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definitions was entered) is reinstated, and the class object is bound here to class name given in the class definition header (ClassName in the example).

9.3.2 Class objects

Class objects support two kinds of operations: attribute references and instantiation.

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 name space when the class object was created. So, if the class definition looked like this:

```
class MyClass:
        i = 12345
        def f(x):
                return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example, (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

9.3.3 Instance objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names.

The first I'll call *data attributes*. These correspond to "instance variables" in Smalltalk, and to "data members" in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` in the instance of `MyClass` created above, the following piece of code will print the value 16, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
        x.counter = x.counter * 2
print x.counter
del x.counter
```

The second kind of attribute references understood by instance objects are *methods*. A method is a function that "belongs to" an object. (In Python, the term method is not unique to class instances: other object types can have methods as well, e.g., list objects have methods called append, insert, remove, sort, and so on. However, below, we'll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are (user-defined) function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

9.3.4 Method objects

Usually, a method is called immediately, e.g.:

```
x.f()
```

In our example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later moment, for example:

```
xf = x.f
while 1:
        print xf()
```

will continue to print `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used...

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of *n* arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When an instance attribute is referenced that isn't a data attribute, its class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, it is unpacked again, a new argument list is constructed from the instance object and the original argument list, and the function object is called with this new argument list.

9.4 Random remarks

[These should perhaps be placed more carefully...]

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts, e.g., capitalize method names, prefix data attribute names with a small unique string (perhaps just an underscore), or use verbs for methods and nouns for data attributes.

Data attributes may be referenced by methods as well as by ordinary users ("clients") of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Conventionally, the first argument of methods is often called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. (Note, however, that by not following the convention your code may be less readable by other Python programmers, and it is also conceivable that a *class browser* program be written which relies upon such a convention.)

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
# Function defined outside the class
def f1(self, x, y):
        return min(x, x+y)

class C:
        f = f1
        def g(self):
                return 'hello world'
        h = g
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument, e.g.:

```
    class Bag:
            def empty(self):
                    self.data = []
            def add(self, x):
                    self.data.append(x)
            def addtwice(self, x):
                    self.add(x)
                    self.add(x)
```

The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects in a known initial state. Therefore a class may define a special method named `__init__`, like this:

```
            def __init__(self):
                    self.empty()
```

When a class defines an `__init__` method, class instantiation automatically invokes `__init__` for the newly-created class instance. So in the `Bag` example, a new and initialized instance can be obtained by:

```
    x = Bag()
```

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__`. For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0,-4.5)
>>> x.r, x.i
(3.0, -4.5)
>>>
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing the class definition. (The class itself is never used as a global scope!) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class!

9.5 Inheritance

Of course, a language feature would not be worthy of the name "class" without supporting inheritance. The syntax for a derived class definition looks as follows:

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

The name `BaseClassName` must be defined in a scope containing the derived class definition. Instead of a base class name, an expression is also allowed. This is useful when the base class is defined in another module, e.g.,

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, it is searched in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class, may in fact end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are "virtual functions".)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is defined or imported directly in the global scope.)

9.5.1 Multiple inheritance

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks as follows:

```
class DerivedClassName(Base1, Base2, Base3):
        <statement-1>
        .
```

```
        .
        .
        <statement-N>
```

The only rule necessary to explain the semantics is the resolution rule used for class attribute references. This is depth-first, left-to-right. Thus, if an attribute is not found in `DerivedClassName`, it is searched in `Base1`, then (recursively) in the base classes of `Base1`, and only if it is not found there, it is searched in `Base2`, and so on.

(To some people breadth first—searching `Base2` and `Base3` before the base classes of `Base1`—looks more natural. However, this would require you to know whether a particular attribute of `Base1` is actually defined in `Base1` or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of `Base2`. The depth-first rule makes no differences between direct and inherited attributes of `Base1`.)

It is clear that indiscriminate use of multiple inheritance is a maintenance nightmare, given the reliance in Python on conventions to avoid accidental name conflicts. A well-known problem with multiple inheritance is a class derived from two classes that happen to have a common base class. While it is easy enough to figure out what happens in this case (the instance will have a single copy of "instance variables" or data attributes used by the common base class), it is not clear that these semantics are in any way useful.

9.6 Odds and ends

Sometimes it is useful to have a data type similar to the Pascal "record" or C "struct", bundling together a couple of named data items. An empty class definition will do nicely, e.g.:

```
class Employee:
        pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that gets the data from a string buffer instead, and pass it as an argument. (Unfortunately, this technique has its limitations: a class can't define operations that are accessed by special syntax such as sequence subscripting or arithmetic operators, and assigning such a "pseudo-file" to `sys.stdin` will not cause the interpreter to read further input from it.)

Instance method objects have attributes, too: `m.im_self` is the object of which the method is an instance, and `m.im_func` is the function object corresponding to the method.

Chapter 10

Recent Additions

Python is an evolving language. Since this tutorial was last thoroughly revised, several new features have been added to the language. While ideally I should revise the tutorial to incorporate them in the mainline of the text, lack of time currently requires me to take a more modest approach. In this chapter I will briefly list the most important improvements to the language and how you can use them to your benefit.

10.1 The Last Printed Expression

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 = 17.5 / 100
>>> price = 3.50
>>> price * tax
0.6125
>>> price + _
4.1125
>>> round(_, 2)
4.11
>>>
```

For reasons too embarrassing to explain, this variable is implemented as a built-in (living in the module `__builtin__`), so it should be treated as read-only by the user. I.e. don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

10.2 String Literals

10.2.1 Double Quotes

Python can now also use double quotes to surround string literals, e.g. `"this doesn't hurt a bit"`. There is no semantic difference between strings surrounded by single or double quotes.

10.2.2 Continuation Of String Literals

String literals can span multiple lines by escaping newlines with backslashes, e.g.

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
 significant.\n"
print hello
```

which would print the following:

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

10.2.3 Triple-quoted strings

In some cases, when you need to include really long strings (e.g. containing several paragraphs of informational text), it is annoying that you have to terminate each line with `\n\`, especially if you would like to reformat the text occasionally with a powerful text editor like Emacs. For such situations, "triple-quoted" strings can be used, e.g.

```
hello = """

    This string is bounded by triple double quotes (3 times ").
Unescaped newlines in the string are retained, though \
it is still possible\nto use all normal escape sequences.

    Whitespace at the beginning of a line is
significant.  If you need to include three opening quotes
you have to escape at least one of them, e.g. \""".

    This string ends in a newline.
"""
```

Triple-quoted strings can be surrounded by three single quotes as well, again without semantic difference.

10.2.4 String Literal Juxtaposition

One final twist: you can juxtapose multiple string literals. Two or more adjacent string literals (but not arbitrary expressions!) separated only by whitespace will be concatenated (without intervening whitespace) into a single string object at compile time. This makes it possible to continue a long string on the next line without sacrificing indentation or performance, unlike the use of the string concatenation operator + or the continuation of the literal itself on the next line (since leading whitespace is significant inside all types of string literals). Note that this feature, like all string features except triple-quoted strings, is borrowed from Standard C.

10.3 The Formatting Operator

10.3.1 Basic Usage

The chapter on output formatting is really out of date: there is now an almost complete interface to C-style printf formats. This is done by overloading the modulo operator (`%`) for a left operand which is a string, e.g.

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

If there is more than one format in the string you pass a tuple as right operand, e.g.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> for name, phone in table.items():
...     print '%-10s ==> %10d' % (name, phone)
...
Jack       ==>       4098
Dcab       ==>    8637678
Sjoerd     ==>       4127
>>>
```

Most formats work exactly as in C and require that you pass the proper type (however, if you don't you get an exception, not a core dump). The `%s` format is more relaxed: if the corresponding argument is not a string object, it is converted to string using the `str()` built-in function. Using `*` to pass the width or precision in as a separate (integer) argument is supported. The C formats `%n` and `%p` are not supported.

10.3.2 Referencing Variables By Name

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by using an extension of C formats using the form `%(name)format`, e.g.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
>>>
```

This is particularly useful in combination with the new built-in `vars()` function, which returns a dictionary containing all local variables.

10.4 Optional Function Arguments

It is now possible to define functions with a variable number of arguments. There are two forms, which can be combined.

10.4.1 Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined, e.g.

```
def ask_ok(prompt, retries = 4, complaint = 'Yes or no, please!'):
        while 1:
                ok = raw_input(prompt)
                if ok in ('y', 'ye', 'yes'): return 1
                if ok in ('n', 'no', 'nop', 'nope'): return 0
                retries = retries - 1
                if retries < 0: raise IOError, 'refusenik user'
                print complaint
```

This function can be called either like this: `ask_ok('Do you really want to quit?')` or like this: `ask_ok('OK to overwrite the file?', 2)`.

The default values are evaluated at the point of function definition in the *defining* scope, so that e.g.

```
i = 5
def f(arg = i): print arg
i = 6
f()
```

will print `5`.

10.4.2 Arbitrary Argument Lists

It is also possible to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur, e.g.

```
def fprintf(file, format, *args):
        file.write(format % args)
```

This feature may be combined with the previous, e.g.

```
def but_is_it_useful(required, optional = None, *remains):
        print "I don't know"
```

10.5 Lambda And Functional Programming Tools

10.5.1 Lambda Forms

By popular demand, a few features commonly found in functional programming languages and Lisp have been added to Python. With the `lambda` keyword, small anonymous functions can be created. Here's a function that returns the sum of its two arguments: `lambda a, b: a+b`. Lambda forms can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda forms cannot reference variables from the containing scope, but this can be overcome through the judicious use of default argument values, e.g.

```
def make_incrementor(n):
        return lambda x, incr=n: x+incr
```

10.5.2 Map, Reduce and Filter

Three new built-in functions on sequences are good candidate to pass lambda forms.

Map.

`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:

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

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or `None` if some sequence is shorter than another). If `None` is passed for the function, a function returning its argument(s) is substituted.

Combining these two special cases, we see that `map(None, list1, list2)` is a convenient way of turning a pair of lists into a list of pairs. For example:

```
>>> seq = range(8)
>>> map(None, seq, map(lambda x: x*x, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
>>>
```

Filter.

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

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

Reduce.

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

```
>>> reduce(lambda x, y: x+y, range(1, 11))
55
>>>
```

If there's only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on. For example,

```
>>> def sum(seq):
...     return reduce(lambda x, y: x+y, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
>>>
```

10.6 Continuation Lines Without Backslashes

While the general mechanism for continuation of a source line on the next physical line remains to place a backslash on the end of the line, expressions inside matched parentheses (or square brackets, or curly braces) can now also be continued without using a backslash. This is particularly useful for calls to functions with many arguments, and for initializations of large tables.

For example:

```
month_names = ['Januari', 'Februari', 'Maart',
               'April',   'Mei',      'Juni',
               'Juli',    'Augustus', 'September',
               'Oktober', 'November', 'December']
```

and

```
CopyInternalHyperLinks(self.context.hyperlinks,
                       copy.context.hyperlinks,
                       uidremap)
```

10.7 Regular Expressions

While C's printf-style output formats, transformed into Python, are adequate for most output formatting jobs, C's scanf-style input formats are not very powerful. Instead of scanf-style input, Python offers Emacs-style regular expressions as a powerful input and scanning mechanism. Read the corresponding section in the Library Reference for a full description.

10.8 Generalized Dictionaries

The keys of dictionaries are no longer restricted to strings — they can be any immutable basic type including strings, numbers, tuples, or (certain) class instances. (Lists and dictionaries are not acceptable as dictionary keys, in order to avoid problems when the object used as a key is modified.)

Dictionaries have two new methods: `d.values()` returns a list of the dictionary's values, and `d.items()` returns a list of the dictionary's (key, value) pairs. Like `d.keys()`, these operations are slow for large dictionaries. Examples:

```
>>> d = {100: 'honderd', 1000: 'duizend', 10: 'tien'}
>>> d.keys()
[100, 10, 1000]
>>> d.values()
['honderd', 'tien', 'duizend']
>>> d.items()
[(100, 'honderd'), (10, 'tien'), (1000, 'duizend')]
>>>
```

10.9 Miscellaneous New Built-in Functions

The function `vars()` returns a dictionary containing the current local variables. With a module argument, it returns that module's global variables. The old function `dir(x)` returns `vars(x).keys()`.

The function `round(x)` returns a floating point number rounded to the nearest integer (but still expressed as a floating point number). E.g. `round(3.4) == 3.0` and `round(3.5) == 4.0`. With a second argument it rounds to the specified number of digits, e.g. `round(math.pi, 4) == 3.1416` or even `round(123.4, -2) == 100.0`.

The function `hash(x)` returns a hash value for an object. All object types acceptable as dictionary keys have a hash value (and it is this hash value that the dictionary implementation uses).

The function `id(x)` return a unique identifier for an object. For two objects `x` and `y`, `id(x) == id(y)` if and only if `x is y`. (In fact the object's address is used.)

The function `hasattr(x, name)` returns whether an object has an attribute with the given name (a string value). The function `getattr(x, name)` returns the object's attribute with the given name. The function `setattr(x, name, value)` assigns a value to an object's attribute with the given name. These three functions are useful if the attribute names are not known beforehand. Note that `getattr(x, 'spam')` is equivalent to `x.spam`, and `setattr(x, 'spam', y)` is equivalent to `x.spam = y`. By definition, `hasattr(x, name)` returns true if and only if `getattr(x, name)` returns without raising an exception.

10.10 Else Clause For Try Statement

The `try...except` statement now has an optional `else` clause, which must follow all `except` clauses. It is useful to place code that must be executed if the `try` clause does not raise an exception. For example:

```
for arg in sys.argv:
        try:
                f = open(arg, 'r')
        except IOError:
                print 'cannot open', arg
        else:
                print arg, 'has', len(f.readlines()), 'lines'
                f.close()
```

10.11 New Class Features in Release 1.1

Some changes have been made to classes: the operator overloading mechanism is more flexible, providing more support for non-numeric use of operators (including calling an object as if it were a function), and it is possible to trap attribute accesses.

10.11.1 New Operator Overloading

It is no longer necessary to coerce both sides of an operator to the same class or type. A class may still provide a `__coerce__` method, but this method may return objects of different types or classes if it feels like it. If no `__coerce__` is defined, any argument type or class is acceptable.

In order to make it possible to implement binary operators where the right-hand side is a class instance but the left-hand side is not, without using coercions, right-hand versions of all binary operators may be defined. These have an 'r' prepended to their name, e.g. `__radd__`.

For example, here's a very simple class for representing times. Times are initialized from a number of seconds (like time.time()). Times are printed like this: `Wed Mar 15 12:28:48 1995`. Subtracting two Times gives their difference in seconds. Adding or subtracting a Time and a number gives a new Time. You can't add two times, nor can you subtract a Time from a number.

```
import time

class Time:
    def __init__(self, seconds):
        self.seconds = seconds
    def __repr__(self):
        return time.ctime(self.seconds)
    def __add__(self, x):
        return Time(self.seconds + x)
    __radd__ = __add__              # support for x+t
    def __sub__(self, x):
        if hasattr(x, 'seconds'): # test if x could be a Time
            return self.seconds - x.seconds
        else:
            return self.seconds - x

now = Time(time.time())
tomorrow = 24*3600 + now
yesterday = now - today
print tomorrow - yesterday          # prints 172800
```

10.11.2 Trapping Attribute Access

You can define three new "magic" methods in a class now: `__getattr__(self, name)`, `__setattr__(self, name, value)` and `__delattr__(self, name)`.

The `__getattr__` method is called when an attribute access fails, i.e. when an attribute access would otherwise raise AttributeError — this is *after* the instance's dictionary and its class hierarchy have been searched for the named attribute. Note that if this method attempts to access any undefined instance attribute it will be called recursively!

The `__setattr__` and `__delattr__` methods are called when assignment to, respectively deletion of an attribute are attempted. They are called *instead* of the normal action (which is to insert or delete the attribute in the instance dictionary). If either of these methods most set or delete any attribute, they can only do so by using the instance dictionary directly — `self.__dict__` — else they would be called recursively.

For example, here's a near-universal "Wrapper" class that passes all its attribute accesses to another object. Note how the `__init__` method inserts the wrapped object in `self.__dict__` in order to avoid endless recursion (`__setattr__` would call `__getattr__` which would call itself recursively).

```
class Wrapper:
    def __init__(self, wrapped):
        self.__dict__['wrapped'] = wrapped
    def __getattr__(self, name):
        return getattr(self.wrapped, name)
    def __setattr__(self, name, value):
        setattr(self.wrapped, name, value)
    def __delattr__(self, name):
        delattr(self.wrapped, name)

import sys
f = Wrapper(sys.stdout)
f.write('hello world\n')            # prints 'hello world'
```

A simpler example of `__getattr__` is an attribute that is computed each time (or the first time) it it accessed. For instance:

```
from math import pi

class Circle:
    def __init__(self, radius):
        self.radius = radius
    def __getattr__(self, name):
        if name == 'circumference':
            return 2 * pi * self.radius
        if name == 'diameter':
            return 2 * self.radius
        if name == 'area':
           return pi * pow(self.radius, 2)
        raise AttributeError, name
```

10.11.3 Calling a Class Instance

If a class defines a method `__call__` it is possible to call its instances as if they were functions. For example:

```
class PresetSomeArguments:
```

```
    def __init__(self, func, *args):
        self.func, self.args = func, args
    def __call__(self, *args):
        return apply(self.func, self.args + args)

f = PresetSomeArguments(pow, 2)     # f(i) computes powers of 2
for i in range(10): print f(i),     # prints 1 2 4 8 16 32 64 128 256 512
print                               # append newline
```

Chapter 11

New in Release 1.2

This chapter describes even more recent additions to the Python language and library.

11.1 New Class Features

The semantics of `__coerce__` have been changed to be more reasonable. As an example, the new standard module `Complex` implements fairly complete complex numbers using this. Additional examples of classes with and without `__coerce__` methods can be found in the `Demo/classes` subdirectory, modules `Rat` and `Dates`.

If a class defines no `__coerce__` method, this is equivalent to the following definition:

```
def __coerce__(self, other): return self, other
```

If `__coerce__` coerces itself to an object of a different type, the operation is carried out using that type — in release 1.1, this would cause an error.

Comparisons involving class instances now invoke `__coerce__` exactly as if `cmp(x, y)` were a binary operator like + (except if `x` and `y` are the same object).

11.2 Unix Signal Handling

On Unix, Python now supports signal handling. The module `signal` exports functions `signal`, `pause` and `alarm`, which act similar to their Unix counterparts. The module also exports the conventional names for the various signal classes (also usable with `os.kill()`) and `SIG_IGN` and `SIG_DFL`. See the section on `signal` in the Library Reference Manual for more information.

11.3 Exceptions Can Be Classes

User-defined exceptions are no longer limited to being string objects — they can be identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the raise statement:

```
raise Class, instance

raise instance
```

In the first form, `instance` must be an instance of `Class` or of a class derived from it. The second form is a shorthand for

```
raise instance.__class__, instance
```

An except clause may list classes as well as string objects. A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Note that if the except clauses were reversed (with "`except B`" first), it would have printed B, B, B — the first matching except clause is triggered.

When an error message is printed for an unhandled exception which is a class, the class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function `str()`.

In this release, the built-in exceptions are still strings.

11.4 Object Persistency and Object Copying

Two new modules, `pickle` and `shelve`, support storage and retrieval of (almost) arbitrary Python objects on disk, using the `dbm` package. A third module, `copy`, provides flexible object copying operations. More information on these modules is provided in the Library Reference Manual.

11.4.1 Persistent Objects

The module `pickle` provides a general framework for objects to disassemble themselves into a stream of bytes and to reassemble such a stream back into an object. It copes with reference sharing, recursive objects and instances of user-defined classes, but not (directly) with objects that have "magical" links into the operating system such as open files, sockets or windows.

The `pickle` module defines a simple protocol whereby user-defined classes can control how they are disassembled and assembled. The method `__getinitargs__()`, if defined, returns the argument list for the constructor to be used at assembly time (by default the constructor is called without arguments). The methods `__getstate__()` and `__setstate__()` are used to pass additional state from disassembly to assembly; by default the instance's `__dict__` is passed and restored.

Note that `pickle` does not open or close any files — it can be used equally well for moving objects around on a network or store them in a database. For ease of debugging, and the inevitable occasional manual patch-up, the constructed byte streams consist of printable ASCII characters only (though it's not designed to be pretty).

The module `shelve` provides a simple model for storing objects on files. The operation `shelve.open(filename)` returns a "shelf", which is a simple persistent database with a dictionary-like interface. Database keys are strings, objects stored in the database can be anything that `pickle` will handle.

11.4.2 Copying Objects

The module `copy` exports two functions: `copy()` and `deepcopy()`. The `copy()` function returns a "shallow" copy of an object; `deepcopy()` returns a "deep" copy. The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A shallow copy constructs a new compound object and then (to the extent possible) inserts *the same objects* into in that the original contains.
- A deep copy constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.

Both functions have the same restrictions and use the same protocols as `pickle` — user-defined classes can control how they are copied by providing methods named `__getinitargs__()`, `__getstate__()` and `__setstate__()`.

11.5 Documentation Strings

A variety of objects now have a new attribute, `__doc__`, which is supposed to contain a documentation string (if no documentation is present, the attribute is `None`). New syntax, compatible with the old interpreter, allows for convenient initialization of the `__doc__` attribute of modules, classes and functions by placing a string literal by itself as the first statement in the suite. It must be a literal — an expression yielding a string object is not accepted as a documentation string, since future tools may need to derive documentation from source by parsing.

Here is a hypothetical, amply documented module called `Spam`:

```
"""Spam operations.

This module exports two classes, a function and an exception:

class Spam: full Spam functionality --- three can sizes
class SpamLight: limited Spam functionality --- only one can size

def open(filename): open a file and return a corresponding Spam or
SpamLight object

GoneOff: exception raised for errors; should never happen

Note that it is always possible to convert a SpamLight object to a
Spam object by a simple method call, but that the reverse operation is
generally costly and may fail for a number of reasons.
"""

class SpamLight:
    """Limited spam functionality.

    Supports a single can size, no flavor, and only hard disks.
    """

    def __init__(self, size=12):
        """Construct a new SpamLight instance.

        Argument is the can size.
        """
        # etc.

    # etc.

class Spam(SpamLight):
    """Full spam functionality.

    Supports three can sizes, two flavor varieties, and all floppy
    disk formats still supported by current hardware.
    """

    def __init__(self, size1=8, size2=12, size3=20):
        """Construct a new Spam instance.
```

```
        Arguments are up to three can sizes.
        """
        # etc.

    # etc.

def open(filename = "/dev/null"):
    """Open a can of Spam.

    Argument must be an existing file.
    """
    # etc.

class GoneOff:
    """Class used for Spam exceptions.

    There shouldn't be any.
    """
    pass
```

After executing "`import Spam`", the following expressions return the various documentation strings from the module:

```
Spam.__doc__
Spam.SpamLight.__doc__
Spam.SpamLight.__init__.__doc__
Spam.Spam.__doc__
Spam.Spam.__init__.__doc__
Spam.open.__doc__
Spam.GoneOff.__doc__
```

There are emerging conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one of more of paragraphs describing the objects calling conventions, its side effects, etc.

Some people like to copy the Emacs convention of using UPPER CASE for function parameters — this often saves a few words or lines.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace "equivalent" to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

In this release, few of the built-in or standard functions and modules have documentation strings.

11.6 Customizing Import and Built-Ins

In preparation for a "restricted execution mode" which will be usable to run code received from an untrusted source (such as a WWW server or client), the mechanism by which modules are imported has been redesigned. It is now possible to provide your own function `__import__` which is called whenever an `import` statement is executed. There's a built-in function `__import__` which provides the default implementation, but more interesting, the various steps it takes are available separately from the new built-in module `imp`. (See the section on `imp` in the Library Reference Manual for more information on this module.)

When you do `dir()` in a fresh interactive interpreter you will see another "secret" object that's present in every module: `__builtins__`. This is either a dictionary or a module containing the set of built-in objects used by functions defined in current module. Although normally all modules are initialized with a reference to the same dictionary, it is now possible to use a different set of built-ins on a per-module basis. Together with the fact that the `import` statement uses the `__import__` function it finds in the importing modules' dictionary of built-ins, this forms the basis for a future restricted execution mode.

11.7 Python and the World-Wide Web

There is a growing number of modules available for writing WWW tools. The previous release already sported modules `gopherlib`, `ftplib`, `httplib` and `urllib` (which unifies the other three) for accessing data through the commonest WWW protocols. This release also provides `cgi`, to ease the writing of server-side scripts that use the Common Gateway Interface protocol, supported by most WWW servers. The module `urlparse` provides precise parsing of a URL string into its components (address scheme, network location, path, parameters, query, and fragment identifier).

A rudimentary, parser for HTML files is available in the module `htmllib`. It currently supports a subset of HTML 1.0 (if you bring it up to date, I'd love to receive your fixes!). Unfortunately Python seems to be too slow for real-time parsing and formatting of HTML such as required by interactive WWW browsers — but it's good enough to write a "robot" (an automated WWW browser that searches the web for information).

11.8 Miscellaneous

- The `socket` module now exports all the needed constants used for socket operations, such as `SO_BROADCAST`.

- The functions `popen()` and `fdopen()` in the `os` module now follow the pattern of the built-in function `open()`: the default mode argument is `'r'` and the optional third argument specifies the buffer size, where `0` means unbuffered, `1` means line-buffered, and any larger number means the size of the buffer in bytes.