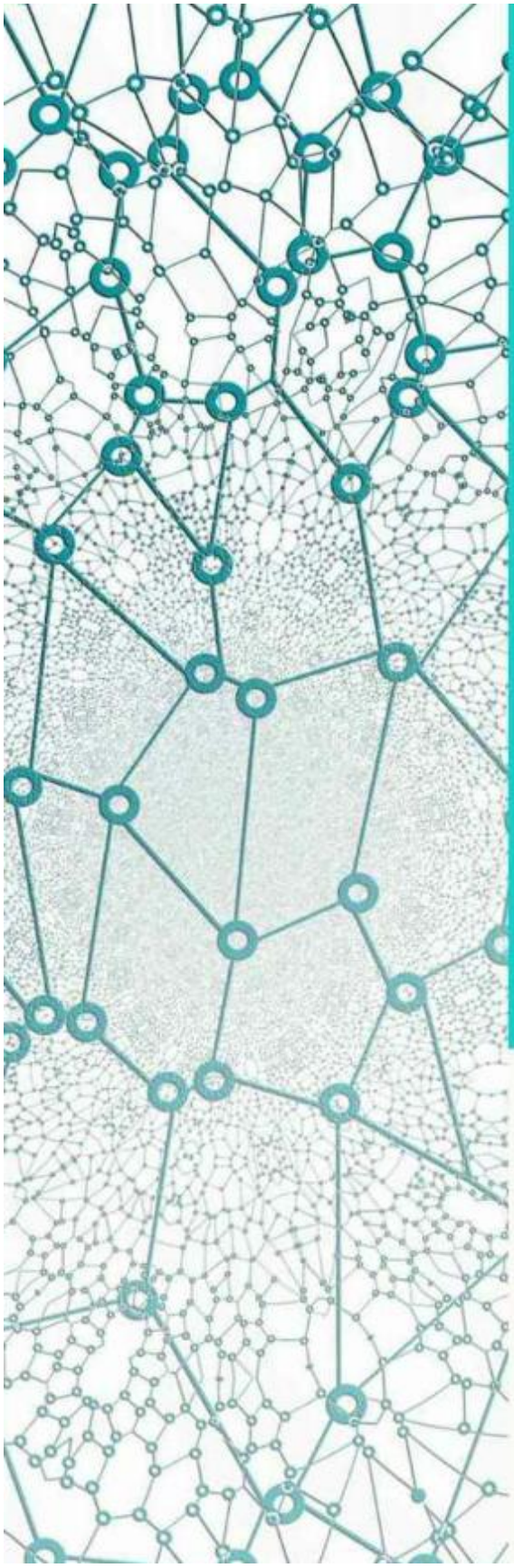


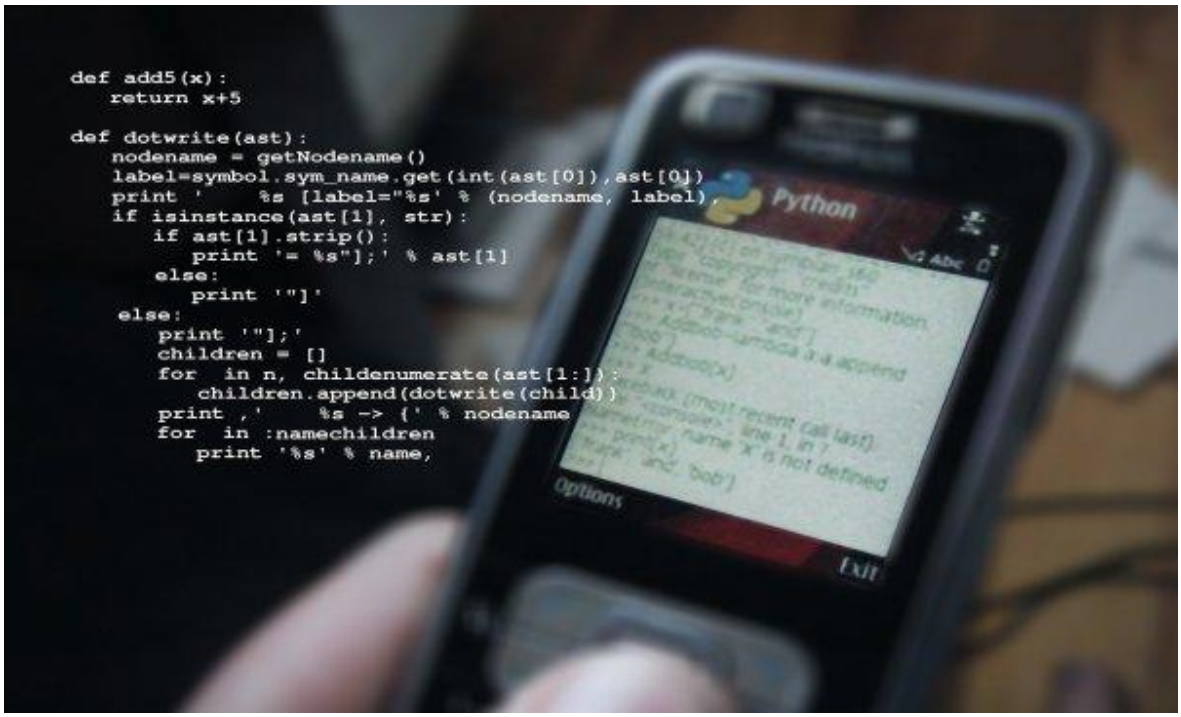
PALPABLE PYTHON

beat it in 7 days

Learn it fast, Use it more Effective Step by Step
Practical Programming for Newbies,
Introduction Encoding functions Data Science

Table of content

The python tutorial



Python is a powerful and easy-to-learn programming language. It has high-level data structures and a simple but effective object-oriented programming approach. Because its syntax is elegant, its typing is dynamic and interpreted, Python is an ideal language for scripting and rapid application development in many domains and on many platforms.

The Python interpreter and its extensive standard library are freely available, as sources or binaries, for all major platforms, from the website http://www.python.org/ and can be freely redistributed. The same site distributes and contains links to third-party modules, programs, tools, and additional documentation.

The Python interpreter can be easily extended by new functions and data types implemented in C or C ++ (or any other callable language from C). Python is also suitable as an extension language for customizing applications.

This tutorial informally introduces the basic concepts and features of the Python language and its ecosystem. It helps to take over the Python interpreter for use on practical cases. As the examples are independent, the tutorial is suitable for offline reading.

For a description of standard library objects and modules, see The Python Standard Library. The Python Language Reference presents a more formal definition of language. To write extensions in C or C ++, read Extending and Embedding the Python Interpreter and Python / C API Reference Manual. Books are also available that cover Python in detail.

The ambition of this tutorial is not to be exhaustive and cover every feature, or even all the most

used features. It seeks, however, to introduce many of the most notable features and give you a good idea of the flavor and style of language. After reading it, you will be able to read and write Python modules and programs and you will be ready to learn more about the Python library modules described in The Python Standard Library.

1. Appetizer

If you do a lot of computer work, you will eventually want to automate some tasks. For example, you may need to search and replace a large number of text files, or rename and reorder photos in a sophisticated way. Maybe you need to create a small database or a graphics application, or a simple game.

If you are a professional developer, you may need to work with some C / C ++ / Java libraries, but you find that the usual write / compile / test / recompile cycle is too heavy. Perhaps you are writing a test suite for such a library and find that writing the test code is painful. Or you have written software that needs to be extensible through a scripting language, but you do not want to design or implement a new language for your application.

Python is the perfect language for you.

You can write a Unix shell script or Windows batch files for some of these tasks. Shell scripts are appropriate for moving files and editing text data, but not for an application with a GUI or for games. You can write a program in C / C ++ / Java, but it can take a lot of time, if only to have a first model. Python is easier to use, it is available on Windows, Mac OS X and Unix, and it will help you finish your work faster.

Python is easy to use, but it is a real programming language, offering much better structure and support for large programs than shell scripts or batch files. On the other hand, Python offers many more error checking methods than the C language and, being a very high level language, natively has very advanced data types such as flexible arrays or dictionaries. Thanks to its more universal data types, Python can be used for many more different domains than Awk or even Perl. Yet, many things are at least as easy in Python as in these languages.

Python allows you to split your program into modules that can be reused in other programs in Python. It comes with an extensive collection of standard modules that you can use as a basis for your programs, or as examples to learn how to program. Some of these modules provide services such as I / O, system calls, sockets, and even access to tools like Tk to create GUIs.

Python is an interpreted language, which can save you a lot of time during the development of the program because no compilation or editing of links is necessary. The interpreter can be used interactively, so you can experiment with language features, write throw programs, or test functions during incremental development. It is also a practical desk calculator.

Python allows you to write compact and readable programs. Programs written in Python are generally much shorter than the equivalent in C, C ++, or Java, for several reasons:

High level data types allow you to express complex operations in a single statement
the instructions are grouped together by indentation, rather than by the use of braces;
no declaration of variable or argument is necessary.
Python is extensible: if you know how to write a program in C, it is easy to add to the interpreter a new primitive function or a module, either to perform critical operations at maximum speed, or to link programs in Python to libraries available only in binary form (eg graphic libraries dedicated to hardware). Once you are comfortable with these principles, you can connect the Python interpreter to an application written in C and use it as a language of extensions or commands for that application.

In this regard, the name of the language comes from the BBC program "Monty Python's Flying Circus" and has nothing to do with reptiles. Referring to Monty Python sketches in documentation is not only allowed, it's encouraged!

Your sudden enthusiasm about Python will push you to look at it in a little more detail. Since the best way to learn a language is to use it, the tutorial prompts you to play with the interpreter during playback.

In the next chapter, we will explain how to use the interpreter. This is not the most exciting section, but it is a must to test the examples shown below.

The rest of the tutorial introduces various features of the Python language and system through examples, from simple expressions, statements, or data types, to functions and modules, to finally addressing advanced concepts such as exceptions and classes.

2. Use the Python interpreter

2.1. Summon interpreter

The Python interpreter is usually installed as /usr/local/bin/python3.5 on machines where possible. Adding / usr / local / bin in your PATH makes it possible to start it by typing the command:
python3.5
in the shell. The choice of the directory where the interpreter is an installation option, other paths are possible; see with your local Python guru or system administrator. (For example, / usr / local / python is a current location.)

On Windows machines, the installation of Python is often in C: \ Python35, however, it is possible to change this at installation. To add this folder to your PATH, you can copy this command into a DOS interpreter

set path =% path%; C: \ python35

Typing an end-of-file character (Ctrl-D on Unix, Ctrl-Z on Windows) following a primary command prompt causes the interpreter to close with a null error status. If that does not work, you can close the interpreter by typing the command quit ().

The editing capabilities of the interpreter include interactive editing, substitution from history, and

completion, on systems that manage readline. The quickest way to test if editing the command line is being handled might be to type Control-P following the first command prompt you encounter. If this beep, you have the command line edition; see the Appendix Interactive Editing of Entries and History Substitution for an introduction to keys. If nothing seems to happen or if ^ P is displayed, the command line is not available; you will only be able to use the backspace key to delete characters from the current line.

The interpreter operates in a similar way to the Unix shell: when called with the standard input connected to a tty device, it reads and executes the commands interactively; when called with a file name as an argument or with a file as a standard input, it reads and executes a script from that file.

Another way to run the interpreter is python -c command [arg] .... This executes the command statements in a similar way to the -c shell option. Because Python instructions often contain spaces and other special characters for the shell, it is generally advisable to put a command in single quotation marks.

Some Python modules are also useful as scripts. They can be called with python -m module [arg] ... which runs the module source file as if you typed its full name in the command line.

When a script file is used, it is sometimes useful to be able to launch the script and then enter the interactive mode afterwards. This is possible by passing -i before the script.

All parameters that can be used on the command line are documented in Command Line and Environment.

2.1.1. Passing arguments

When they are known to the interpreter, the script name and additional arguments are represented as a list assigned to the argv variable of the sys module. You can access it by running import sys. The list contains at least one element; when no script or arguments are given, sys.argv [0] is an empty string. When '-' (which represents the standard input) is passed as the script name, sys.argv [0] contains '-'. When -c command is used, sys.argv [0] contains '-c'. Finally, when -m module is used, the module's full name is assigned to sys.argv [0]. Options found after -c command or -m module are not read as Python interpreter options but left in sys.argv for use by the module or command.

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 requests the following command with the primary prompt, usually three plus-big-signs (>>>); for the continuation lines, it displays the secondary prompt, by default three points (...). The interpreter displays a welcome message indicating its version number and a copyright notice before displaying the first prompt:

$ python3.5

Python 3.5 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>

The continuation lines are needed to enter a multi-line construct. For example, look at this if statement

>>>
>>> **the_world_is_flat = True**
>>> **if the_world_is_flat:**
... print ("Be careful not to fall off!")
…

Be careful not to fall off!
For more information on interactive mode, see Interactive Mode.

2.2. The interpreter and its environment

2.2.1. Encoding the source code

By default Python considers that its source files are encoded in UTF-8. In this encoding, characters from most languages can be used together in strings, identifiers, and comments, although the standard library uses only ASCII characters in its identifiers, a good habit that any portable code should to follow. To correctly display all these characters, your editor must recognize that the file is in UTF-8, and use a character font that includes all the characters used in the file.

It is possible to use another encoding in a Python source file. The best way to do this is to place an extra special comment just after the #! to define the encoding of the source file:

- * - coding: encoding - * -
With this declaration, all characters in the source file are treated as in the encoding encoding instead of UTF-8. The list of available encodings can be found in the reference of the Python library in the section about codecs.

For example, if your editor does not support UTF-8 and insists on using another encoding, say Windows-1252, you can write:

- * - coding: cp-1252 - * -
and continue to use all Windows-1252 characters in your code. This special comment specifying the encoding must be in the first or second line of the file.

3. Informal introduction to Python

In the examples that follow, the inputs and outputs are distinguished by the presence or absence of prompt (>>> and ...): to reproduce the examples, you must type everything after the prompt, when it appears; lines that do not display a prompt are the outputs of the interpreter. Note that a secondary prompt displayed alone on a line in an example indicates that you must enter a blank line; this is used to terminate a multi-line command.

Many examples in this manual, even those entered at the interpreter prompt, include comments. Python comments begin with a pound sign, #, and extend to the end of the line. A comment may appear at the beginning of a line or as a result of a space or code, but not within a literal string. A hash character inside a string is just a pound sign. Because comments are only used to explain the code and are not interpreted by Python, they can be ignored when you type the examples.

Some examples :

```
# this is the first comment
spam = 1 # and this is the second comment
        # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1. Use Python as a calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, >>>. It should not be long.

3.1.1. Numbers

The interpreter acts like a simple calculator: you can enter an expression and it displays the value. The syntax of the expressions is simple: the +, -, * and / operators work as in most languages (for example, Pascal or C); parentheses can be used to group. For example :

```
>>>
>>> 2 + 2
4
>>> 50 - 5 * 6
20
>>> (50 - 5 * 6) / 4
5.0
>>> 8/5 # division always returns a floating point number
1.6
```

Integer numbers (such as 2, 4, 20) are of type int, while decimals (such as 5.0, 1.6) are of type float. More details are given on numeric types later in this tutorial.

Divisions (/) always give floats. Use the // operator to perform integer divisions, and thus get an integer result. To get the rest of this entire division, use the% operator

```
>>>
>>> 17/3 # classic division returns a float
5.666666666666667
```

```
>>>
>>> 17 // 3 # floor division discards the fractional share
5
>>> 17% 3 # the% operator returns the remainder of the division
2
>>> 5 * 3 + 2 # result * divisor + remainder
17
```

With Python it is possible to calculate powers with the operator ** [1]

```
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128
```

The equal sign (=) is used to assign a value to a variable. After that, no results are displayed until the following prompt:

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

If a variable is not "defined" (if no value has been assigned to it), using it will generate an error:

```
>>>
>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point numbers; operators with mixed operand types convert the entire operand to a floating point:

```
>>>
>>> 4 * 3.75 - 1
14.0
```

In interactive mode, the last expression displayed is assigned to the variable _. Which means that when you use Python as a calculator, it is sometimes easier to continue calculations, for example:

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

This variable should be considered as a read-only variable by the user. Do not assign a value explicitly - you would create an independent local variable with the same name that would hide the native variable and its magic operation.

In addition to int and float, there are Decimal and Fraction. Python also handles complex numbers, using the suffix j or J to indicate the imaginary part (such as: 3 + 5j).

3.1.2. Strings

Beyond numbers, Python can also manipulate strings of characters, which can be expressed in different ways. They can be written in single quotation marks ('...') or quotation marks ("...") without distinction [2]. \ can be used to protect a quotation mark:

```
>>>
>>> 'spam eggs' # single quotes
'spam eggs'
>>> 'doesn \' t '#use \' to escape the single quote ...
"does not"
>>> "does not" # ... or use double quotes instead
"does not"
>>> '"Yes," he said.
"Yes," he said.
>>> "\" Yes, \ "he said."
"Yes," he said.
>>> '"Isn \' t," she said.
'' Isn''t, 'she said.'
```

In interactive mode, the interpreter displays quoted strings and protects quotation marks and other special characters with backslashes. Although it may look different from what was given (the quotes can change) The string is enclosed in quotes if it contains a single quotation mark but no quotation marks, otherwise it is displayed as single quotes. The print () function displays strings in a more readable way, removing quotation marks and displaying special characters that were protected by a backslash:

```
>>>
>>> '"Isn \' t," she said.
'' Isn''t, 'she said.'
>>> print ('"Isn \' t," she said. ')
"Is not," she said.
>>> s = 'First line. \ nSecond line.' # \ n means newline
>>> s # without print (), \ n is included in the output
'First line. \ NSecond line.'
>>> print (s) # with print (), \ n produces a new line
```

First line.
Second line.
To prevent characters preceded by \ to be interpreted as special, use raw strings prefixing the string of a r

>>>
>>> **print ('C: \ some \ name') # here \ n means newline!**
C: \ some
soul
>>> **print (r'C: \ some \ name ') # note the r before the quote**
C: \ some \ name

Strings can span multiple lines. You can use triple quotation marks, single or double: " '...' " or "" "..." "". Line breaks are automatically included, but can be prevented by adding \ at the end of the line. The following example:

```
print ( "" "\
Usage: thingy [OPTIONS]
    -h Display this usage message
    -H hostname Hostname to connect to
"" ")
```

produces the following display (note that the first line break is not included):

```
Usage: thingy [OPTIONS]
    -h Display this usage message
    -H hostname Hostname to connect to
```

Channels can be concatenated (pasted together) with the + operator, and repeated with the operator *:

```
>>>
>>> # 3 times 'a', followed by 'ium'
>>> 3 * 'an' + 'ium'
'Unununium'
```

Several character strings, written literally (ie in quotation marks), side by side, are automatically concatenated.

```
>>>
>>> 'Py' 'tuna'
'Python'
```

However, this only works with literal strings, not variables or expressions:

```
>>>
>>> prefix = 'Py'
>>> prefix 'tuna' # can not concatenate a variable and a string literal
  ...
SyntaxError: invalid syntax
>>> ('a' * 3) 'ium'
  ...
```

SyntaxError: invalid syntax
To concatenate variables, or variables with literal strings, use the +:

```
>>>
>>> prefix + 'tuna'
'Python'
```

This feature is especially interesting for cutting too long strings:

```
>>>
>>> text = ('Put several strings within parentheses'
... 'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

Strings can be indexed (access to characters by their position), the first character of a string is at position 0. There is no distinct type for characters, a character is simply a string of length 1

```
>>>
>>> word = 'Python'
>>> word [0] # character in position 0
'P'
>>> word [5] # character in position 5
'not'
```

Indices can also be negative, to count down from the right. For example :

```
>>>
>>> word [-1] # last character
'not'
>>> word [-2] # second-last character
'O'
>>> word [-6]
'P'
```

Note that since -0 equals 0, the negative indices start with -1.

In addition to accessing an element by its index, it is also possible to decide a list. Accessing a string by an index allows to obtain a character, while slicing it allows to obtain a sub-string:

```
>>>
>>> word [0: 2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word [2: 5] # characters from position 2 (included) to 5 (excluded)
'Tho'
```

Note that the beginning is always included and the end always excluded. This ensures that s [: i] + s [i:] is always equal to s

```
>>> word [: 2] + word [2:]
'Python'
>>> word [: 4] + word [4:]
'Python'
```

Slice indexes have useful default values; the first clue when omitted equals zero, the

second clue the size of the string:

```
>>>
>>> word [: 2] # character from the beginning to position 2 (excluded)
'Py'
>>> word [4:] # characters from position 4 (included) to the end
'we'
>>> word [-2:] # characters from the second-last (included) to the end
'we'
```

One way to memorize how slices work is to think that clues point between characters, with the left side of the first character having position 0. The right side of the last character of a string of n characters then has index n , for example :

```
 + --- + --- + --- + --- + --- + --- +
 | P | y | t | h | o | n |
 + --- + --- + --- + --- + --- + --- +
 0 1 2 3 4 5 6
-6 -5 -4 -3 -2 -1
```

The first line of numbers gives the position of the indices 0 ... 6 in the chain; the second line gives the corresponding negative index. The slice from i to j consists of all the characters between the edges labeled i and j, respectively.

For non-negative indices, the length of a slice is the difference between these indices, if the two are between the terminals. For example, the word long [1: 3] is 2.

Using too large an index will generate an error:

```
>>>
>>> word [42] # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out-of-bounds are handled silently when used in slices:

```
>>>
>>> word [4:42]
'we'
>>> word [42:]
''
```

Strings in Python can not be modified, we say that they are immutable. Assigning a new value to an index in a string produces an error:

```
>>>
>>> word [0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word [2:] = 'py'
  ...
```

TypeError: 'str' object does not support item assignment
If you need a different channel, you need to create another one:

```
>>>
>>> 'J' + word [1:]
'Jython'
>>> word [: 2] + 'py'
'Pypy'
```

The native function len () returns the length of a string:

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

See as well
Type Text Sequence - str
Character strings are examples of sequence types, and thus support the typical operations supported by these types.
String methods
Strings support a wide range of basic transformation and search methods.
Syntax for string formatting
Information about formatting strings with the str.format () method.
Formatting chains at the printf
The old formatting operations are described in the following pages.

3.1.3. The lists

Python knows different types of data combined, used to group multiple values. The most flexible is the list, which can be written as a sequence of comma separated values (elements) enclosed in square brackets. The elements of a list are not necessarily all of the same type, although in use it is often the case.

```
>>>
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and any other type of sequence), lists can be indexed and cut:

```
>>>
>>> squares [0] # indexing returns the item
1
>>> squares [-1]
25
>>> squares [-3:] # slicing returns a new list
[9, 16, 25]
```

All sliced operations return a new list containing the requested items. Which means that the following operation returns a shallow copy of the list:

```
>>>
>>> squares [:]
[1, 4, 9, 16, 25]
```
Lists also handle operations such as concatenations:

```
>>>
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```
But unlike lists that are immutable, the lists are mutable: it is possible to change their content:

```
>>>
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes [3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```
It is also possible to add new elements at the end of a list with the append () method. (The methods will be discussed later)

```
>>>
>>> cubes.append (216) # add the cube of 6
>>> cubes.append (7 ** 3) # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```
Slice assignments are also possible, which can even change the size of the list or completely empty it:

```
>>>
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters [2: 5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters [2: 5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear list by replacing the elements with an empty list
>>> letters [:] = []
>>> letters
[]
```
The len () primitive also applies to lists:

```
>>>
>>> letters = ['a', 'b', 'c', 'd']
```

```
>>> len (letters)
4
```
It is possible to nest lists (to create lists containing other lists), for example:

```
          >>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x [0]
['a', 'b', 'c']
>>> x [0] [1]
'B'
```

3.2. First steps towards programming

Of course, you can use Python for more complicated tasks than adding two and two. For example, we can write an initial subsequence of the Fibonacci sequence like this:

```
>>>
>>> # Fibonacci series:
... # the sum of two elements
... 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 are simultaneously assigned their new values 0 and 1. This method is still used in the last line, to show that the expressions on the right side of the assignment are all evaluated. before the assignments are made. These right-hand expressions are always evaluated from left to right.

The while loop executes as long as the condition (here: b <10) remains true. In Python, as in C, any integer other than zero is true and zero is false. The condition can also be a string, a list, or in fact any sequence; a sequence with a non-zero value is true, an empty sequence is false. The test used in the example is a simple comparison. The standard comparison operators are written as in C: <(lower),> (higher), == (equal), <= (lower or equal),> = (greater or equal) and! = (Not equal).

The body of the loop is indented: indentation is the method used by Python to group instructions. In interactive mode, you must enter a tab or spaces for each indented line. In practice, you will want to use a text editor for more complicated entries; all text editors worthy of the name have an

auto-indent function. When a compound expression is entered in interactive mode, it must be followed by a blank line to indicate that it is complete (because the parser can not guess that you just typed the last line). Note that all lines within a block must be indented at the same level.

The print () function writes the values of the parameters supplied to it. It's not the same thing as writing the expression you want to display (as we did in the calculator example), because of how to print to handle multiple parameters, decimal numbers, and chains. Strings are displayed without apostrophes and a space is inserted between the elements so that you can easily format things, like this:

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

The parameter named end can be used to remove the newline, or end the line with another string:

```
>>>
>>> a, b = 0, 1
>>> while b <1000:
... print (b, end = ',')
... a, b = b, a + b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Notes

[1] Since ** has priority over -, -3 ** 2 will be interpreted - (3 ** 2) and will be worth -9. To avoid this and get 9, use parentheses: (-3) ** 2.
[2] Unlike other languages, special characters like \ n have the same meaning in quotation marks ("...") or single quotation marks ("..."). The only difference is that in a quoted string there is no need to protect single quotation marks and vice versa.

4. Other flow control tools

In addition to the while statement that has just been introduced, Python has the usual flow control instructions found in other languages, but always with its own twist.

4.1. The if statement

The if statement is probably the best known. For example :

```
>>>
>>> x = int (input ("Please enter an integer:"))
Please enter an integer: 42
>>> if x <0:
```

```
... x = 0
... print ('Negative changed to zero')
... elif x == 0:
... print ('Zero')
... elif x == 1:
... print ('Single')
... else:
... print ('More')
...
more
```

There can be any number of elif parts, and the else part is optional. The key word "elif" is a shortcut for "else if", but allows to gain a level of indentation. An if ... elif ... elif ... sequence is also equivalent to the switch or case statements available in other languages.

4.2. The instruction for

The instruction for that Python offers is a little different from that found in C or Pascal. Instead of always iterating over an arithmetic sequence of numbers (as in Pascal), or giving the user the ability to define the iteration step and the end condition (as in C), the for statement in Python iterates over the elements of a sequence (which can be a list, a string of characters ...), in the order in which they appear in the sequence. For example (no pun):

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

If you have to modify the sequence on which the iteration takes place inside the loop (for example to duplicate or delete an element), it is more than recommended to start by making a copy, this one does not being not implied. The notation "slices" makes this operation particularly simple:

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

4.3. The range () function

If you have to iterate over a series of numbers, the built-in function range () is made for that. It generates arithmetic sequences:

```
>>>
>>> for i in range (5):
... print (i)
```

...
0
1
2
3
4
The last element provided as parameter is never part of the generated list; range (10) generates a list of 10 values, whose values range from 0 to 9. It is possible to specify a start value and / or a different increment value (s) (including negative for the latter, which sometimes called "not")

range (5, 10)
5 through 9

range (0, 10, 3)
0, 3, 6, 9

range (-10, -100, -30)
-10, -40, -70
To iterate on the indices of a sequence, we can combine the range () and len () functions.

```
>>>
>>> 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
```
However, in most cases, it is more convenient to use the enumerate () function. See for this Loop Techniques.

A strange thing happens when displaying a range:

```
>>>
>>> print (range (10))
range (0, 10)
```
Objects given by range () behave almost like lists, but they are not. These are objects that generate the elements of the sequence as they are iterated, thus saving space.

Such objects are called iterables, that is objects that are suitable for iterators, functions or constructions that expect something from which they can draw elements, successively successive, until exhaustion. We have seen that the for statement is an iterator. The list () function is another one, which creates lists from iterables:

```
>>>
>>> list (range (5))
[0, 1, 2, 3, 4]
```

Further we will see other functions that give iterables or take as parameter.

4.4. The break and continue statements, and the else clauses within the loops
The break statement, like in C, breaks out of the innermost enclosing for or while loop.

Loops can also have an else statement; it is executed when a loop ends when all its elements have been processed (in the case of a for) or the condition becomes false (in the case of a while), but not when the loop is interrupted by a break statement. The following example, which does a search for prime numbers, is a proof of this:

```
>>>
>>> for n in range (2, 10):
... for x in range (2, n):
... if n% x == 0:
... print (n, 'equals', x, '*', n // x)
... break
... else:
... # loop fell through without finding a factor
... print (n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this code is correct.) Look closely: the else statement is attached to the for loop, not the if statement.)

When used in a loop, the else clause is therefore closer to the one associated with a try statement than the one associated with an if statement: the else clause of a try statement executes when no exception is thrown , and that of a loop when no break intervenes. For more information on the try statement and exception handling, see Managing Exceptions.

The continuation instruction, also borrowed from C, moves the loop to its next iteration:

```
>>>
>>> for num in range (2, 10):
... if num% 2 == 0:
... print ("Found an even number", num)
... keep on going
... print ("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
```

Found a number 7
Found an even number 8
Found a number 9
4.5. The pass instruction
The pass instruction does nothing. It can be used when an instruction is needed to provide correct syntax, but no action should be performed. For example :

>>>
>>> while True:
... pass # Busy-wait for keyboard switch (Ctrl + C)
...
This statement is commonly used to create minimal classes:

>>>
>>> class MyEmptyClass:
... pass
...
Another use of the pass is to reserve a space in the development phase for a function or conditional processing, allowing you to build your code at a more abstract level. The pass statement is silently ignored:

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

4.6. Define functions

You can create a function that writes the Fibonacci sequence to an imposed limit:

>>>
>>> def fib (n): # write Fibonacci series up to n
... "" "Print a Fibonacci series up to n." ""
... a, b = 0, 1
... while a <n:
... print (a, end = '')
... a, b = b, a + b
... print ()
...
>>> # Now call the function we just defined:
... fib (2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
The def keyword introduces a function definition. It must be followed by the name of the function and a list in parentheses of its parameters. The instruction that constitutes the body of the function starts at the next line, and must be indented.

The first statement of a function can optionally be a literal character string; this string will then be the documentation string for the function, or docstring (see the Documentation String section for more information). There are tools that use these documentation strings to automatically generate

online or printed documentation, or to allow the user to interactively navigate the code; make it a habit, it's a good habit to document the code you write!

The execution of a function introduces a new symbol table used by the local variables of the function. Specifically, all variable assignments performed within a function store the value in the local symbol table; while variable references are searched for in the local symbol table, then in the local symbol table of the enclosing functions, then in the global symbol table and finally in the primitive name table. As a result, you can not assign a value to a global variable (except by using a global statement), although they can be referenced.

The actual parameters (arguments) of a function are introduced into the local symbol table of the called function when it is called; therefore, the parameter passes are by value, the value always being a reference to an object, not the value of the object itself. [1] When a function calls another function, a new local symbol table is created for that call.

A function definition introduces the name of the function into the current symbol table. The value of the function name is a type that is recognized by the interpreter as a user function. This value can be assigned to another name which can then be used as a function. This provides a general renaming mechanism:

```
>>>
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f (100)
0 1 1 2 3 5 8 13 21 34 55 89
```

If you come from other languages, you may think that fib is not a function but a procedure, since it does not return a result. In fact, even functions without a return statement return a value, albeit boring. This value is called None (this is the name of a primitive). Writing the None value is normally suppressed by the interpreter when it is the only value written. You can see it if you really care about it using print ()

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

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

```
>>> def fib2 (n): # return Fibonacci series up to n
... "" "Return to list containing the Fibonacci series up to n." ""
... result = []
... a, b = 0, 1
... while a <n:
... result.append (a) # see below
... a, b = b, a + b
... return result
...
>>> f100 = fib2 (100) # call it
```

```
>>> f100 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```
This example, as usual, illustrates new features of Python:

The return statement causes the output of the function to return a value. return without expression as parameter returns None. Reaching the end of a function also returns None.
The result.append (a) statement calls a method of the result object that is a list. A method is a function that "belongs to" an object and is named obj.methodname, where obj is an object (it can also be an expression), and methodname is the name of a method defined by the type of the object. Different types define different methods. Different types of methods can have the same name without ambiguity (you can define your own object types and methods using classes, see Classes). The append () method given in this example is defined for lists; they add a new item at the end of the list. In this example, it is the equivalent of result = result + [a], but it is more efficient.

4.7. More about defining functions

It is also possible to define functions with a variable number of arguments. Three syntaxes can be used, possibly combined.

4.7.1. Default value of arguments

The most useful form is to specify a default value for some arguments. This creates a function that can be called with fewer arguments than those present in its definition. For example :

```
def ask_ok (prompt, retries = 4, reminder = 'Please try again!'):
   while True:
      ok = input (prompt)
      if ok in ('y', 'ye', 'yes'):
         return True
      if ok in ('n', 'no', 'nop', 'nope'):
         return False
      retries = retries - 1
      if retries <0:
         raise ValueError ('invalid user response')
      print (reminder)
```
This function can be called in several ways:

providing only the required arguments: ask_ok ('Do you really want to leave?')
by providing some of the optional arguments: ask_ok ('OK to overwrite the file?', 2)
providing all the arguments: ask_ok ('OK to overwrite the file?', 2, 'Come on, only yes or no!')
This example also has the keyword in. This one allows to test if a sequence contains a certain value.

The default values are evaluated when defining the function in the definition scope, so that:

```
i = 5

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

```
i = 6
f ()
```
will print 5.

Important Warning: The default value is evaluated only once. This makes a difference when this default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments that are passed to it over successive calls:

```
def f (a, L = []):
    L.append (a)
    return L

print (f (1))
print (f (2))
print (f) (3)
```
This will print:

```
[1]
[1, 2]
[1, 2, 3]
```
If you do not want this default value to be shared between successive calls, you can write the function this way:

```
def f (a, L = None):
    if L is None:
        L = []
    L.append (a)
    return L
```

4.7.2. Named arguments

Functions can also be called using arguments named as kwarg = value. For example, the following function:

```
        def parrot (voltage, state = 'a stiff', action = 'voom', type = 'Norwegian Blue'):
    print ("- This parrot would not," action, end = '')
    print ("if you put", voltage, "volts through it.")
    print ("- Lovely plumage, the", type)
    print ("- It's", state, "!")
```
accepts a mandatory argument (voltage) and three optional arguments (state, action and type). This function can be called in any of the following ways:

```
parrot (1000) # 1 positional argument
parrot (voltage = 1000) # 1 keyword argument
parrot (voltage = 1000000, action = 'VOOOOOM') # 2 keyword arguments
parrot (action = 'VOOOOOM', voltage = 1000000) # 2 keyword arguments
parrot ('a million', 'bereft of life', 'jump') # 3 positional arguments
```

parrot ('a thousand', state = 'pushing up the daisies') # 1 positional, 1 keyword
but all subsequent calls are incorrect:

parrot () # required argument missing
parrot (voltage = 5.0, 'dead') # non-keyword argument
parrot (110, voltage = 220) # duplicate value for the same argument
parrot (actor = 'John Cleese') # unknown keyword argument
In a function call, named arguments must follow the positioned arguments. All named arguments must match one of the arguments accepted by the function (for example, actor is not an argument accepted by the parrot function), but their order is not important. This also includes the optional arguments (parrot (voltage = 1000) is also correct). No argument can receive a value more than once, as this incorrect example illustrates because of this restriction:

>>>
>>> def function (a):
... pass
...
>>> function (0, a = 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function () got multiple values for keyword argument 'a'
When a last formal parameter is present in the form ** name, it receives a dictionary (see Mapping Types - dict) containing all named arguments except those corresponding to a formal parameter. This can be combined with a formal parameter in the form * name (described in the next section) which receives a tuple containing the arguments positioned beyond the list of formal parameters (* name must be present before ** name). For example, if you define a function like this:

```
def cheeseshop (kind, * arguments, ** keywords):
   print ("- Do you have any", kind, "?")
   print ("- I'm sorry, we're all out of", kind)
   for arg in arguments:
      print (arg)
   print ("-" * 40)
   keys = sorted (keywords.keys ())
   for kw in keys:
      print (kw, ":", keywords [kw])
```

It could be called like this:

```
cheeseshop ("Limburger", "It's very runny, sir.",
         "It's really very, VERY runny, sir."
         shopkeeper = "Michael Palin",
         customer = "John Cleese",
         sketch = "Cheese Shop Sketch")
```

and of course, it would show:

```
- Do you have any Limburger?
- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
```

```
----------------------------------------
Customer: John Cleese
shopkeeper: Michael Palin
sketch: Cheese Shop Sketch
```
Note that the list of named arguments is created by sorting the dictionary keys extracted by the keys () method before printing them. If this is not done, the order in which the arguments are printed is not defined.

4.7.3. Lists of arbitrary arguments

Finally, the least common option is to indicate that a function can be called with an arbitrary number of arguments. These arguments are embedded in a tuple (see Tuples and Sequences). Before the variable number of arguments, zero or more normal arguments may appear:

```
def write_multiple_items (file, separator, * args):
    file.write (separator.join (args))
```
Normally, it's variadic arguments are the last parameters, because they aggregate all the following values. Any parameter placed after the * arg parameter can only be used by name.

```
>>>
>>> def concat (* args, sep = "/"):
... return sep.join (args)
...
>>> concat ("earth", "march", "venus")
'Earth / mars / venus'
>>> concat ("earth", "march", "venus", sep = ".")
'Earth.mars.venus'
```

4.7.4. Separation of the argument lists

The opposite situation occurs when the arguments are already in a list or tuple but must be separated for a function call that requires separate positioned arguments. For example, the range () primitive expects separate start and stop arguments. If they are not available separately, write the function call using the * operator to separate the arguments in a list or tuple:

```
        >>> list (range (3, 6)) # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list (range (* args)) # call with arguments unpacked from a list
[3, 4, 5]
```
Similarly, dictionaries can provide named arguments using the operator **

```
>>>
>>> def parrot (voltage, state = 'a stiff', action = 'voom'):
... print ("- This parrot would not", action, end = ")
... print ("if you put", voltage, "volts through it.", end = ")
... print ("E's", state, "!")
...
```

```
>>> d = \'7b"voltage": "four million", "state": "bleedin 'demised", "action": "VOOM"\'7d
>>> parrot (** d)
- This parrot would not VOOM if you put four million volts through it. E's bleedin 'demised!
```

4.7.5. Anonymous functions

With the keyword lambda, one can create small anonymous functions. This is a function that returns the sum of its two arguments: lambda a, b: a + b. Lambda functions can be used anywhere a function object is expected. They are syntactically restricted to a single expression. Semantically, they are only syntactic sugar for a normal function definition. Like nested functions, lambda functions can reference variables in the enclosing scope:

```
>>>
>>> def make_incrementor (n):
... return lambda x: x + n
...
>>> f = make_incrementor (42)
>>> f (0)
42
>>> f (1)
43
```

The previous example uses an anonymous function to return a function. Another typical use is to give a minimalist function directly as a parameter:

```
>>>
>>> peers = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> peers.sort (key = lambda even: pair [1])
>>> peers
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.6. Documentation chains

Here are some conventions regarding the content and format of the documentation strings.

The first line should always be short, a concise summary of the utility of the object. To be brief, there is no need to recall the name of the object or its type, which is accessible by other means (unless the name is a verb that describes an operation). This line should start with a capital letter and end with a dot.

If there are other lines in the documentation chain, the second line should be empty, to separate it visually from the rest of the description. The other lines can then be one or more paragraphs describing how the object is used, its edge effects, and so on.

The Python code parser does not remove indentation from multi-line literal strings, so tools that use the documentation need to do this themselves. The following convention applies: the first nonempty line after the first one determines the depth of indentation of the entire documentation string (you can not use the first line, which is usually contiguous to the opening quotation marks of the string and whose indentation is not visible). The spaces "corresponding" to this depth of indentation are then removed from the beginning of each line of the chain. No line should have a

lower level of indentation, but if this happens, all spaces at the beginning of the line should be removed. The equivalent of the spaces must be tested after expansion of the tabs (normally replaced by 4 spaces).

Here is an example of a multi-line documentation string:

```
        >>> def my_function ():
... "" "Do nothing, but document it.
...
... No, really, it does not do anything.
... "" "
... pass
...
>>> print (my_function .__ doc__)
Do nothing, but document it.

   No, really, it does not do anything.
```

4.7.7. Annotations of functions

Function annotations are optional metadata describing the types used by a user-defined function (see PEP 484 for more information).

Annotations are stored in the __annotations__ attribute of the function, as a dictionary, and have no other effect. The annotations on the parameters are defined by two periods (:) after the name of the parameter followed by an expression giving the value of the annotation. The return annotations are defined by -> followed by an expression, between the list of parameters and the two end points of the def statement. The following example has a positional parameter, a named parameter, and an annotated return value:

```
>>>
>>> def f (ham: str, eggs: str = 'eggs') -> str:
... print ("Annotations:", f .__ annotations__)
... print ("Arguments:", ham, eggs)
... return ham + 'and' + eggs
...
>>> f ('spam')
Annotations: \'7b'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>\'7d
Arguments: spam eggs
'spam and eggs'
```

4.8. A coding style: Intermezzo

Now that you're ready to write longer and more complex programs, it's time to talk about the coding style. Most languages can be written (or rather formatted) according to different styles; some are more legible than others. Making reading your code easier for others is always a good idea, and adopting a good coding style can help you a lot.

In Python, PEP 8 has emerged as a guide to which most projects adhere; it puts forward a

very legible coding style and pleasing to the eye. Every Python developer should read it and get as much inspiration as possible; Here are his main points:

Use indentations of 4 spaces, and no tabulation.

4 spaces are a good compromise between a short indentation (which allows for a greater nesting depth) and a longer one (which makes the code easier to read). Tabs introduce confusion, and should be proscribed as much as possible.

Make line breaks, so that they do not exceed 79 characters.

This helps users with only small screens, and allows larger ones to have multiple files side by side without difficulty.

Use empty lines to separate functions and classes, or to split large blocks of code into functions.

Where possible, place comments on their own lines.

Use the documentation strings

Use spaces around the operators and after the commas, but not directly inside the parentheses: a = f (1, 2) + g (3, 4).

Always name your classes and functions in the same way; the convention is to use a CamelCase notation for classes, and lowercase_with_train_string for functions and methods. Always use self as the name of the first argument of the methods (see A first approach of classes to learn more about classes and methods).

Do not use exotic encodings when your code is meant to be used in international environments. By default, Python works in UTF-8, or otherwise simple ASCII works in most cases.

In the same way, use only ASCII characters for your variable names if you suspect that someone speaking another language will read or have to modify your code.

Notes

[1] In fact, calls by object reference would probably be a more accurate description, since if a mutable object is passed as an argument, the caller will see all the changes that have been made by the called party (inserting items into a list ...).

5. Data structures

This chapter takes some of the points already described in more detail, and introduces new concepts.

5.1. Complements on the lists

The list type has additional methods. Here is the complete list of methods for list objects:

list.append (x)
Add an item to the end of the list. Equivalent to a [len (a):] = [x].

list.extend (iterable)
Extend the list by appending all items from the iterable. Equivalent to a [len (a):] = iterable.

list.insert (i, x)
Inserts an item at the specified position. The first argument is the position of the current element before the insertion must occur, so a.insert (0, x) inserts the element at the top of the list, and a.insert (len (a), x) is equivalent to a.append (x).

list.remove (x)
Removes from the list the first item whose value is x. An exception is thrown if there is no element with this value.

list.pop ([i])
Removes the item at the specified position from the list and returns it. If no position is specified, a.pop () removes and returns the last item in the list (the square brackets around the i in the method signature indicate that this parameter is optional, not that you need to place brackets in your code! You will find this notation frequently in the Python Library Reference Guide).

list.clear ()
Deletes all items in the list, equivalent to del a [:].

list.index (x [, start [, end]])
Return zero-based index in the list of the first item whose value is x. Raises a ValueError if there is no such item.

The optional arguments start and end are in the slice notation and are used in a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument.

list.count (x)
Returns the number of items with the value x in the list.

list.sort (key = None, reverse = False)
Sort the elements in place, (the arguments can customize sorting, see sorted () for their explanation).

list.reverse ()
Reverse the order of the items in the list, on the spot.

list.copy ()
Returns a shallow copy of the list. Equivalent to a [:].

The following example uses most of the list methods:

```
        >>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count ('apple')
2
>>> fruits.count ('tangerine')
0
>>> fruits.index ('banana')
3
>>> fruits.index ('banana', 4) # Find next banana starting at position 4
6
>>> fruits.reverse ()
>>> fruits
[banana, apple, kiwi, banana, pear, apple, orange]
>>> fruits.append ('grape')
>>> fruits
banana, apple, kiwi, banana, pear, apple, orange, grape
>>> fruits.sort ()
>>> fruits
apple, banana, banana, grape, kiwi, orange, pear
>>> fruits.pop ()
'Pear'
```

You have probably noticed that methods such as insert, remove, or sort, which only modify the list, do not return a value, but None. [1] This is a principle respected by all variable data structures in Python.

5.1.1. Use lists like batteries

The list methods make it very easy to use them as stacks, where the last element added is the first one retrieved ("last in, first out", or LIFO for "last-in, first-out"). To add an element to the stack, use the append () method. To retrieve the object at the top of the stack, use the pop () method, without a position indicator. For example :

```
>>>
>>> stack = [3, 4, 5]
>>> stack.append (6)
>>> stack.append (7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop ()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop ()
6
>>> stack.pop ()
5
>>> stack
[3, 4]
```

5.1.2. Use lists as queues

It is also possible to use a list as a queue, where the first element added is the first one retrieved ("first in, first out", or FIFO for "first-in, first-out"); however, lists are not very effective for this type of treatment. While the additions and deletions at the end of the list are fast, insertions or withdrawals at the beginning of the list are slow (because all the other elements must be shifted by one position).

To implement a queue, use the collections.deque class that was designed to provide fast add and drop operations at both ends. For example :

```
>>>
>>> from collections import deque
>>> tail = deque (["Eric", "John", "Michael"])
>>> tail.append ("Terry") # Terry arrives
>>> tail.append ("Graham") # Graham arrives
>>> tail.popleft () # The first to arrive now
'Eric'
>>> tail.popleft () # The second to arrive now
'John'
>>> queue # Remaining queue in order of arrival
deque (['Michael', 'Terry', 'Graham'])
```

5.1.3. List comprehensions

List comprehensions provide a way to build lists in a very concise manner. A typical application is the construction of new lists where each element is the result of an operation applied to each element of another sequence, or to create a subsequence of elements satisfying a specific condition.

For example, suppose we want to create a list of squares, like:

```
>>>
>>> squares = []
>>> for x in range (10):
... squares.append (x ** 2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named x that still exists after running the loop. We can calculate a list of squares without edge effects, with:

```
squares = list (map (lambda x: x ** 2, range (10)))
```

or :

```
squares = [x ** 2 for x in range (10)]
```

which is shorter and readable.

A list understanding consists of square brackets containing an expression followed by a for clause, and then zero or more clauses for or if. The result will be a new result list of the evaluation of the expression in the context of the for and if clauses that follow it. For example, this list comprehension combines the elements of two lists if they are not equal:

```
>>>
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x! = y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```
>>>
>>> combs = []
>>> for x in [1,2,3]:
... for y in [3,1,4]:
... if x! = y:
... combs.append ((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note that the order of the for and if statements is the same in these different code snippets.

If the expression is a tuple (ie (x, y) in this example), it must be surrounded by parentheses:

```
>>>
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x * 2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in x if x> = 0]
[0, 2, 4]
>>> # a function to all the elements
>>> [abs (x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['banana', 'loganberry', 'passion fruit']
>>> [weapon.strip () for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x ** 2) for x in range (6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x ** 2 for x in range (6)]
  File "<stdin>", line 1, in <module>
    [x, x ** 2 for x in range (6)]
            ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
```

```
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```
List comprehensions can contain complex expressions and nested functions:

```
>>>
>>> from math import pi
>>> [str (round (pi, i)) for i in range (1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```
5.1.4. Nested list comprehensions

The first expression in a list comprehension can be any expression, including another list comprehension.

See the following example of a 3x4 matrix, implemented as 3 lists of 4 elements:

```
>>>
>>> matrix = [
... [1, 2, 3, 4],
... [5, 6, 7, 8],
... [9, 10, 11, 12],
...]
```
This list comprehension will transpose the rows and columns:

```
>>>
>>> [[row [i] for row in matrix] for i in range (4)
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```
As we saw in the previous section, the nested list comprehension is evaluated in the context of the for statement that follows it, so this example is equivalent to:

```
>>>
>>> transposed = []
>>> for i in range (4):
... transposed.append ([row [i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```
which in turn is equivalent to:

```
>>>
>>> transposed = []
>>> for i in range (4):
... # the following 3 lines implement the nested listcomp
... transposed_row = []
... for row in matrix:
... transposed_row.append (row [i])
... transposed.append (transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```
In concrete cases, it is always better to use native functions rather than complex flow control

instructions. The zip () function would do an excellent job in this case:

```
>>>
>>> list (zip (* matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

5.2. The instruction del

There is a way to remove an item from a list from its position instead of its value: the del statement. It differs from the pop () method, which returns a value. The del statement can also be used to delete slices from a list or empty it completely (which we did before by assigning an empty list to the slice). For example :

```
>>>
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a [0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a [2: 4]
>>> a
[1, 66.25, 1234.5]
>>> del a [:]
>>> a
[]
```

del can also be used to delete variables:

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

From there, referencing the name a is an error (at least until another value is assigned to it). You will find other uses of the del function later.

5.3. Tuples and sequences

We have seen that lists and character strings have many properties in common, such as indexing and slice operations.

As Python is a constantly evolving language, other types of sequences may be added. There is also another standard type of sequence: tuple.

A tuple consists of different values separated by commas, such as:

```
>>>
>>> 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))
>>> # Tuples are immutable:
... t [0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```
As you can see, on display, tuples are always surrounded by parentheses, so nested tuples are interpreted correctly; they can be entered with or without parentheses, even if they are often necessary (especially when a tuple is part of a longer expression). It is not possible to assign a value to an element of a tuple; on the other hand, it is possible to create tuples containing mutable objects, like lists.

While tuples may look similar to lists, they are often used in different cases and for different reasons. The tuples are immutable and often contain heterogeneous sequences of elements that are accessed by "unpacking" (see below) or indexing (or even by attributes in the case of namedtuples). Lists are often mutable, and contain homogeneous elements that are iteratively accessed on the list.

A specific problem is the construction of tuples containing none or only one element: the syntax has some specific turns to accommodate it. Empty tuples are built by a pair of empty parentheses; a tuple with a single element is constructed by following the value with a comma (it is not sufficient to put this value in parentheses). Not very pretty, but effective. For example :

```
>>>
>>> empty = ()
>>> singleton = 'hello', # <- note trailing comma
>>> len (empty)
0
>>> len (singleton)
1
>>> singleton
('Hello',)
```
The instruction t = 12345, 54321, 'hello!' is an example of a tuple package: the values 12345, 54321 and hello! are packed together in a tuple. The reverse operation is also possible:

```
>>>
>>> x, y, z = t
```
This is called, more or less appropriately, a sequence unpacking and works for any sequence placed to the right of the expression. This unpacking requires as many variables on the left as there are elements in the sequence. Also note that this multiple assignment is just a combination of a tuple package and a sequence unpack.

5.4. Sets

Python also provides a data type for sets. A set is an unordered collection with no duplicate elements. Basic uses concern, for example, membership tests or deletions of duplicates. Sets also support mathematical operations such as symmetric unions, intersections, differences, and differences.

Braces, or the set () function can be used to create sets. Note that to create an empty set, \'7b\'7d does not work, it creates an empty dictionary. Use set () instead.

Here is a brief demonstration:

```
>>>
>>> basket = \'7b'apple', 'orange', 'apple', 'pear', 'orange', 'banana'\'7d
>>> print (basket) # show that duplicates have been removed
\'7b'orange', 'banana', 'pear', 'apple'\'7d
>>> 'orange' in basket # fast membership testing
true
>>> 'crabgrass' in basket
false

>>> # Demonstrate set operations on a single letters from two words
...
>>> a = set ('abracadabra')
>>> b = set ('alacazam')
>>> a # unique letters in a
\'7b'a', 'r', 'b', 'c', 'd'\'7d
>>> a - b # letters in a word not in b
\'7b'r', 'd', 'b'\'7d
>>> a | b # letters in either a or b
\'7b'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'\'7d
>>> a & b # letters in both a and b
\'7b'a', 'c'\'7d
>>> a ^ b # letters in a or b but not both
\'7b'r', 'd', 'b', 'm', 'z', 'l'\'7d
```

Just like list comprehensions, it is possible to write sets of understandings:

```
>>>
>>> a = \'7bx for x in 'abracadabra' if x not in 'abc'\'7d
>>> a
\'7b'r', 'd'\'7d
```

5.5. Dictionaries

Another very useful type of data, native in Python, is the dictionary. These dictionaries are sometimes present in other languages under the name of "associative memories" or "associative arrays". Unlike sequences, which are indexed by numbers, dictionaries are indexed by keys, which can be of any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains a mutable object, directly or

indirectly, it can not be used as a key. You can not use lists as keys, because lists can be edited in place using position, slice or method assignments such as append () or extend ().

The simplest is to consider dictionaries as unordered sets of key pairs: value, the keys to be unique (within a dictionary). A pair of braces creates an empty dictionary: \'7b\'7d. Placing a comma-separated list of key-value pairs inside braces adds the corresponding values to the dictionary; this is also the way the dictionaries are displayed as output.

The main operations performed on a dictionary consist in storing a value for a key and extracting the value corresponding to a key. It is also possible to delete a key pair: value with del. If you store a value for a key that is already in use, the old value associated with that key is lost. If you try to retrieve a value associated with a key that does not exist, an exception is thrown.

Execute list (d.keys ()) on a dictionary d returns a list of all the keys used in the dictionary, in an arbitrary order (if you want them sorted, use sorted (d.keys ())). [2] To test if a key is in the dictionary, use the keyword in.

Here is a small example using a dictionary:

```
        >>> tel = \'7b'jack': 4098, 'sape': 4139\'7d
>>> such ['guido'] = 4127
>>> such
\'7b'sape': 4139, 'guido': 4127, 'jack': 4098\'7d
>>> such ['jack']
4098
>>> del tel ['sape']
>>> such ['irv'] = 4127
>>> such
\'7b'guido': 4127, 'irv': 4127, 'jack': 4098\'7d
>>> list (tel.keys ())
['irv', 'guido', 'jack']
>>> sorted (tel.keys ())
['guido', 'irv', 'jack']
>>> 'guido' in such
true
>>> 'jack' not in such
false
```

The dict () constructor builds a dictionary directly from a list of key-value pairs stored as tuples:

```
>>>
>>> dict ([('sape', 4139), ('guido', 4127), ('jack', 4098)])
\'7b'sape': 4139, 'jack': 4098, 'guido': 4127\'7d
```

In addition, it is possible to create dictionaries by understanding from a set of key and values:

```
>>>
>>> \'7bx: x ** 2 for x in (2, 4, 6)\'7d
\'7b2: 4, 4: 16, 6: 36\'7d
```

When keys are simple strings, it is sometimes easier to specify pairs using named parameters:

```
>>>
```

```
>>> dict (sape = 4139, guido = 4127, jack = 4098)
\'7b'sape': 4139, 'jack': 4098, 'guido': 4127\'7d
```

5.6. Loop techniques

When looping a dictionary, keys and their values can be retrieved at the same time using the items () method

```
>>>
>>> knights = \'7b'gallahad': 'the pure', 'robin': 'the brave'\'7d
>>> for k, v in knights.items ():
... print (k, v)
...
gallahad the pure
robin the brave
```

When you iterate over a sequence, the position and the corresponding value can be retrieved at the same time using the enumerate () function.

```
>>>
>>> for i, v in enumerate (['tic', 'tac', 'toe']):
... print (i, v)
...
0 tic
1 tac
2 toe
```

To make loops on two or more sequences at the same time, the elements can be associated by the zip () function

```
>>>
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, in zip (questions, answers):
... print ('What is your \'7b0\'7d? it is \'7b1\'7d.'. format (q, a))
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

To loop through an inverted sequence, start by creating the sequence in its normal order, then apply the reversed () function

```
>>>
>>> for i in reversed (range (1, 10, 2)):
... print (i)
...
9
7
5
3
1
```

To loop over a sorted sequence, use the sorted () function, which returns a new sorted list without altering the source:

```
>>>
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted (basketball set):
... print (e)
...
apple
banana
orange
pear
```
It is sometimes tempting to change a list while iterating, however, it is often simpler and safer to create a new list instead.

```
>>>
>>> import math
>>> raw_data = [56.2, float ('NaN'), 51.7, 55.3, 52.5, float ('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
... if not math.isnan (value):
... filtered_data.append (value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```
5.7. More information about the conditions

Conditions used in a while or if statement may contain any operator, not just comparisons.

The in and not in comparison operators test whether a value is present in a sequence or not. The is and is not operators test whether two objects are really the same object; this is only important for mutable objects such as lists. All comparison operators have the same priority, which is lower than that of digital operators.

The comparisons can be chained. For example, a <b == c tests if a is less than or equal to b and moreover if b is equal to c.

Comparisons can be combined using the Boolean operators and and or, the result of a comparison (or any Boolean expression) that can be reversed with not. These operators have a lower priority than comparison operators; between them, not at the highest priority and lowest gold, so that A and not B or C is equivalent to (A and (not B)) or C. As always, parentheses can be used to express the desired instruction.

The Boolean operators and and or are called short-circuit operators: their arguments are evaluated from left to right, and the evaluation stops as soon as the result is determined. For example, if A and C are true and B is false, A and B and C do not evaluate the C expression. When used as a value and not as a boolean, the return value of a short-circuit operator is that of 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, assignments can not occur inside expressions. The C programmers may complain after that, but this avoids frequent errors that occur in C, when the = expression is placed when an == expression was expected.

5.8. Compare sequences with other types

Sequences can be compared with other sequences of the same type. The comparison uses a lexicographic order: the first two elements of each sequence are compared, and if they differ it determines the result of the comparison; if they are equal, the next two elements are compared in turn, and so on until one of the sequences is exhausted. If two elements to be compared are themselves sequences of the same type, then the lexicographic comparison is performed recursively. If all elements of both sequences are equal, then both sequences are considered equal. If one sequence is a subsequence of the other, the shortest sequence is the one whose value is lower. The lexicographic comparison of strings uses the Unicode code of the characters. Here are some examples of comparisons between sequences of the same type:

```
(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)
```

Comparing objects of different types with <or> is allowed if the objects have appropriate comparison methods. For example, numeric types are compared via their numeric value, so 0 is equal to 0.0, and so on. In other cases, instead of giving an unpredictable command, the interpreter will throw a TypeError exception.

Notes

[1] Other languages could return the modified object, which allows to chain the methods, such as: d-> insert ("a") -> remove ("b") -> sort () ;.
[2] Calling d.keys () returns a dictionary view object, which handles the operations of the membership test type (in) and iteration. But its content is not independent of the original dictionary, it's a simple view.

6. Modules

When you quit and re-enter the Python interpreter, everything you said in the previous session is

lost. In order to write longer programs, you should use a text editor, prepare your code in a file, and run Python with this file as a parameter. It's called created a script. When your program becomes longer, you will be able to separate your code into several files, and you will also find it convenient to reuse functions written for one program in another without having to copy them.

To handle this, Python has a way to write definitions in a file and use them in a script or interactive session. Such a file is called a module, and the definitions of a module can be imported into another module or into the main module (which is the module that contain you variables and definitions when running a script or in mode interactive).

A module is a file containing definitions and instructions. Its file name is the same as its name, suffixed by .py. Inside a module, its own name is accessible in the __name__ variable. For example, take your favorite editor and create a fibo.py file containing:

```
# Fibonacci numbers module

def fib (n): # write Fibonacci series up to n
   a, b = 0, 1
   while b <n:
      print (b, end = '')
      a, b = b, a + b
   print ()

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, by being in the same folder, open an interpreter and import the module by typing:

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

It does not matter the names of the functions defined in fibo directly in the current symbol table, but simply adds fibo. You can therefore call the functions via the module name:

```
>>>
>>> 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, it is possible to assign a local name:

```
>>>
>>> fib = fibo.fib
>>> fib (500)
```

1 1 2 3 5 8 13 21 34 55 89 144 233 377

6.1. The modules in detail

A module can contain both instructions and function declarations. These instructions allow you to initialize the module, and are only executed the first time that the name of a module is found in an import. [1] (They are also executed when the file is executed as script.)

Each module has its own symbol table, used as a global symbol table by all functions defined by the module. Thus the author of a module can use global variables in a module without worrying about name collisions with global defined by the user of the module. On the other hand, if you know what you are doing, you can modify a global variable of a module with the same notation as to access the functions: modname.itemname.

Modules can import other modules. It is usual but not mandatory to store all the import at the beginning of the module (or script). Imported module names are inserted into the global symbol table of the module that imports.

There is a variation to the import statement that imports the names of a module directly into the symbol table of the module that imports it, for example:

```
>>>
>>> from fibo import fib, fib2
>>> fib (500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```
This does not insert the module name from which the definitions are retrieved from the local symbol table (in this example, fibo is not defined).

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

```
>>>
>>> from fibo import *
>>> fib (500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```
All names that do not begin with a low hyphen (_) are imported. In the vast majority of cases, developers do not use this syntax because by importing an indefinite set of names, already defined names can be hidden.

Note that in general, import * of a module or a package is deprecated, in general it generates code difficult to read. However, it is acceptable to use it to gain a few seconds in interactive mode.

Note For performance reasons, each module is imported only once per session. If you change the code of a module you must restart the interpreter to see the impact. Or re-import it explicitly using importlib.reload (), for example: import importlib; importlib.reload (modulename).

6.1.1. Run modules like scripts

When you run a Python module with:

```
python fibo.py <arguments>
```

the module code will be executed as if you had imported it, but its __name__ will be "__main__". So by adding these lines at the end of the module:

```
if __name__ == "__main__":
   import sys
   fib (int (sys.argv [1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line is only started if the module is executed as "main" file:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the file is imported, the code is not executed:

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

This is typically used to either provide a user interface for a module, or to run the tests on the module (where to run the module as script runs the tests).

6.1.2. Module search files

When a module named for example spam is imported, it is first searched among the native modules, then if it is not found, the interpreter will look for a file named spam.py in a list of given folders. by the sys.path variable, sys.path is initialized by default to:

The folder containing the current script (or the current folder if no script is given).
PYTHONPATH (a list of folders, using the same syntax as the PATH shell variable).
The default value, dependent on the installation.
Note On systems that handle symbolic links, the folder containing the current script is resolved after following the link sublimate script. In other words, the folder containing the symbolic link is not added to the module search folders.
After initialization, Python programs can modify their sys.path. The folder containing the current script is placed at the beginning of the list of folders to be searched, before the library folders. This means that a module in this folder, having the same name as a module, will be loaded in its place. This is a typical mistake, unless it is wanted. See Standard Modules for more information.

6.1.3. Python files "compiled"

To speed up the loading of modules, Python hides a compiled version of each module, in a file named module (version) .pyc (or version represents the format of the compiled file, typically a version of Python) in the folder __pycache__. For example with CPython 3.3 the compiled version of spam.py would be __pycache __ / spam.cpython-33.pyc. This naming rule allows versions compiled by different Python versions to coexist.

Python compares the modification dates of the source file and its compiled version to see if the module needs to be recompiled. This process is fully automatic, and the compiled versions are independent of the platform, and can be shared between different architecture systems.

There are two circumstances in which Python does not check the cache: The first case when the module is given by the command line (in which case the module is always recompiled, without even hiding its compiled version), the second case is when the module has no source in this case Python does not attempt to load the compiled version. To manage a module without source (only compiled) the compiled module must be in the source folder and its source must not be present.

Tips for experts:

You can use the -O or -OO options when calling Python to reduce the size of the compiled modules. The -O option removes the assert statements, and the -OO option also deletes the __doc__ documentations. However, since some programs need these __doc__, you should only use -OO if you know what you are doing. "Optimized" modules are marked with an opt-in and are usually smaller. Future versions of Python could change the effects of optimization.
A program does not run faster when read from a .pyc, but it is loaded faster since the .pyc is smaller than the .py.
The compileall module can create .pyc files for all modules in a folder.
These are more details about the process, as well as a flow chart of decisions, in PEP 3147.

6.2. Standard modules

Python comes with a library of standard modules, described in the Python Library documentation, below. Some modules are integrated in the interpreter, they expose tools that are not part of the language, but that are still part of the interpreter, either for the practical side, or to expose essential tools such as access calls to systems. The composition of these modules is configurable at compilation, and also depends on the platform targeted. For example, the winreg module is only available on Windows systems. One module deserves special attention, the sys module, which is present in all Python interpreters. The sys.ps1 and sys.ps2 variables define the primary and secondary prompt strings:

```
>>> 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 sys.path variable is a string list that determines the module search paths for the interpreter. It is initialized to a default path taken from the PYTHONPATH environment variable, or an internal default if PYTHONPATH is not defined. sys.path is editable using the usual list operations:

```
>>>
```

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

6.3. The dir () function

The internal function dir () is used to find which names are defined by a module. It gives a sorted list of strings:;

```
>>>
>>> import fibo, sys
>>> dir (fibo)
['__name__', 'fib', 'fib2']
>>> dir (sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 'package__ ','standard','standard ',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'exceptionthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 getrefcount, getsizeof, getswitchinterval, gettotalrefcount,
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'info_version', 'warnoptions']
```

Without parameters, dir () lists the currently defined names:

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

Note that it lists all types of names: variables, functions, modules, and so on.

dir () does not list primitive functions or internal variables. If you want to list them, they are defined in the builtins module

```
>>>
>>> import builtins
>>> dir (builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
```

'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
'__debug__', '__doc__', '__import__', '__name__','package__ ',' abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
hash, help, hex, id, input, int, isinstance, issubclass,
iter, len, license, list, locals, map, max, memoryview
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
'Zip']

6.4. The packets

Packages are a way of structuring the namespaces of Python modules using a "dotted" notation. For example, the module name AB designates the submodule B of the A package. In the same way that the use of the modules avoids the authors of different modules having to worry about the names of the global variables of the others. Using module names with periods avoids authors of packages containing multiple modules such as NumPy or "Python Image Library" from having to worry about the module names of others.

Imagine that you want to build a collection of modules (a "package") to uniformly manage files containing sound and sound data. There are a large number of file formats for storing sound (usually spotted by their extension, eg .wav, .aiff, .au), so you'll want to create and maintain a growing number of modules to handle the conversion between all these formats. There is also a lot of operations you would like to do on sound (mix, add echo, equalize, add a stereo effect articifiel), so, in addition to the conversion modules, you will write an unlimited number modules for performing these operations. Here is a possible structure for your package (expressed as a file system, hierarchically):

sound / Top-level package
    __init__.py Initialize the sound package
    formats / Subpackage for file format conversions
        __init__.py
        wavread.py
        wavwrite.py
        aiffread.py

aiffwrite.py
auread.py
auwrite.py
...
effects / Subpackage for sound effects
__init__.py
echo.py
surround.py
reverse.py
...
filters / Subpackage for filters
__init__.py
equalizer.py
vocoder.py
karaoke.py
...
When importing packages, Python looks in each sys.path folder, looking for the package folder.

__init__.py files are needed for Python to consider folders as containing packages, so it avoids folders with common names like string to hide modules that might have been found later in the search folders. In the simplest of cases, __init__.py can be empty, but it can execute initialization code for its package or configure the __all__ variable (documented later).

Users of a module can import its modules individually, for example:

import sound.effects.echo
Load the sound.effects.echo sub-module. He says to be referenced by his full name.

sound.effects.echo.echofilter (input, output, delay = 0.7, atten = 4)
Another way to import submodules is:

from sound.effects import echo
Will also load the echo sub-module, and make it available in the prefix of the package, so it can be used like this:

echo.echofilter (input, output, delay = 0.7, atten = 4)
Another method would be to import the desired function or variable directly:

from sound.effects.echo import echofilter
The echo submodule is still loaded, but here the echofilter () function is available directly:

echofilter (input, output, delay = 0.7, atten = 4)
Note that when using from package import item, item can be either a submodule, a subpacket, or simply a name declared in the package (a variable, a function, or a class). The import statement searches first if item is defined in the package, if it is not, it tries to load a module, and if it fails, an ImportError exception is thrown.

On the other hand, using the import item.item.subitement.subscript.subitem syntax, each item except the last must be packets. The last item can be a module or a package, but can not be a

function, a class, or a variable defined in the previous element.

6.4.1. Import * from a package

What happens when a user writes from sound.effects import *? Ideally we could hope that it would look for all the sub-modules of the package on the file system, and that they would all be imported. It could be long, and importing some submodules could have unwanted side effects, at least, desired only when the submodule is explicitly imported.

The only solution, for the author of the package, is to provide an explicit index of the contents of the package. The import statement uses the following convention: If the __init__.py file in the package defines a list named __all__, this list will be used as a list of module names to be imported when from package import * is used. It is the role of the package author to keep this list up to date when new versions of the package are released. A package author may also decide not to allow * import of their package. For example, the sound / effects / __ init__.py file might contain the following code:

__all__ = ["echo", "surround", "reverse"]
This would mean that from sound.effects import * imported the three sub-modules of the sound package.

If __all__ is not set, the statement from sound.effects import * will not import all sub-modules of the sound.effects package into the current namespace, but will only make sure that the package sound.effects has been imported (all the code in the __init__.py file has been executed) and then imports any names defined in the package. This includes all defined names (and explicitly loaded sub-modules) by __init__.py. It also includes all submodules in the package that have been explicitly loaded by an import statement. Typically:

import sound.effects.echo
import sound.effects.surround
from sound.effects import *
In this example, the echo and surround modules are imported into the current namespace when from ... import is executed because they are defined in the sound.effects package. (It works when __all__ is set.)

Although some modules are designed to export only names that follow certain patterns when you use import *, it is still considered bad practice in production code.

Remember there is nothing wrong with using from import specific_submodule! This is also the recommended way unless the module that makes the imports need submodules with the same name but from different packages.

6.4.2. Internal references in a package

When packets are organized into subpackets (like the sound package for example), you can use absolute imports to target neighboring packets. For example, if the sound.filters.vocoder module needs the echo module of the sound.effects package, it can use from sound.effects import echo.

It is also possible to write relative imports of the form from module import name. These imports are prefixed with dots to indicate their origin (current or parent package). From the surround module, for example, you could do:

from. import echo
from .. import formats
from ..filters import equalizer
Note that the relative imports trust the name of the current module. Since the name of the main module is always "__main__", the modules used by the main module of an application can only be imported by absolute imports.

6.4.3. Packages in multiple folders

The packages expose an additional attribute, __path__, containing a list, initialized before running the __init__.py file, containing the name of its folder in the file system. This list can be modified, thus altering future searches for modules and subpackets contained in the package.

Although this feature is only rarely useful, it can be used to expand the list of modules found in a package.

Notes

[1] In reality, the declaration of a function is itself an instruction, the execution records the name of the function in the global symbol table of the module.

7. Inputs / outputs

There are many ways to present the outputs of a program; the data can be printed in human readable form, or saved in a file for future use. This chapter will therefore present various possibilities.

7.1. Data formatting

So far, we've come across two ways to write data: expression declarations and the print () function. (A third method is to use the write () file method, the standard output file can be referenced as sys.stdout, see the Standard Library Reference Guide for more information.)

But you'll often want more control over formatting your output than just displaying space-separated values. There are two methods for formatting these outputs, the first is to manage strings by yourself, using slices on your strings and concatenation operations, you can create any imaginable layout. But strings also have methods that make it easy, for example, to align string strings with a certain column width, and that will be presented quickly. The second method is to use the str.format () method.

The string module contains a Template class that offers yet another way to replace values within strings.

But one question remains, of course: how to convert values into strings? Fortunately, Python provides several ways to convert any value into a string: the repr () and str () functions.

The str () function is intended to return human-readable representations of values, whereas the repr () function is intended to generate representations that can be read by the interpreter (or will force a SyntaxError if there is no equivalent syntax). For objects that do not have a specific human representation, str () will return the same value as repr (). Many values, such as numbers or structures such as lists or dictionaries, have the same representation using both functions. Strings and floating point numbers, on the other hand, have two distinct representations.

Some examples :

```
>>>
>>> s = 'Hello, world.'
>>> str (s)
'Bonjour Monde.'
>>> repr (s)
"'Bonjour Monde.'"
>>> str (1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is' + repr (x) + ', and y is' + repr (y) + '...'
>>> print (s)
The value of x is 32.5, and y is 40000 ...
>>> # The repr () of a string adds string quotes and backslashes:
... hello = 'hello, world \ n'
>>> hellos = repr (hello)
>>> print (hellos)
'hello, world \ n'
>>> # The argument to repr () may be any Python object:
... repr ((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
Here are two ways to write a table of squares and cubes:

>>>
>>> for x in range (1, 11):
... print (repr (x) .rjust (2), repr (x * x) .rjust (3), end = '')
# Note use of 'end' on previous line
... print (repr (x * x * x) .rjust (4))
...
 1 1 1
 2 4 8
 3 9 27
 4 16 64
 5 25 125
 6,366,216
 7,449,343
```

```
 8 64 512
 9 81 729
10 100 1000

>>> for x in range (1, 11):
... print ('\'7b0: 2d\'7d \'7b1: 3d\'7d \'7b2: 4d\'7d' format (x, x * x, x * x * x))
...
 1 1 1
 2 4 8
 3 9 27
 4 16 64
 5 25 125
 6,366,216
 7,449,343
 8 64 512
 9 81 729
10 100 1000
```
(Note that in this first example, a space between each column was added by the way print () works, which always adds spaces between its parameters.)

This example demonstrates the use of str.rjust () strings, which makes a right-justified string in a field of a given width by adding spaces to the left. There are similar methods str.ljust () and str.center (). These methods do not write anything, they simply return a new string. If the string passed in parameter is too long, it is not truncated but returned without modification; which can disturb your layout but is often preferable to the alternative, which might lie on a value (and if you really want to truncate your values, you can still use a slice, as in x.ljust (n) [: not]).

There is another method, str.zfill (), which bridges a numeric string on the left with zeros. It includes the plus and minus signs:

```
>>>
>>> '12'.zfill (5)
'00012'
>>> '-3.14'.zfill (7)
'-003.14'
>>> '3.14159265359'.zfill (5)
'3.14159265359'
```
The basic use of the str.format () method looks like this:

```
>>>
>>> print ('We are the \'7b\'7d who say "\'7b\'7d!" ". format (' knights', 'Ni'))
We are the knights who say "Ni!"
```
The braces and the characters they contain (called the formatting fields) are replaced by objects passed as parameters to the str.format () method. A number in braces refers to the position of the object passed to the str.format () method.

```
>>>
>>> print ('\'7b0\'7d and \'7b1\'7d'. format ('spam', 'eggs'))
spam and eggs
```

```
>>> print ('\'7b1\'7d and \'7b0\'7d'. format ('spam', 'eggs'))
eggs and spam
```
If named arguments are used in the str.format () method, their values are used based on the name of the arguments:

```
>>>
>>> print ('This \'7bfood\'7d is \'7badjective\'7d.'. format (
... food = 'spam', adjective = 'absolutely horrible'))
This spam is absolutely horrible.
```
Positioned and named arguments can be arbitrarily combined:

```
>>>
>>> print ('The story of \'7b0\'7d, \'7b1\'7d, and \'7bother\'7d.'. format ('Bill', 'Manfred',
                                       other = Georg '))
The story of Bill, Manfred, and Georg.
```
'! a' (apply ascii ()), '! s' (apply str ()) and '! r' (apply repr ()) can be used to convert values before they are formatted:

```
>>>
>>> content = 'eels'
>>> print ('My hovercraft is full of \'7b\'7d.'. format (contents))
My hovercraft is full of eels.
>>> print ('My hovercraft is full of \'7b! r\'7d.'. format (contents))
My hovercraft is full of 'eels'.
```
':' Characters followed by a formatting specification may follow the field name. This provides a finer level of control over how values are formatted. The following example rounds Pi to three decimal places:

```
>>>
>>> import math
>>> print ('The value of PI is approximately \'7b0: .3f\'7d.'. format (math.pi))
The value of PI is approximately 3.142.
```
Specifying an integer after the ':' indicates the minimum width of this field in number of characters. This is useful for making pretty paintings:

```
>>>
>>> table = \'7b'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678\'7d
>>> for name, phone in table.items ():
... print ('\'7b0:10\'7d ==> \'7b1: 10d\'7d'. format (name, phone))
...
Jack ==> 4098
Dcab ==> 7678
Sjoerd ==> 4127
```
If you really have a long formatting string that you do not want to split, it would be nice to be able to reference variables to format by their name rather than their position. This can be done simply by passing a dictionary and using '[]' brackets to access the keys:

```
>>>
>>> table = \'7b'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678\'7d
```

>>> print ('Jack: \'7b0 [Jack]: d\'7d; Sjoerd: \'7b0 [Sjoerd]: d\'7d;'
... 'Dcab: \'7b0 [Dcab]: d\'7d'. Format (table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
We could also do this by passing the array as named arguments using the notation "**"

>>> table = \'7b'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678\'7d
>>> print ('Jack: \'7bJack: d\'7d; Sjoerd: \'7bSjoerd: d\'7d; Dcab: \'7bDcab: d\'7d'. format (** table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
This is particularly useful in combination with the native vars () function, which returns a dictionary containing all local variables.

For a complete view of string formatting with the str.format () method, see: String formatting syntax.

7.1.1. Old methods of formatting strings
The% operator can also be used to format strings. It interprets the left argument a bit like a formatting string of a sprintf () function to be applied to the right argument, and returns the string resulting from this formatting operation. For example :

>>>
>>> import math
>>> print ('The value of PI is about% 5.3f.'% math.pi)
The value of PI is approximately 3.142.
You can find more information in the section Formatting channels at printf.

7.2. Reading and writing files
open () returns a file object, and is most often used with two arguments: open (filename, mode).

>>>
>>> f = open ('workfile', 'w')
The first argument is a string containing the file name. The second argument is another string containing a few characters describing how the file will be used. mode can be 'r' when the file will only be accessed in read, 'w' in write only (an existing file with the same name will be overwritten), and 'a' open the file in add mode (any written data in the file is automatically added at the end). 'r +' opens the file in read / write mode. The mode argument is optional; its default value is 'r'.

Normally, files are opened in text mode, ie you read and write strings from and into this file, which are encoded with a given encoding. If no encoding is specified, the default encoding will depend on the platform (see open ()). 'b' pasted at the end of the mode indicates that the file must be open in binary mode ie the data is read and written as bytes. This mode is for files containing anything other than text.

In text mode, the default behavior, on reading, is to convert the end of specific lines to the platform (\ n on Unix, on windows etc ...) into simple \ n. When writing, the default is to apply the opposite operation: the \ n are converted to their equivalent on the current platform. These auto-made changes are normal for text but would degrade binary data like a JPEG or EXE file. Be especially careful to open these binary files in binary mode.

It is good practice to use the keyword when dealing with file objects. The advantage is that the file is properly closed after its finishes, even if an exception is raised at some point. Using is also much shorter than writing equivalent try-finally blocks:

```
>>>
>>> with open ('workfile') as f:
... read_data = f.read ()
>>> f.closed
true
```

If you're not using the keyword, then you should call f.close () to close the file. If you do not explicitly close a file, Python's garbage collector will eventually destroy the object and open the file for you. Another risk is that different Python implementations will do this clean-up at different times.

After a file object is closed, by attempts to use the file object will automatically fail.

```
>>>
>>> f.close ()
>>> f.read ()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I / O operation on closed file
```

7.2.1. Methods of file objects

The last examples in this section will assume that a file object called f has already been created. To read the contents of a file, call f.read (size), which reads a certain amount of data and gives it as a string (in text mode) or in a bytes object (in binary mode). size is an optional numeric argument. When size is omitted or negative, the entire contents of the file are read and given, this is your problem if the file is twice as big as the memory of your machine. Otherwise, a maximum of size bytes are read and rendered. When the end of the file is reached, f.read () returns an empty string ('').

```
>>>
>>> f.read ()
'This is the entire file. \ N'
>>> f.read ()
''
```

f.readline () reads only one line of the file; an end-of-line character (\ n) is left at the end of the string, and is omitted only on the last line of the file if it does not end an end-of-line character. This makes it possible to make the return value unambiguous: if f.readline () returns an empty string, it means that the end of the file has been reached, whereas an empty line is represented by '\ n', a string characters containing only one end of line.

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

'Second line of the file \ n'
>>> f.readline ()
"
Another approach to reading lines is to loop the file object. This is more efficient in terms of memory management, faster, and gives a simpler code:

>>>
>>> for line in f:
... print (line, end = '')
...
This is the first line of the file.
Second line of the file
To read all the lines of a file, it is also possible to use list (f) or f.readlines ().

f.write (string) writes the string content to the file, and returns the number of characters written.

>>>
>>> f.write ('This is a test \ n')
15
Other types must be converted, either into a string (in text mode) or a bytes object (in binary mode), before writing them:

>>>
>>> value = ('the answer', 42)
>>> s = str (value) # convert the tuple to string
>>> f.write (s)
18
f.tell () returns an integer indicating the current position in the file, measured in bytes from the beginning of the file, when the file is opened in binary mode, or an obscure number in text mode.

To change the position in the file, use f.seek (offset, from_what). Laposition is calculated by adding offset to a reference point; this reference point is selected by the argument from_what: 0 for the beginning of the file, 1 for the current position, and 2 for the end of the file. from_what can be omitted and its default value is 0, using the beginning of the file as a reference point:

>>>
>>> f = open ('workfile', 'rb +')
>>> f.write (b'0123456789abcdef ')
16
>>> f.seek (5) # Go to the 6th byte in the file
5
>>> f.read (1)
b'5 '
>>> f.seek (-3, 2) # Go to the 3rd byte before the end
13
>>> f.read (1)
B'D '
On a text mode file (those opened without b in the mode), only positional changes relative to the beginning of the file are allowed (except one exception: go to the end of the file with seek (0, 2))

and the only possible values for the offset parameter are the values returned by f.tell (), or zero. Any other value for the offset parameter will generate undefined behavior.

Files have additional methods, such as isatty () and truncate () that are used less often; see the Standard Library Reference for a complete guide to file objects.

7.2.2. Back up structured data with the json module

Strings can easily be written to a file and read again. The numbers require a little more effort, because the read () method only returns strings, which must be passed to a function like int (), which takes a string like '123' as input and returns its numeric value 123. But as soon as you want to record more complex data types like lists, dictionaries, or class instances, things get a lot more complicated.

Rather than spend time writing and debugging code to save complicated data types, Python allows you to use JSON (JavaScript Object Notation), a popular format for data representation and exchange. The standard module named json can transform Python data hierarchies into their string representation. This process is named serialize. Rebuilding data from their string representation is called deserializing. Between serialization and deserialization, the data string may have been stored or forwarded to another machine.

Note The JSON format is commonly found in modern applications for exchanging data. Many developers are already familiar with JSON, making it a preferred format for interoperability.
If you have an x object, you can just see its JSON representation

```
>>>
>>> import json
>>> json.dumps ([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

A variation of the dumps () function, named dump (), simply serializes the given object to a text file. So if f is a text file open for writing, it becomes possible to do:

```
json.dump (x, f)
```

To rebuild the object, if f is an open text file for reading:

```
x = json.load (f)
```

This method of serialization can serialize lists and dictionaries, but serializing other types of data requires a little more work. The json module documentation explains how to do it.

8. Errors and exceptions

Until now, error messages have only been mentioned, but if you have tried the examples you have certainly seen more than that. In fact, there are at least two types of errors to distinguish: syntax errors and exceptions.

8.1. Syntax errors

Syntax errors, which are code analysis errors, may be the ones you encounter most often when you are still learning Python:

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

The analyzer repeats the offending line and displays a small "arrow" pointing to the first place in the line where the error was detected. The error is caused (or, at least, has been detected as such) by the symbol before the arrow. In this example the arrow is on the function print () because it misses two points (':') just before. The file name and line number are displayed so you can easily locate the error when the code comes from a script.

8.2. Exceptions

Even if an instruction or expression is syntactically correct, it can generate an error when it is executed. Errors detected during execution are called exceptions and are not always fatal: you will soon learn how to treat them in your programs. Most exceptions, however, are not supported by programs, which generates error messages like this:

```
>>>
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam * 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can not convert 'int' object to str implicitly
```

The last line of the error message indicates the cause of the error. The exceptions can be of different types, and this type is indicated in the message: The types shown in the example are ZeroDivisionError, NameError, and TypeError. The text displayed as the type of the exception is the name of the native exception that was raised. This is true for all native exceptions, but is not a

requirement for user-defined exceptions (even if it is a convenient convention). Standard exception names are native identifiers (not reserved words).

The rest of the line provides more details depending on the type of the exception and what caused it.

The previous part of the error message shows the context in which the exception occurred, in the form of an execution stack trace. In general, this contains the lines of the source code; however, lines read from standard input will not be displayed.

In Built-in Exceptions you will find the list of native exceptions and their meaning.

8.3. Exception Management

It is possible to write programs that support some exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to abort the program (using Control-C or another shortcut other than system supports); note that a user-generated trap is reported by throwing the KeyboardInterrupt exception.

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

The try statement works like this.

First, the try clause (statement (s) placed between the keywords try and except) is executed.
If no exceptions occur, the except clause is skipped and the execution of the try statement is complete.
If an exception occurs during the execution of the "try" clause, the rest of this clause is skipped. If its type matches an exception name specified after the except keyword, the corresponding "except" clause is executed, and the execution continues after the try statement.
If an exception occurs that does not match any exception mentioned in the "except" clause, it is passed to the top-level try statement; if no exception handler is found, this is an unhandled exception and execution stops with a message as shown above.
A try statement can have several except clauses to support different exceptions. But only one manager, at most, will be executed. Managers only support exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. But the same except clause can cite several exceptions in the form of a tuple in parentheses, as in this example:

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

A class in an except clause is compatible with an exception if it is the same class or a base class it is not compatible with a base class. For example, the following code will print B, C, D in that order:

```
class B (Exception):
```

```
   pass

class C (B):
   pass

class D (C):
   pass

for cls in [B, C, D]:
   try:
      raise cls ()
   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 exception clause is triggered.

The last except clause may omit the name (s) of exception (s), to serve as a wildcard. However, it is to use with great care, because it is very easy to hide a real programming error in this way. It can also be used to display an error message before re-raising the exception (allowing a caller to also support the exception)

```
import sys

try:
   f = open ('myfile.txt')
   s = f.readline ()
   i = int (s.strip ())
except OSError as err:
   print ("OS error: \'7b0\'7d". format (err))
except ValueError:
   print ("Could not convert data to an integer.")
except:
   print ("Unexpected error:", sys.exc_info () [0])
   raise
```

The try ... except statement also has an optional else clause that, when present, must follow all except clauses. It is useful for code that must be executed when no exception has been thrown by the try clause. For example :

```
for arg in sys.argv [1:]:
   try:
      f = open (arg, 'r')
   except IOError:
      print ('can not open', arg)
   else:
      print (arg, 'has', len (f.readlines ()), 'lines')
```

```
    f.close ()
```
It is better to use the else clause, rather than adding code to the try clause, as this avoids accidentally capturing an exception that was not thrown by the code initially protected by the try ... except statement.

When an exception occurs, a value can be associated with it, which is also called the exception argument. The presence of this argument and its type depend on the type of the exception.

The except clause can specify a variable name after the name of the exception. This variable is linked to an exception instance with the arguments stored in instance.args. For convenience, the instance of the exception sets the __str __ () method so that arguments can be displayed directly without having to reference .args. It is possible to build an exception, add its attributes, and then launch it later.

2

4386/5000

```
>>> try:
... raise Exception ('spam', 'eggs')
Except as inst:
... print (type (inst)) # the exception instance
... print (inst.args) # arguments stored in .args
... print (inst) # __str__ allows args to be printed directly,
... # but may be overridden in exception subclasses
... x, y = inst.args # unpack args
... print ('x =', x)
... print ('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```
If an exception has an argument, it is displayed in the last part ("detail") of the unhandled exception message.

Exception handlers do not only intercept exceptions that are raised immediately in their try clause, but also those that are raised within called functions (sometimes indirectly) in the try clause. For example :

```
>>>
>>> def this_fails ():
... x = 1/0
...
>>> try:
... this_fails ()
... except ZeroDivisionError as err:
... print ('Handling run-time error:', err)
...
```

Handling run-time error: division by zero
8.4. Trigger exceptions
The raise statement allows the programmer to raise a specific exception. For example :

```
>>>
>>> raise NameError ('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to raise indicates the exception to be raised. This must be an exception instance or an exception class (a class that derives from Exception). If an exception is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError ()'
```

If you need to know that an exception has been thrown but not intended to support it, a simpler form of the raise statement will re-trigger the exception:

```
>>>
>>> try:
... raise NameError ('HiThere')
... except NameError:
... print ('An exception flew by!')
... raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

8.5. User defined exceptions
Programs can name their own exceptions by creating a new exception class (see Classes to learn more about Python classes). Exceptions are typically derived from the Exception class, directly or indirectly.

Exception classes can be set to do anything another class can do. But they are usually quite simple, offering only the attributes that allow the managers of these exceptions to retrieve information about the error that has occurred. When creating a module that can trigger several different types of errors, a common practice is to create a base class for all the exceptions defined in this module, and to create specific subclasses of exceptions for the different error conditions:

```
class Error (Exception):
   "" "Base class for exceptions in this module." ""
   pass

class InputError (Error):
   "" "Exception raised for errors in the input.

   Attributes:
      expression - input expression in which the error occurred
      message - explanation of the error
```

```
    "" "

    def __init __ (self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError (Error):
    "" "Raised when the operations a transition transition that's not
    allowed.

    Attributes:
        previous - state at beginning of transition
        next - attempted new state
        message - explanation of why the specific transition is not allowed
    "" "

    def __init __ (self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Most exceptions are defined with names that end with "Error", just as for standard exceptions.

Most standard modules define their own exceptions to describe errors that may be encountered in the functions they define. More information about classes is presented in the Classes chapter.

8.6. Definition of cleaning actions

The try statement has another optional clause that is intended to define cleanup actions that should be performed under certain circumstances. For example :

```
>>> try:
... raise KeyboardInterrupt
... finally:
... print ('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

A finally clause is always executed before leaving the try statement, regardless of whether an exception was thrown or not. When an exception was triggered in the try clause and was not supported by a except clause (or if it was triggered in a except or else clause), it is re-triggered after the finally clause is executed . The finally clause is also executed "on exit" when any other clause of the try statement is dropped by a break, continue, or return statement. Here is a more complicated example:

```
>>>
>>> def divide (x, y):
... try:
```

```
... result = x / y
... except ZeroDivisionError:
... print ("division by zero!")
... else:
... print ("result is", result)
... finally:
... print ("executing finally clause")
...
>>> divide (2, 1)
result is 2.0
executing finally clause
>>> divide (2, 0)
division by zero!
executing finally clause
>>> divide ("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type (s) for /: 'str' and 'str'
```

As you can see, the finally clause is executed in all cases. The typeError exception raised by dividing two strings is not supported by the except clause and is re-triggered after the finally clause has been executed.

In actual applications, the finally clause is particularly useful for releasing external resources (such as files or network connections), whether or not the use of these resources is successful.

8.7. Predefined cleanup actions

Some objects define standard cleanup actions that must be performed when the object is no longer needed, regardless of whether the operation that used the object was successful or not. Look at the following example, which attempts to open a file and display its contents on the screen:

```
for line in open ("myfile.txt"):
    print (line, end = "")
```

The problem with this code is that it leaves the file open indefinitely after the code finishes executing. This is not a problem with simple scripts, but can be in larger applications. The with statement allows you to use certain objects as files in a way that ensures that they will always be cleaned up quickly and correctly.

```
with open ("myfile.txt") as f:
    for line in f:
        print (line, end = "")
```

As soon as the instruction is executed, the file f is always closed, even if a problem occurred during the execution of these lines. Other objects that, like files, provide predefined cleanup actions indicate this in their documentation.

9. Classes

The mechanism of Python classes adds to the language the notion of classes with a minimum of new syntax and semantics. It is a mixture of the mechanisms encountered in C ++ and Modula-3. In the same way as for modules, Python classes do not pose a rigid barrier between their definition and the user, but rely on the respect of the user not to cause break-ins in the definition. However, the most important features of classes are preserved with all their power: the inheritance mechanism allows to have several basic classes, a derived class can overload all the methods of its (or its) class (es) base and a method can use the method of a base class with the same name. Objects can contain an arbitrary number of data.

In C ++ terminology, class members (including data) are public (except for exception, see Private variables) and all member functions are virtual. As with Modulo-3, there is no way to access the members of an object from its methods: a method is declared with an explicit first argument representing the object, and this argument is transmitted implicitly during the call. As with Smalltalk, the classes themselves are objects. There is a semantic for importing and renaming them. Unlike C ++ and Modulo-3, base types can be used as base classes for the user to extend. Finally, as in C ++, most basic operators with a special syntax (arithmetic operators, subindicating, etc.) can be redefined for class instances.

(Due to the lack of a universally accepted terminology for talking about classes, we will make occasional use of the terms of Smalltalk and C ++.We wanted to use Modula-3 terms since its object-oriented semantics is closer to that of Python than to C ++. but it is likely that only a small number of readers are likely to know them.)

9.1. A few words about names and objects

Objects have their own existence and several names can be used (in various contexts) to refer to the same object. These are known as aliases in other languages. This is usually unappreciated at first glance at Python and can be ignored when working with non-mutable base types (numbers, strings, tuples). However, aliases may have surprising effects on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used for the benefit of the program because the aliases behave, from a certain point of view, as pointers. For example, passing an object has no cost because it is simply a pointer that is transmitted by the implementation; and if a function modifies an object passed as an argument, the code at the origin of the call will see the change. This eliminates the need for two arguments transmission mechanisms as in Pascal.

9.2. Python Scopes and Namespaces

Before introducing classes, we need to talk a little bit about the notion of Python scope. Class definitions make clever use of namespaces, and you need to know how scopes and namespaces work. By the way, any knowledge on this topic is also helpful to experienced Python developers.

First, some definitions.

A namespace is a lookup table between names and objects. Most namespaces are currently

implemented as Python dictionaries, but this is not normally visible (except for performance) and may change in the future. Examples of namespaces include primitives (functions like abs (), and base exception names); global names in a module; and local names during a function call. In a way, the set of attributes of an object itself forms a namespace. The important thing to remember about namespaces is that there is absolutely no connection between the names of multiple namespaces; for example, two different modules can define a maximize function without confusion. Users of the modules must prefix the name of the function with that of the module.

In this regard, we use the word attribute for any name following a point. For example, in the z.real expression, real is an attribute of the z object. Rigorously speaking, references to names in modules are attribute references: in the modname.funcname expression, modname is a module object and funcname is an attribute of that object. Under these conditions, there is a direct correspondence between the attributes of the module and the global names defined in the module: they share the same namespace! [1]

Attributes can only be readable or editable. If they are modifiable, the assignment to an attribute is possible. Module attributes are editable: you can write modname.the_answer = 42. Editable attributes can also be deleted with the del statement. For example, del modname.the_answer removes the_answer attribute from the object named modname.

Namespaces are created at different times and have different lifetimes. The namespace containing the primitives is created when the Python interpreter starts and is never deleted. The global namespace for a module is created when the module definition is read. Usually, module namespaces also last until you stop the interpreter. Instructions executed by the first interpreter invocation, whether read from a script file or interactively, are considered part of a module called __main__, so that they have their own namespace . (the primitives live in a module themselves, called builtins.)

The local namespace of a function is created when it is called, and then cleared when it returns a result or throws an unsupported exception. (In fact, "forgotten" would be a better way to describe what is really happening). Of course, recursive invocations each have their own namespace.

A scope is a text box in a Python program where a namespace is directly accessible. "Directly accessible" means here that an unqualified reference to a name will be searched in the namespace.

Although the staves are determined statically, they are used dynamically. At any point during execution, there are at least three nested staves whose namespaces are directly accessible:

The most central reach, the one that is consulted first, contains the local names
Scopes of bounding functions, which are accessed starting with the closest bounding scope, contain non-local but also non-global names
the penultimate scope contains the global names of the current module
the enclosing scope, last viewed, is the namespace containing the primitives
If a name is declared global, all references and assignments go directly into the intermediate scope containing the global names of the module. To reattach variables found outside the most local scope, the nonlocal statement can be used. If they are not declared nonlocal, these variables are read-only (any attempt to modify such a variable will simply create a new variable in the most local scope, leaving the variable of the same name unchanged in its original scope).

Usually, the local scope refers to the local names of the current function. Outside of functions, the local scope references the same namespace as the global scope: the namespace of the module. Class definitions create a new namespace in the local scope.

It is important to realize that the scopes are determined in a textual way: the global scope of a function defined in a module is the namespace of this module, whatever the origin of this call. On the other hand, the actual search for names is done dynamically at the time of execution. However the definition of the language is evolving towards a static resolution of the names at the time of the "compilation", thus without being based on a dynamic resolution! (In reality, local variables are already determined statically).

A special feature of Python is that if no global statement is active, the name assignments always go to the nearest scope. Assignments do not copy any data: they simply bind names to objects. This is also true for deletion: the del x statement removes the binding of x in the namespace referenced by the local scope. In fact, all operations that involve 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 certain variables exist in the global scope and must be locally linked; the nonlocal instruction indicates that some variables exist in a higher scope and must be locally bound.

9.2.1. Example of scopes and namespaces

This is an example showing how to use the different scopes and namespaces, and how global and nonlocal modify the variable assignment:

```
def scope_test ():
    def do_local ():
        spam = "local spam"

    def do_nonlocal ():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global ():
        global spam
        spam = "global spam"

    spam = "spam test"
    do_local ()
    print ("After local assignment:", spam)
    do_nonlocal ()
    print ("After nonlocal assignment:", spam)
    do_global ()
    print ("After global assignment:", spam)

scope_test ()
print ("In global scope:", spam)
```

This code gives the following result:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

You may find that the local assignment (which is done by default) has not changed the spam binding in scope_test. The nonlocal assignment changed the spam binding in scope_test and the global assignment changed the link at the module level.

You can also see that no spam links have been made before the global assignment.

9.3. A first approach to classes

The concept of classes introduces some new syntax elements, three new object types as well as new semantic elements

9.3.1. Syntax for defining classes

The simplest form of class definition looks like this:

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

Class definitions, such as function definitions (def definitions), must be executed before having an effect. (You can quite put a class definition in a branch of a conditional if statement or inside a function.)

In practice, statements in a class definition will usually be function definitions, but other statements are allowed, and sometimes helpful - we'll come back to this later. Function definitions within a class normally have a particular form of argument list, dictated by the method call conventions - again, all of this will be explained later.

When a class is defined, a new namespace is created and used as the local scope - so all local variable assignments enter this new namespace. In particular, function definitions link the name of the new function.

At the end of the definition of a class, a class object is created. This is, for simplicity, an encapsulation of the namespace content created by the class definition. We will talk again about objects that are classes in the next section. The initial local scope (the one that prevails before the beginning of the class definition) is reinstantiated, and the class object is linked here to the given class name in the class definition header (ClassNameName in the class definition header). example).

9.3.2. Class objects

Class objects support two types of operations: references to attributes and instantiation.Attribute references use the standard syntax used for all Python attribute references: obj.name. Valid attribute names are all names that were in the namespace of the class when the class object was created. So, if the class definition looks like this:

```
MyClass class:
   "" "A simple example class" ""
   i = 12345

   def f (self):
      return 'hello world'
```

then MyClass.i and MyClass.f are valid references to attributes, returning an integer and a function object respectively. Class attributes can also be assigned, so you can change the value of MyClass.i by assignment. __doc__ is also a valid attribute, returning the docstring belonging to the class: "A simple class of example".

Class instantiation uses function notation. Just consider that the class object is a function without parameters that returns a new instance of the class. For example (considering the class defined above)

```
        x = MyClass ()
```

creates a new instance of the class and assigns that object to the local variable x.

The instantiation operation (by "calling" a class object) creates an empty object. Many classes like to create custom objects with custom instances based on a specific initial state. So a class can define a special method named: __init __ (), like this:

```
def __init __ (self):
   self.data = []
```

When a class defines a __init __ () method, the instantiation of the class automatically calls __init __ () for the new instance of the class. So, in this example, the initialization of a new instance can be obtained by:

```
x = MyClass ()
```

Of course, the __init __ () method can have arguments for greater flexibility. In this case, the arguments given to the class instantiation operator are passed 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)
```

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 given attributes and the methods.

The attributes given correspond to "instance variables" in Smalltalk, and to "data members" in C ++. The given attributes do not have to be declared. Like local variables, they exist as soon as they are allocated a first time. For example, if x is the instance of MyClass created above, the following code displays the value 16, without leaving any traces:

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

The other type of reference to an instance attribute is a method. A method is a function that "belongs to" an object (in Python, the method term is not unique to class instances: other types of objects can also have methods.) For example, list objects have methods called append, insert, remove, sort, and so on, but in the following discussion, unless otherwise noted, we will use the term method exclusively with reference to class instance object methods).

The valid method names of an instance object depend on its class. By definition, all the attributes of a class that are function objects define the corresponding methods of its instances. So, in our example, x.f is a valid reference method because MyClass.f is a function, but not x.i because MyClass.i is not one. Be careful though, x.f is not the same as MyClass.f - It's a method object, not a function object.

9.3.4. Method objects

Most often, a method is called just after being linked:

```
x.f ()
```

In the example of the class MaClass, this will return the string hello world. However, it is not necessary to call the method directly: x.f is a method object, it can be kept aside and called later. For example:

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

will show hello world until the end of time.

What exactly happens when a method is called? You must have noticed that x.f () was called in the code above without arguments, while the definition of the f () method specifies that it takes an argument. What happened to the argument? Python must throw an exception when a function that requires an argument is called without - even if the argument is not 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, the method is in the form of an argument that is created by inserting the method object instance before the first argument.

If you still do not understand how the methods work, a look at the implementation may help you. When the instance of an attribute is referenced that is not a given attribute, its class is searched. If the name corresponds to a valid attribute that is a function, a method object is created by associating (via their pointers) the instance object and the function object found together in a new abstract object: this is the method object. When the method object is called with a list of arguments, a new argument list is constructed from the method object and the argument list. The function object is called with this new argument list.

9.3.5. Instance classes and variables

In general, instance variables store information about each instance while class variables are used to store attributes and methods common to all instances of the class:

Dog class:

```
    kind = 'canine' # class variable shared by all instances

    def __init __ (self, name):
        self.name = name # variable instance unique to each instance
```

```
>>> d = Dog ('Fido')
>>> e = Dog ('Buddy')
>>> d.kind # shared by all dogs
'canine'
>>> e.kind # shared by all dogs
'canine'
>>> d.name # unique to d
'Fido'
>>> e.name # unique to e
'Buddy'
```

As seen in A Few Words About Names and Objects, editable shared data (such as lists, dictionaries, etc.) can have surprising effects. For example, the list tricks in the following code should not be a class variable, because jiate only one list would be shared by all instances of Dog:

Dog class:

```
    tricks = [] # mistaken use of a variable class

    def __init __ (self, name):
        self.name = name

    def add_trick (self, trick):
        self.tricks.append (trick)
```

```
>>> d = Dog ('Fido')
```

```
>>> e = Dog ('Buddy')
>>> d.add_trick ('roll over')
>>> e.add_trick ('play dead')
>>> d.tricks # unexpectedly shared by all dogs
['roll over', 'play dead']
```
A correct design of the class would be to use an instance variable instead:

Dog class:

```
   def __init __ (self, name):
      self.name = name
      self.tricks = [] # creates a new empty list for each dog

   def add_trick (self, trick):
      self.tricks.append (trick)
```

```
>>> d = Dog ('Fido')
>>> e = Dog ('Buddy')
>>> d.add_trick ('roll over')
>>> e.add_trick ('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4. Miscellaneous remarks

Data attributes override methods with the same name; to avoid naming conflicts, which can cause hard-to-find bugs in large programs, it is wise to adopt certain conventions that minimize the risk of conflict. Possible conventions include capitalizing method names, prefixing data attribute names with a short and unique string (sometimes just the underlined character), or using verb expressions for methods and names for names. data attributes.

Data attributes can be referenced by methods as by ordinary users ("clients") of an object. In other words, classes can not be used to implement purely abstract data types. In fact, nothing in Python makes it possible to impose masking of data - everything is based on conventions (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 Python extensions written in C).

Clients should use data attributes with care - they could mess up the invariants managed by the methods with their own attribute values. Note that clients can add their own data attributes to an object instance without altering the validity of the methods, as long as the names do not conflict - so adopting a naming convention can avoid many problems.

There is no abbreviated notation for referencing data attributes (or other methods!) From methods. We think this actually improves the readability of the methods: there is no chance of confusing local variables and instance variables when looking at the code of a method.

Often, the first argument of a method is named self. This is just a convention: the name self has no

special meaning in Python. Note, however, that if you do not follow this convention, your code may be less readable for other Python programmers, and it is also possible that a program that does introspection of classes relies on such a convention.

Any function object that is a class attribute defines a method for instances of that class. The definition text of the function does not have to be in the class definition: it is possible to assign a function object to a local variable of the class. 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 classes C referring to object functions, and therefore all methods of instances of C - h are exactly the same as g. Note that in practice this only confuses the reader of a program.

Methods can call other methods by using methods that are attributes of the self argument

```
class Bag:
   def __init __ (self):
      self.data = []

   def add (self, x):
      self.data.append (x)

   def addtwice (self, x):
      self.add (x)
      self.add (x)
```

Methods can refer to global names in the same way as functions. The global scope associated with a method is the module containing the definition of the class (the class itself is never used as a global scope). While there is rarely a good reason to use global data in a method, there are many legitimate uses of global scope: for example, imported functions and modules in a global scope can be used by methods , as well as the functions and classes defined in this same scope. Usually, the class containing the method is itself defined in this global scope, and in the next section, we will see good reasons for a method to refer to its own class.

Any value is an object, and therefore has a class (also called its type). It is stored in object .__ class__.

9.5. The legacy

Of course, this term "class" would not be used if there was no inheritance. The syntax for

defining a subclass looks like this:

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

The ClassNameName must be defined in a space containing the definition of the derived class. Instead of the name of a base class, an expression is also allowed. This can be useful, for example, when the class is defined in another module:

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

The execution of a derived class definition proceeds as for a base class. When the object of the class is built, the base class is memorized. It is used for the resolution of attribute references: if an attribute is not found in the class, the search proceeds by looking in the base class. This rule is applied recursively if the base class itself is derived from another class.

There is nothing special about the instantiation of derived classes: DerivedClassName () creates a new instance of the class. The references to the methods are solved as follows: the corresponding attribute of the class is searched, going up the base class hierarchy if necessary, and the method reference is valid if it leads to a function.

Derived classes can override methods from their base classes. Since methods have no particular privilege when they call other methods of the same object, a method of a base class that calls another method defined in the same class may actually call a method of a class derivative that overload it (for C ++ programmers: all methods of Python are indeed virtual).

An overloaded method in a derived class may actually want to extend rather than simply replace the method of the same name from its base class. There is a simple way to call the base class method directly: just call BaseClassName.methodname (self, arguments). This is sometimes useful for clients as well (note that this only works if the base class is accessible as a ClassBase in the global scope).

Python has two primitive functions that handle inheritance:

Use isinstance () to test the type of an instance: isinstance (obj, int) will return True only if obj .__ class__ is equal to int or another class derived from int.
Use issubclass () to test the inheritance of a class: issubclass (bool, int) returns True because the class bool is a subclass of int. On the other hand, issubclass (float, int) returns False because float is not a subclass of int.

9.5.1. Multiple inheritance

Python also supports a form of multiple inheritance. A class definition with several basic classes looks like:

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

.
<Statement-N>
In most cases, you can imagine looking for attributes in the parent classes as: the deepest first, from left to right, without searching twice in the same class if it appears several times in the hierarchy. Thus, if an attribute is not found in DerivedClassName, it is searched in Base1, and then (recursively) in base classes of Base1; if it is not found, it is searched in Base2 and its base classes, and so on.

In fact, it's a little more complex than that, the order resolution order (or MRO) dynamically changes to handle cooperative calls to super (). This approach is known as the "call-next-method" in other languages supporting multiple inheritance, and is more powerful than the call to super found in languages to simple inheritance.

The dynamically defined order is needed because all multiple inheritance cases have one or more diamond relationships (where at least one class can be accessed from multiple paths by betting the most basic class). For example, since all classes inherit from object, multiple inheritance opens multiple paths to achieve object. For a base class not to be called multiple times, the dynamic algorithm linearizes the search order in a way that preserves the inheritance order from left to right, specified in each class, that calls each parent class only once, which is monotonous (which means that a class can be subclassed without affecting the inheritance order of its parents). Taken together, these properties make it possible to design classes reliably and extensible in a context of multiple inheritance.

9.6. Private variables

"Private" members, which can not be accessed outside an object, do not exist in Python. However, there is a convention respected by the majority of Python code: a name prefixed by a low hyphen (such as _spam) must be seen as a non-public part of the API (whether it is a function, a 'a method or member variable). It must be considered as an implementation detail that can be modified without notice.

Since there is a valid use case to have private members (especially to avoid conflicts with names defined in subclasses), there is support (albeit limited) for such a mechanism, called name mangling. Any identifier in the form __spam (with at least two underscores at the beginning, and at most one at the end) is replaced literally by _classname__spam, where classname is the name of the class without the first underscore (s) . This "DIY" is done without taking into account the syntactic position of the identifier, as long as it is present in the definition of a class.

This name change is useful for allowing subclasses to override methods without interrupting intra-class method calls. For example :

```
class Mapping:
   def __init __ (self, iterable):
      self.items_list = []
      self .__ update (iterable)

   def update (self, iterable):
      for item in iterable:
```

```
        self.items_list.append (item)

    __update = update # private copy of original update () method

class MappingSubclass (Mapping):

    def update (self, keys, values):
        # provides new signature for update ()
        # but does not break __init __ ()
        for item in zip (keys, values):
            self.items_list.append (item)
```

Note that these rules are designed primarily to prevent accidents; it remains possible to access or modify a variable considered as private. This may even be useful in some circumstances, such as within the debugger.

Note that code passed to exec (), eval () does not consider the name of the calling class as the current class; the same effect applies to the global directive, whose effect is similarly restricted to compiled code in the same byte-code set. The same restrictions apply to getattr (), setattr (), and delattr (), as well as direct references to __dict__.

9.7. Tips and tricks

It is sometimes useful to have a datatype similar to Pascal's "record" or C's "struct", which group together some named attributes. The definition of an empty class perfectly fulfills this need:

```
Employee class:
    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
```

Python code that expects to receive a specific type of abstract data can often be provided with a class that simulates such methods. For example, if you have a function that formats data extracted from a file object, you can define a class with read () and readline () methods that extracts its data from a string buffer instead, and pass him an instance as an argument.

Instance method objects also have attributes: m.im_self is the object instance with the m () method, and m.im_func is the function object corresponding to the method.

9.8. iterators

You have now certainly noticed that you can iterate over most container objects by using a for statement

```
for element in [1, 2, 3]:
```

```
    print (element)
for element in (1, 2, 3):
    print (element)
for key in \'7b'one': 1, 'two': 2\'7d:
    print (key)
for char in "123":
    print (char)
for line in open ("myfile.txt"):
    print (line, end = '')
```

This mode of access is simple, concise and practical. The use of iterators permeates and unifies Python. In the background, the for statement calls the iter () function on the container object. This function returns an iterator that defines the __next __ () method, which accesses the container's elements one by one. When there is no more element, __next __ () throws a StopIteration exception that tells the for statement's loop to complete. You can call the __next __ () method using the native next () function. This example shows how it all works:

```
>>>
>>> s = 'abc'
>>> it = iter (s)
>>> it
<iterator object at 0x00A1DB50>
>>> next (it)
'at'
>>> next (it)
'B'
>>> next (it)
'C'
>>> next (it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next (it)
StopIteration
```

Once you understand the iterator management mechanisms, it's easy to add this behavior to your classes. Define an __iter __ () method, which returns an object with a __next __ () method. Sila class itself defines the __next __ () method, so __iter __ () can simply return self

```
class Reverse:
    "" "Iterator for looping over a sequence backwards." ""
    def __init __ (self, data):
        self.data = data
        self.index = len (data)

    def __iter __ (self):
        return self

    def __next __ (self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
```

```
        return self.data [self.index]
>>>
>>> rev = Reverse ('spam')
>>> iter (rev)
<__ hand __. Reverse object at 0x00A1DB50>
>>> for char in rev:
... print (char)
...
m
at
p
s
```

9.9. Generators

Generators are simple and powerful tools for creating iterators. They are written as standard functions but use the yield statement when they want to return data. Whenever next () is called, the builder resumes execution where it left off (keeping all of its execution context). An example shows very well how generators are simple to create:

```
def reverse (data):
    for index in range (len (data) -1, -1, -1):
        yield data [index]
>>>
>>> for char in reverse ('golf'):
... print (char)
...
f
l
o
g
```

Anything that can be done with generators can also be done with class-based iterators, as described in the previous paragraph. If that makes the generators so compact, the __iter __ () and __next __ () methods are created automatically.

Another key feature is that local variables and the execution context are automatically saved between calls. This further simplifies the writing of these functions, and makes their code much more readable than with an approach using instance variables such as self.index and self.data.

In addition to automatically creating methods and saving the execution context, builders automatically raise a StopIteration exception when they complete their execution. Combined, these features make it very easy to create iterators without more effort than writing a classic function.

9.10. Expressions and generators

Simple generators can be coded very quickly with expressions using the same syntax as list comprehensions, but using parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away in a function. These expressions are more compact but less flexible than complete definitions of generators, and tend to be more

memory efficient than their list comprehension counterparts.

Examples:

```
>>>
>>> sum (i * i for i in range (10)) # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum (x * y for x, y in zip (xvec, yvec)) # dot product
260

>>> from math import pi, sin
>>> sine_table = \'7bx: sin (x * pi / 180) for x in range (0, 91)\'7d

>>> unique_words = set (word for line in page for word in line.split ())

>>> valedictorian = max ((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list (data [i] for i in range (len (data) -1, -1, -1))
['f', 'l', 'o', 'g']
```

Notes

[1] Except for one thing. The modules have a read-only secret attribute called __dict__, which returns the dictionary used to implement the module namespace; the name __dict__ is an attribute but not a global name. Obviously, its use breaks the abstraction of the implementation of namespaces, and should be restricted only to things like post-mortem debuggers.

10. Overview of the Standard Library

10.1. Interface with the Operating System

The os module offers plethora functions to interact with the operating system:

```
>>>
>>> import os
>>> os.getcwd () # Return the current working directory
'C: \\ Python35'
>>> os.chdir ('/ server / accesslogs') # Change current working directory
>>> os.system ('mkdir today') # Run the command mkdir in the shell system
0
```

But, again, prefer import os, to from os import *, otherwise os.open () would hide the primitive open (), which works differently.

The dir () and help () primitives are useful tools when working in interactive mode have big

modules like bones

>>>
>>> import os
>>> dir (os)
<list of all module functions>
>>> help (os)
<returns to extensive manual page from the module's docstrings>
For file and folder management, the shutil module exposes a more abstract and easier-to-use interface:

>>>
>>> import shutil
>>> shutil.copyfile ('data.db', 'archive.db')
'Archive.db'
>>> shutil.move ('/ build / executables', 'installdir')
'Installdir'

10.2. Jokers on File Names

The glob module provides a function to build file lists from patterns:

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

10.3. Online Order Settings

Typically, command line tools need to read the parameters given to them. These parameters are stored in the argv variable in the sys module as a list. For example, the following display comes from running python demo.py one two three from the command line:

>>>
>>> import sys
>>> print (sys.argv)
['demo.py', 'one', 'two', 'three']
The getopt module includes sys.argv using the usual Unix function conventions getopt (). More flexible and advanced command line understanding tools are available in the argparse module.

10.4. Redirection of the error output and end of execution

The sys module also has attributes for stdin, stdout, and stderr. The latter is useful for issuing warning or error messages that remain visible even if stdout is redirected:

>>>
>>> sys.stderr.write ('Warning, log file not found starting a new one \ n')
Warning, log file not found starting a new one
The most direct way to complete a script is to use sys.exit ().

10.5. Pattern Search in Channels

The re module provides regular expression-based tools for complex chain operations. It is an optimized solution, using a consise syntax, to look for complex patterns, or to make complex substitutions in strings:

```
>>>
>>> import re
>>> re.findall (r '\ bf [a-z] *', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub (r '(\ b [a-z] +) \ 1', r '\ 1', 'cat in the hat')
'cat in the hat'
```

When the operations are simple, it is better to use the methods of the chains, they are more readable and easier to debug:

```
>>>
>>> 'tea for too'.replace (' too ',' two ')
'tea for two'
```

10.6. Mathematics

The math module exposes operation functions on the floats of library C

```
>>>
>>> import math
>>> math.cos (math.pi / 4)
0.70710678118654757
>>> math.log (1024, 2)
10.0
```

The random module offers tools for making random selections:

```
>>>
>>> import random
>>> random.choice (['apple', 'pear', 'banana'])
'Apple'
>>> random.sample (range (100), 10) # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random () # random float
0.17970987693706186
>>> random.randrange (6) # random integer chosen from range (6)
4
```

The statistics module is used to calculate basic statistical values (average, median, variance, ...)

```
>>>
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean (data)
1.6071428571428572
```

```
>>> statistics.median (data)
1.25
>>> statistics.variance (data)
1.3720238095238095
```
The SciPy project <https://scipy.org> contains many other modules around numerical calculations.

10.7. Internet access

There is a whole state of modules for accessing the internet and managing protocols used on the internet. The two simplest ones are urllib.request, which can download from a URL, and smtplib to send emails:

```
>>>
>>> from urllib.request import urlopen
>>> with urlopen ('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
... for line in response:
... line = line.decode ('utf-8') # Decoding the binary data to text.
... if 'EST' in line or 'EDT' in line: # look for Eastern Time
... print (line)

<BR> Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP ('localhost')
>>> server.sendmail ('soothsayer@example.org', 'jcaesar@example.org',
... "" "To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... "" ")
>>> server.quit ()
```
(Note that the second example needs a mail server running locally.)

10.8. Dates and times

The datetime module provides classes for manipulating dates and times, whether the need is simple or complicated. Although doing date calculations is possible, the implementation to be optimized for access to properties, formatting and manipulation. The module also manages objects aware of time zones:

```
>>>
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today ()
>>> now
datetime.date (2003, 12, 2)
>>> now.strftime ("% m-% d-% y.% d% b% Y is a% A on the% d day of% B.")
'12 -02-03. 02 Dec 2003 is a Tuesday on the 02 day of December. '
```

```
>>> # dates support calendar arithmetic
>>> birthday = date (1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9. Data compression

The most common archiving and compression formats are directly managed by zlib, gzip, bz2, lzma, zipfile, and tarfile modules

```
>>>
>>> import zlib
>>> s = b'witch which has which witches wrist watch '
>>> len (s)
41
>>> t = zlib.compress (s)
>>> len (t)
37
>>> zlib.decompress (t)
b'witch which has which witches wrist watch '
>>> zlib.crc32 (s)
226805979
```

10.10. Performance Measurement

Some Python users develop a deep interest in the performance of different approaches to the same problem. Python offers a measurement tool that simply answers these questions.

For example, to exchange two variables, it may be tempting to use the packaging and unpacking of tuples rather than the traditional method. The timeit module simply shows which one is the most efficient:

```
>>>
>>> from timeit import Timer
>>> Timer ('t = a; a = b; b = t', 'a = 1; b = 2'). Timeit ()
0.57535828626024577
>>> Timer ('a, b = b, a', 'a = 1, b = 2'). Timeit ()
0.54962537085770791
```

In contrast to timeit and its fine granularity, profile and pstats provide tools to identify the most time-consuming parts of execution in larger code volumes.

10.11. Quality Control

One possible approach to developing high quality applications is to write tests for each function as it develops, and to run these tests frequently during the development process.

The doctest module is used to search for tests in documentation strings. A test looks like a simple copy-paste of a call and its result from the interactive mode. This improves the documentation by

providing examples while proving that they are just:

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

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

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

The unittest module is heavier than the doctest module, but it allows you to build a more complete, maintainable, and understandable test set in a separate file:

```
import unittest

class TestStatisticalFunctions (unittest.TestCase):

   def test_average (self):
      self.assertEqual (average ([20, 30, 70]), 40.0)
      self.assertEqual (round (average ([1, 5, 7]), 1), 4.3)
      with self.assertRaises (ZeroDivisionError):
         average ([])
      with self.assertRaises (TypeError):
         average (20, 30, 70)

unittest.main () # Calling from the line invokes all tests
```

10.12. Batteries Supplied

Python respects the "batteries provided" philosophy. It's more obvious looking at the sophisticated and solid capabilities of its larger packages. For example:

The modules xmlrpc.client and xmlrpc.server let you call functions remotely with virtually no effort. Despite the name of the modules, no knowledge of XML is needed.
The email package is a library for managing e-mail messages, including MIME and other encodings based on RFC 2822. Unlike smtplib and poplib, which send and receive messages, the e-mail package is a toolbox for building, reading and complex message structures (including attachments), or implement encodings and protocols.
The json package reads and writes JSON, a popular data encoding format. The csv module handles the reading and writing of data stored as comma-separated values in Coma-Separated Values, typical of databases and spreadsheets. For reading XML, use the xml.etree.ElementTree, xml.dom, and xml.sax packages. Together, these modules and packages greatly simplify the exchange of data between Python applications and other tools.
The sqlite3 module is an abstraction of the SQLite library, allowing to manipulate a persistent database, accessed and manipulated using a quasi standard SQL syntax.
Internationalization is possible thanks to many packages, like gettext, local, or codecs.

11. Brief Tour of the Standard Library - Part II

This second visit will make you discover modules of a more professional use. These modules are rarely needed in small scripts.

11.1. formatting

The reprlib module is a variation of the repr () function, which specializes in concisely displaying large or heavily nested containers:

```
>>>
>>> import reprlib
>>> reprlib.repr (set ('supercalifragilisticexpialidocious'))
"\'7b'a', 'c', 'd', 'e', 'f', 'g', ...\'7d"
```

The pprint module offers a finer control of the display of objects, both primitive and user defined, and often readable by the interpreter. When the result is more than one line, it is separated on several lines and indented to make the structure more visible:

```
>>>
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
... 'yellow'], 'blue']]]
...
>>> pprint.pprint (t, width = 30)
[[[[" black ',' cyan '],
   'White'
   ['green', 'red']],
  [['magenta', 'yellow'],
   'Blue']]]
```

The textwrap module formats paragraphs of text to fit on a screen of a given width:

```
>>>
>>> import textwrap
>>> doc = "" "The wrap () is just like fill () except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines. "" "
...
>>> print (textwrap.fill (doc, width = 40))
The wrap () method is just like
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

The local module provides a database of data formats specific to each region. The grouping attribute of the formatting function allows you to directly format numbers with a separator:

```
>>>
>>> local import
```

```
>>> locale.setlocale (locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = local.localeconv () # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format ("% d", x, grouping = True)
'1,234,567'
>>> locale.format_string ("% s%. * f", (conv ['currency_symbol'],
... conv ['frac_digits'], x), grouping = True)
'$ 1,234,567.80'
```

11.2. Templates

The string module contains a very versatile class: Template for writing templates (called "templates") with a simple syntax, so simple that it is understandable by non-developers. This allows your users to customize their application without modifying it.

The format consists of markers consisting of a $ followed by a valid Python identifier (alphanumeric characters and underscores). Surrounding the braces marker allows you to paste other alphanumeric characters without interposing a space. Writing $$ created a simple $.

```
>>>
>>> from string import Template
>>> t = Template ('$ \'7bvillage\'7d folk send $ 10 to $ cause.')
>>> t.substitute (village = 'Nottingham', cause = 'the ditch fund')
'Nottinghamfolk send $ 10 to the ditch fund.'
```

The override () method throws a KeyError exception when a marker has not been provided, neither in a dictionary, nor as a named parameter. In some cases, when the data to be applied is only partially known, the safe_substitute () method is more appropriate because it will leave the missing markers as they are:

```
>>>
>>> t = Template ('Return the $ item to $ owner.')
>>> d = dict (item = 'unladen swallow')
>>> t.substitute (d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute (d)
'Return the unladen swallow to $ owner.'
```

Template child classes can define their own delimiters. Typically, a batch photo renaming script can choose the percent symbol as a marker for, for example, the current date, the image number, or its format:

```
>>>
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename (Template):
... delimiter = '%'
>>> fmt = input ('Enter rename style (% d-date% n-seqnum% f-format):')
```

```
Enter rename style (% d-date% n-seqnum% f-format): Ashley_% n% f

>>> t = BatchRename (fmt)
>>> date = time.strftime ('% d% b% y')
>>> for i, filename in enumerate (photofiles):
... base, ext = os.path.splitext (filename)
... newname = t.substitute (d = date, n = i, f = ext)
... print ('\'7b0\'7d -> \'7b1\'7d'. format (filename, newname))

img_1074.jpg -> Ashley_0.jpg
img_1076.jpg -> Ashley_1.jpg
img_1077.jpg -> Ashley_2.jpg
```

Another use of the templates is to separate the business logic from the side and the details specific to each output format. It is possible to generate in this way XML files, text, HTML, …

11.3. Work with binary data

The struct module exposes the pack () and unpack () functions for working with binary data. The following example shows how to browse a ZIP file header without using the zipfile module. The markers "H" and "I" represent unsigned integers, stored respectively on two and four bytes. The "<" indicates that they have a standard size and in the little-endian style.

```
struct import

with open ('myfile.zip', 'rb') as f:
    data = f.read ()

start = 0
for i in range (3): # show the first 3 file headers
    start + = 14
    fields = struct.unpack ('<IIIHH', data [start: start + 16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start + = 16
    filename = data [start: start + filenamesize]
    start + = filenamesize
    extra = data [start: start + extra_size]
    print (filename, hex (crc32), comp_size, uncomp_size)

    start + = extra_size + comp_size # skip to the next header
```

11.4. threads

Independent tasks can be run concurrently (using concurrency), using threads. Threads can improve the responsiveness of an application that would interact with the user while other processes are running in the background. Another typical use is to separate on two separate threads I / O (input / output) and calculation.

The following code gives an example of using the threading module running background tasks

while the main program continues to run:

```
import threading, zipfile

class AsyncZip (threading.Thread):
    def __init __ (self, infile, outfile):
        threading.Thread .__ init __ (self)
        self.infile = infile
        self.outfile = outfile

    def run (self):
        f = zipfile.ZipFile (self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write (self.infile)
        f.close ()
        print ('Finished background zip of:', self.infile)

background = AsyncZip ('mydata.txt', 'myarchive.zip')
background.start ()
print ('The main program continues to run in foreground.')

background.join () # Wait for the background
print ('Main program waited until background was done.')
```

The main challenge of multi-tasking applications is coordination between threads that share data or resources. To do this, the threading module exposes some tools dedicated to synchronization such as locks, events, conditional variables, and semaphores.

Although these tools are powerful, design errors can cause problems that are difficult to reproduce. So, the preferred approach for coordinating tasks is to restrict a resource's access to a single thread, and use the queue module to feed this thread of queries coming from other threads. Applications using Queue for their communication and thread coordination are simpler to design, more readable, and more reliable.

11.5. logging

The logging module is a complete logging system. In its most basic use, messages are simply sent to a file or to sys.stderr

```
import logging
logging.debug ('Debugging information')
logging.info ('Informational message')
logging.warning ('Warning: config file% s not found', 'server.conf')
logging.error ('Error occurred')
logging.critical ('Critical error - shutting down')
```

Producing the following display:

```
WARNING: root: Warning: config file server.conf not found
ERROR: root: Error occurred
CRITICAL: root: Critical error - shutting down
```

By default, information and debug messages are ignored, others written to standard output. It is

also possible to send messages by email, datagrams, on sockets, or posted on an HTTP server. The new filters allow you to use different outputs depending on the priority of the message: DEBUG, INFO, WARNING, ERROR, and CRITICAL.

from a configuration file, allowing to customize the log without modifying the application.

11.6. Low references

Python itself manages the memory (by reference count for the majority of objects, and using a garbage collector) to eliminate cycles. The memory is released quickly when its last reference is lost.

This approach works well for most applications, but sometimes it is necessary to monitor an object only during its use by something else. Unfortunately, just following it creates a reference, which makes the object permanent. The weakref module exposes tools for tracking objects without creating a reference. When an object is not used, it is automatically removed from the low reference table, and a callback function is called. A typical example is the cache of expensive objects to create:

```
>>>
>>> import weakref, gc
>>> class A:
... def __init __ (self, value):
... self.value = value
... def __repr __ (self):
... return str (self.value)
...
>>> a = A (10) # create a reference
>>> d = weakref.WeakValueDictionary ()
>>> d ['primary'] = does not create a reference
>>> d ['primary'] # fetch the object if it is still alive
10
>>> del a # remove the one reference
>>> gc.collect () # run garbage collection right away
0
>>> d ['primary'] # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d ['primary'] # entry was automatically removed
  File "C: /python35/lib/weakref.py", line 46, in __getitem__
    o = self.data [key] ()
KeyError: 'primary'
```

11.7. Tools for working with lists

Many data structures can be represented with native lists. However, sometimes other needs emerge, for structures with different characteristics, typically in terms of performance.

The array module provides an array () object, which can store only homogeneous lists, but in a

more compact way. The following example shows a list of numbers, each stored as two unsigned bytes ("H" marker) rather than using 16 bytes as a typical list would have done:

```
>>>
>>> from array import array
>>> a = array ('H', [4000, 10, 700, 22222])
>>> sum (a)
26932
>>> a [1: 3]
array ('H', [10, 700])
```

The collections module provides the class deque (), which looks like a list, but faster to insert or exit items from the left, and slower to access items in the middle. It's objects are particularly suitable for building queues or tree-length algorithm in width:

```
>>>
>>> from collections import deque
>>> d = deque (["task1", "task2", "task3"])
>>> d.append ("task4")
>>> print ("Handling", d.popleft ())
Handling task1
unsearched = deque ([starting_node])
def breadth_first_search (unsearched):
    node = unsearched.popleft ()
    for m in gen_moves (node):
        if is_goal (m):
            return m
        unsearched.append (m)
```

In addition to providing alternative list implementations, the library provides tools such as bisect, a module containing sorted list manipulation functions:

```
>>>
>>> bisect import
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort (scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

The heapq module, which implements heaps from simple lists. The lowest value is always at the first position (index 0). This is useful in cases where the application often needs to find the smallest item without sorting the list entirely:

```
>>>
from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify (data) # rearrange the list into heap order
>>> heappush (data, -5) # add a new entry
>>> [heappop (data) for i in range (3)] # fetch the three smallest entries
[-5, 0, 1]
```

11.8. Floating Point Decimal Arithmetic

The decimal module exposes the Decimal class, which specializes in calculating decimal numbers represented in floating point. Compared to the native float class, it is particularly useful for

applications dealing with finance are other uses requiring exact decimal representation,
control over accuracy,
control over the roundings to correspond to the legal obligations or the regulator,
follow significant decimals, or
applications with which the user expects results identical to the calculations made by hand.
For example, calculating 5% tax on a 70 cent bill gives a different result in binary and decimal floating point numbers. The difference becomes significant when rounding the result to the nearest cent:

```
>>>
>>> from decimal import *
>>> round (Decimal ('0.70') * Decimal ('1.05'), 2)
Decimal ( '0.74')
>>> round (.70 * 1.05, 2)
0.73
```

The result of a calculation given by Decimal preserves the non-significant zero. The class automatically retains four significant digits for operands with two significant decimal places. The Decimal class mimics mathematics as it could be done by hand, avoiding the typical problems of floating-point binary arithmetic that is not able to accurately represent certain decimal quantities.

The exact representation of the Decimal class allows it to do modulus calculations or equality tests that would not be possible with binary floating point:

```
>>>
>>> Decimal ('1.00')% Decimal ('. 10')
Decimal ( '0.00')
>>> 1.00% 0.10
0.09999999999999995

>>> sum ([Decimal ('0.1')] * 10) == Decimal ('1.0')
true
>>> sum ([0.1] * 10) == 1.0
false
```

The decimal module allows to make calculations with as much precision as necessary:

```
>>>
>>> getcontext (). prec = 36
>>> Decimal (1) / Decimal (7)
Decimal ( '0.142857142857142857142857142857142857')
```

12. Virtual Environments and Packages

12.1. Introduction

Python programs often use packages and modules that are not part of the standard library. They also sometimes require a specific version of the library, requiring, for example, that a certain bug has been fixed, or that the program has been implemented using an outdated version of the library interface.

This means that it is not necessarily possible for a Python installation to cover all the needs of all applications. Basically, if an application A depends on version 1.0 of a module and an application B depends on version 2.0, there is conflict, and installing version 1.0 or 2.0 will leave one of the two applications unable to function.

The solution to this problem is to create a virtual environment (often called "virtualenv"), a folder tree containing a standalone Python installation, some version of Python, with additional packages.

Different programs can use different virtual environments. To resolve the previous conflict (conflicting dependencies) application A may have its own virtual environment, with version 1.0 of the installed module, while application B has another virtual environment with version 2.0. If the application B requires an update of the library in version 3.0, it will still not affect the application A.

12.2. Creation of Virtual Environments

The script used to create and manage virtual environments is called pyvenv. pyvenv usually installs the most recent version of Python. There is also a version number suffixed, so that you can choose the version of pyvenv ex running eg pyvenv-3.4 (or any other version).

To create a virtualenv, call the pyvenv program with the name of the folder in which you want to install it:

pyvenv tutorial-env
This will create the tutorial-env folder (if it does not exist) and subfolders containing a copy of the Python interpreter, the standard library, and a few other useful files.

Once the virtualenv is created, you must activate it.

On windows, run:

tutorial-env / Scripts / activate
On Unix and MacOS, run:

source tutorial-env / bin / activate
(This script is written for the bash shell, if you use csh or fish, use the activate.csh or activate.fish variants.)

Enabling a virtualenv changes your shell prompt to tell you which virtualenv you are using, and changes your environment so that when you run python, your shell finds the Python of your virtualenv. For example :

```
-> source ~ / envs / tutorial-env / bin / activate
(tutorial-env) -> python
Python 3.4.3+ (3.4: c7b9645a6f35 +, May 22 2015, 09:31:25)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python34.zip', ...,
'~ / Envs / tutorial-env / lib / python3.4 / site-packages']
>>>
```

12.3. Manage Packages with pip

Once a virtualenv is enabled, you can install, update, or delete packets through a program named pip. By default pip installs its packages from the "Python Package Index", <https: //pypi.python.org/pypi>. You can browse packages by going to PyPI with your browser, or using the search function, basic, pip

```
(tutorial-env) -> pip search astronomy
skyfield - Elegant astronomy for Python
gary - Galactic astronomy and gravitational dynamics.
novas - The US Naval Observatory NOVAS astronomy library
astroobs - Provides astronomy ephemeris to plan telescope observations
PyAstronomy - A collection of astronomy related tools for Python.
...
```

pip has several sub-commands: "search", "install", "uninstall", "freeze", etc ... (See the Python modules installation guide, which contains extensive documentation on pip.)

You can install the latest version of a package just by name:

```
-> pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

It is also possible to install a specific version, by suffixing its name of = and the version number:

```
        -> pip install requests == 2.6.0
Collecting requests == 2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

If screw execute this command again, pip will do nothing: it will be aware that the requested version is already installed. You can provide a different version number, and thus get the package

to the requested version, or run pip install --upgrade to update the package to its most recent version:

```
-> pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

pip uninstall followed by one or more package names will remove them from your virtualenv.

pip show will display information about a given package:

```
(tutorial-env) -> pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: HTTP Python for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
requires:
```

pip list lists the packages installed in the current virtual environment:

```
(tutorial-env) -> pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

pip freeze also produces a list of installed packages, but in a format readable by pip install. It is customary to put this list in a file named requirements.txt

```
          (tutorial-env) -> pip freeze> requirements.txt
(tutorial-env) -> cat requirements.txt
novas == 3.1.1.3
== numpy 1.9.2
requests == 2.7.0
```

The requirements.txt file can be archived in the version control system, and delivered as a part of the application. Users will be able to install the necessary packages with install -r

```
-> pip install -r requirements.txt
Collecting novas == 3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy == 1.9.2 (from -r requirements.txt (line 2))
```

```
  ...
Collecting requests == 2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
pip has many other options, documented in the Python Installation Guide.
```