PYTHON PROGRAMMING

FOR BEGINNERS

HANDS ON PROJECT AND EXAMPLES

LEARN CODING IN 7 DAYS:

CRASH COURSE INTRODUCTION TO PROGRAMMING

J O H N R E E D


PYT
PROGRA
FOR BEC

Python Programming for Beginners

Learn Coding in 7 Days: Crash Course Introduction to Programming | Hands-On Projects and Examples

By John Reed

CODING ACADEMIA

Table of Contents

Introduction

There are many reasons why I think everyone should be able to learn to program:

- It can simplify your daily life, a common cycle for example can be much more powerful than you imagine.
- It can become a job, even if it is well paid.
- It helps you to reason logically, do not underestimate this aspect, we are always surrounded by small problems, and learning to rationalize is a skill you can always rely on.

A computer is a very powerful and at the same time very stupid tool: a practically infinite amount of memory when compared with our brain, a high calculation speed and moreover is not always bored doing the same thing (for a human being it would be instead impossible). What you have to do to use his potential is to learn to "communicate with him", and this can be achieved through the knowledge of programming languages.

To really learn something, you must like it. Learning to programming must become a hobby for you, such as going for a walk or playing the guitar. Initially you will write programs for yourself, to simplify what your current needs are; when you get good enough you can start writing code for someone else, and start working by programming in Python.

In the rest of this book I will teach you all the tools to become a professional programmer. Of course it cannot be an exhaustive book, any book is exaustive and this is even more true in programming. What I can teach you is all the knowledge to be able to walk alone, deepen and become an ever better programmer, and perhaps ... the best, day after day.

To start writing a program, you need an idea, or rather, a "problem". You will need asomething that needs a solution. At that point you start writing code, thinking about each step to get to the solution. You can start thinking sequentially or (as I do) go backwards. Start from the end and step by step analyze all the steps you need until you have reached the starting point.

Once this is done, all you have to do is write it in the form of a code. If you have a little experience in other programming languages you will realize how Python's grammar is much more intuitive. You will find yourself writing code as if you were writing a simple sentence in English, for this reason the learning curve will be very fast. Just like any language we will start by writing small sentences and analyzing them, when you acquire the necessary mastery we will start writing more and more complex programs.

Reading this book, studying it, writing codes with him is only a matter of time before "the light will turn on". This is undoubtedly the best time, you will begin to understand the wonderful world behind programming and you will write codes all day because you want to learn the next step as fast as you can, in short, the computer not obtained for you more secrets, you will be able literally to do anything with it.

CHAPTER 1. Setting Up your Environment

The first thing you need to do with Python is to install it.

If you are using an account on a remote server, your ISP may have already installed Python 3. If you are using Linux at home, you may already have Python 3 in this case too. Most popular GNU / Linux distributions include Python 2 in the own default installation; the number of distributions that also include Python 3 is limited, but steadily increasing. Mac OS X includes a command line version of Python 2, but at the time of writing it does not include Python 3. Microsoft Windows does not include any version of Python. But don't despair! You can pave the way towards installing Python with a mouse click, regardless of which operating system you use.

The easiest way to check if you have Python 3 on your Linux or Mac OS X system is to use the command line. On Linux, look for a program called Terminal in your Applications menu. (It could be in a submenu like Accessories or System Tools.) On Mac OS X, there is an application called Terminal.app in your / Applications / Utilities / folder.

Once you are at the command line prompt, simply type python3 (all in lowercase, no spaces) and see what happens. On my home Linux system Python 3 is already installed, so this command brings me into the interactive Python shell.

```
mark@atlantis:~$ python3
Python 3.0.1+ (r301:69556, Apr 15 2009, 17:25:52)
[GCC 4.3.3] on linux2
Digit "help", "copyright", "credits" o "license" for further informations.
>>>
```

So going back to the question this section started with, "Which Python do you need?" Whatever is already installed on the computer you have.

Installing Python on Windows

Today Windows is distributed for two architectures: 32 bit and 64 bit. Of course, there are many different versions of Windows - XP, Vista, Windows 7 - but Python works on all of them. The most important distinction is between 32 bit and 64 bit. If you have no idea which architecture you are using, it is probably 32-bit.

Visit *python.org/download/* and download the Python 3 for Windows installer appropriate for your architecture. Your choices will look like these:

- Windows Python 3.1 installer (Windows executable - does not include sources)
- Windows Python 3.1 AMD64 installer (Windows AMD64 executable - does not include sources)

I prefer not to provide you with direct links to download installers, because minor Python updates are released at all times and I don't want to be responsible for making you miss any important updates. You should always install the latest version of Python 3.x unless you have some dark reason for not doing so.

Double-click the .msi file when the download is complete. Windows will present you with a security warning while you are about to start an executable file. The official Python installer is digitally signed by the Python Software Foundation, the nonprofit company that oversees the development of Python. Beware of imitations!

Click the Run button to start the Python 3 installer.

First, the installer will ask you if you want to install Python 3 for all users or just you. The default choice is "install for all users" (which is "install for all users"), which is best unless you have a good reason to choose otherwise. (One of the possible reasons why you might prefer "install just for me" is that you are installing Python on your company's computer and you do not have administrative rights on your Windows account. But then why are you installing Python without the administrator's permission Windows of your company? Don't let me have this problem!)

Click the Next button to confirm your choice of installation type.

Next, the installer will ask you to choose a destination directory. The default choice for all versions of Python 3.1.x is C: \ Python31 \, which should be fine for most users unless there is a particular reason for changing it. If you use a separate partition to install applications, you can select it using the built-in controls, or you can simply type the path in the text field below. You don't have to install Python on the C: disk, but you can install it on any disk and in any folder.

Click on the Next button to confirm your choice on the destination directory.

The following screenshot looks complicated, but it really isn't. As with many installers, you have the option of not installing every single Python 3 component. If disk space is limited, you can exclude some components. Ext Register Extensions allows you to double click on Python scripts (.py files) to run them. Recommended but not mandatory. (This option doesn't require disk space, so it doesn't make much sense to rule it out.)

- Tcl / Tk is the graphics library used by Shell Python that you will use throughout this book. I highly recommend keeping this option checked.

- Documentation installs a help file that contains much of the information found on docs.python.org. Recommended if you have a dial-up connection or limited Internet access.
- Utility Scripts includes the 2to3.py script which you will learn about later in this book. This script is necessary if you want to learn how to convert existing Python 2 code to Python 3.

If you don't have existing Python 2 code, you can leave this option out. Suite Test Suite (test series) is a collection of scripts used to test the Python interpreter. We will not use it in this book, nor have I personally used it in Python programming activities. Completely optional.

If you are not sure how much free space you have on disk, click on the Disk Usage button. The installer will list your disks and calculate both how much space is available on each and how much space would remain after installation. Click the OK button to return to the "Customizing Python" screen (ie "Customize Python").

If you decide to exclude an option, click on the button to the left of the option and select "Entire feature will be unavailable" (ie "The entire function will not be available") from the drop-down list that will appear. For example, excluding tests will save the incredible 7908 KB disk space. Click on the Next button to confirm your choice of options.

The installer will copy all the necessary files to the destination directory you have chosen. Click on the Finish button to exit the installer. There should be a new element in your Start menu called Python 3.1.

Inside, there is a program called IDLE. Select this item to launch the interactive Python Shell.

Installing Python on MAC OS X

All modern Macintosh computers use Intel microprocessors (like most Windows PCs). Older Macs use PowerPC microprocessors. You don't need to understand the difference, because there is only one Python installer for all Macs.

Visit *python.org/download/* and download the Mac installer. It will have a name similar to Python 3.1 Mac Installer Disk Image, although the version number may be different. Make sure to download version 3.x, not version 2.x.

Your browser should automatically mount the disk image and open a Finder window to show you its contents. (If this does not happen, you will have to find the disk image in the folder where you downloaded it and double click on it to mount it. Its name should be something like python-3.1.dmg.) The image disk contains a number of text files (Build.txt, License.txt, ReadMe.txt) and the actual installation package, called Python.mpkg.

Double-click on the Python.mpkg installer package to launch the Python installer for Mac.
The first screen of the installer presents you with a brief description of Python, then refers you to the ReadMe.txt file (which you haven't read,

right?) For more details.

Click on the Continue button to continue.

The next page actually contains some important information: Python requires Mac OS X 10.3 or a newer version. If you are still using Mac OS X 10.2, you really should update it. Apple no longer releases security updates for your operating system, and your computer will likely be at risk if you ever go online. In addition, you cannot use Python 3.

Click on the Continue button to advance.

Like all the best installers, the Python installer shows you the software license agreement. Python is open source and its license is approved by the Open Source Initiative. Python has had a number of owners and sponsors throughout its history, each of which has left its mark on the software license. But the end result is this: Python is open source and you can use it

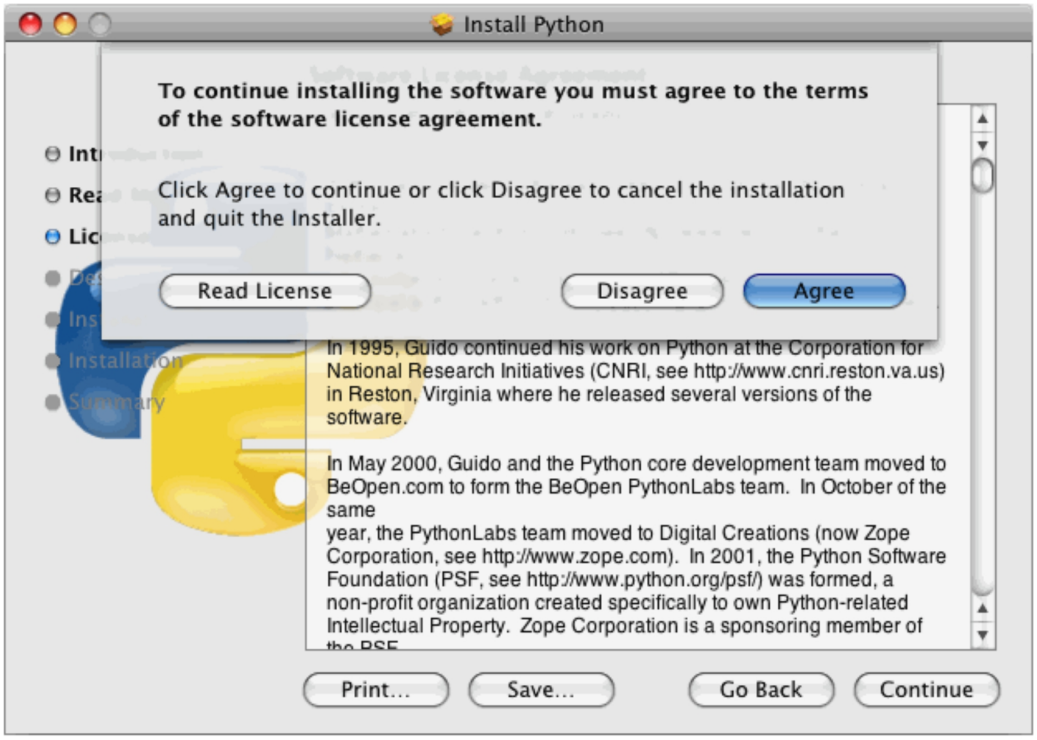
on any platform, for any purpose, without paying anything and without having any reciprocity obligation.

Click on the Continue button once more.

Due to the quibbles of Apple's standard installation framework, you must "accept" the terms of the software license in order to complete the installation.

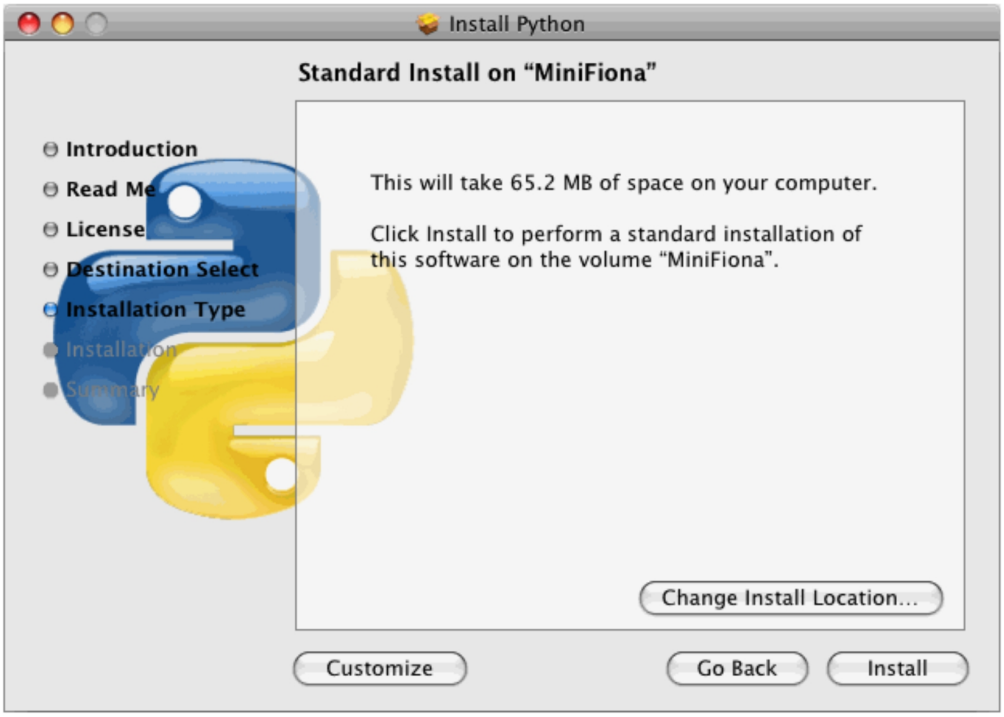
Since Python is open source, you are actually "accepting" that the license grants you additional rights rather than depriving you of them.

Click on the Agree button to continue.



The next screen allows you to change the installation path. You have to install Python on the boot disk, but due to the limitations of the installer this is not imposed. Actually, I never needed to change the installation path.

From this screen you can also customize the installation to exclude some features. If you want to do this, click on the Customize button, otherwise click on the Install button.



If you choose a customized installation, the installation program will show you the following list of features:

- Python Framework (Python framework). This is the heart of Python and is both selected and disabled because it must necessarily be installed. GUI Applications includes IDLE, the graphical Shell Python that you will use throughout this book. I highly recommend keeping this option checked.
- UNIX command-line tools (UNIX command line tools) includes the python3 command line application. I highly recommend keeping this option as well.
- Python Documentation contains much of the information found on docs.python.org. Recommended if you have a dial-up connection or limited Internet access. Shell profile updater checks whether to update your shell profile (used in Terminal.app) to make sure that this version of Python is the one contained in your shell's search path. You probably don't need to change this setting.
- Fix system Python (correction for the system Python interpreter) should not be changed. (Tells your Mac to use Python 3 as the default Python interpreter for all scripts, including Apple's installed system

scripts. This would be a real disaster, as many of those scripts were written for Python 2 and would not run correctly. from Python 3.)

Click on the Install button to continue. Since system frameworks and executables are installed in / usr / local / bin /, the installer will ask you for a password. administration.

There is no way to install Python for Mac without administrative privileges. Click on the OK button to begin the installation. The installer will show a progress indicator while installing the features you have selected.

Assuming everything went well, the installer will show you a large green tick to let you know that the installation was successfully completed. Click on the Close button to exit the installation program. Assuming that you have not changed the installation path, you can find the newly installed files in the Python 3.1 folder inside your / Applications folder.

The most important element is IDLE, the graphical Shell Python. Double click on IDLE to launch the Python Shell.

Using Shell Python

Shell Python is the place where you can explore Python syntax, get interactive help on commands and debug small programs. The Graphical Shell Python (called IDLE) also contains a decent text editor that supports coloring of Python syntax and integrates with the Python Shell. If you don't have your favorite editor yet, you should try using IDLE.

Let's start from the beginning. Shell Python itself is a wonderful interactive playing field. In this book, you will see examples like this:

```
>>> 1 + 1
2
```

The three angle brackets, >>>, denote the Shell Python prompt. That part only serves to let you know that this example is meant to be followed in the Shell Python, so it should not be typed.
1 + 1 is the part you need to type. You can type any valid Python expression or any command in the Python Shell. Don't be shy, it won't bite you! The worst that could happen to you is getting an error message. The commands are executed immediately (as soon as you press ENTER); expressions are evaluated immediately and Shell Python prints the result.
2 is the result of the evaluation of this expression. It turns out that 1 + 1 is a valid Python expression. The result, of course, is 2.
Let's try another one.

```
>>> print('Hello world!')
Hello world!
```

Pretty simple, isn't it? But you can do much more in Shell Python. If you ever get stuck - can't remember a command, or can't remember the appropriate arguments to pass to a certain function - you can get interactive help in Shell Python. Just type in help and press ENTER.

```
>>> help
```

Type help () for interactive help, or help (object) for help on object. There are two ways to help. You can get help on a single object by simply printing the documentation and returning immediately to the Shell Python prompt.

You can also enter the help mode, where instead of evaluating Python expressions, type keywords and command names and get all the known information about those commands printed. To enter interactive help mode, type help () and press ENTER.

```
>>> help()
```

Welcome to Python 3.0! This is the online help mode. If this is your first time using Python, you should definitely give it take a look at the tutorial at http://docs.python.org/tutorial/ .

Type the name of any module, keyword, or topic for get help on writing Python programs and using Python modules. To exit this help mode and

return to the interpreter, vi just type "quit". To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics".

Each module is equipped with a brief summary of its functions; to list the modules whose summaries contain a certain word like "spam", type "modules spam".

```
help>
```

Notice how the prompt changes from >>> to help>. This reminds you that you are in interactive help mode. Now you can enter any keyword, command, module name, function name - just about anything Python can understand - and read the related documentation.

```
help> print print(...)
print(value, ..., sep=' ', end='\n', file=sys.stdout)
```

Print the value on a stream, or on sys.stdout by default.
Optional named arguments:
file: a file-like object (stream); use sys.stdout as default.
sep: string inserted between values, use a space as default value.
end: string added at the bottom after the last value, use a carriage return character as default.

```
help> PapayaWhip
help> quit
```

You are exiting help mode and returning to the Python interpreter.
If you want to ask for help directly on a particular object from within the interpreter, you can type

"Help (object)". Run "help ('string')"

has the same effect as typing a particular string at the help mode prompt.

1. To get the documentation on the print () function, just type print and press ENTER. The interactive help mode will show you something like a manual page: the name of the function, a brief

synopsis, the arguments of the function and their default values, and so on. If the documentation seems obscure to you, don't panic. You will learn more about all these concepts in the next chapters.

2. Of course, the interactive help mode doesn't know everything. If you type something that isn't a Python command, module, function, or other default keyword, interactive help mode will just shrug its virtual shoulders.

3. To exit interactive help mode, type quit and press ENTER.

4. The prompt will return to >>> to let you know that you have exited interactive help mode and returned to Shell Python. IDLE, the Graphical Python Shell, also includes a text editor for Python.

CHAPTER 2. Let’s start programming

As with most programming books, the convention would require you to start with a series of boring chapters on fundamentals and then gradually build something useful. We will spare ourselves this wait. Below you will immediately find a complete and functional Python program. It probably doesn't make any sense to you now, but don't worry because you're going to analyze it line by line. However, try to read it, first of all, to see if you can get anything out of it.

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
" 'Convert the size of a file into a readable form.
```

Arguments with name :

```
size - size of the file in bytes
a_kilobyte_is_1024_bytes - if True (default), use multiples of 1024
if False, use multiples of 1000
Returns: string
" '
if size <0:
raise ValueError ('the number must not be negative')
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
for suffix in SUFFIXES [multiple]:
size / = multiple
if size <multiple:
return '{0: .1f} {1}'. format (size, suffix)
raise ValueError ('number too large')
if __name__ == '__main__':
print (approximate_size (1000000000000, False))
print (approximate_size (1000000000000))
```

Now let's run the program from the command line. On Windows we will have something like this:

```
c:\home\diveintopython3\esempi> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB
```

On Mac OS X or Linux we will have something like this:

```
you@localhost:~/diveintopython3/esempi$ python3 humansize.py
1.0 TB
931.3 GiB
```

What just happened?
You have run your first Python program. You invoked the Python interpreter from the command line and passed the name of the script that you wanted the interpreter to run. The script defines a single function, the approximate_size () function, which takes an exact size of a file in bytes and calculates a more "pleasant" (but approximate) version.

You have probably seen it done in Windows Explorer, or in Mac OS X Finder, or in Nautilus or Dolphin or Thunar on Linux. If you view a folder of documents as a multi-column list, the program will show a table with the icon of each document, the name, size, type, date of the last modification, and so on. If the folder contains a 1093 byte file called TODO, your file manager will not display TODO 1093 bytes, but will show you something similar to TODO 1 KB. This is what the approximate_size () function does.

Look at the bottom of the program and you will see two calls to print (approximate_size (arguments)). These are function invocations - they first call the approximate_size () function by passing it a number of arguments, then take the return value and pass it directly to the print () function. The print () function is predefined, so you will never see its explicit declaration. You can simply use it anytime, anywhere. (There are many predefined functions and many other functions that are divided into modules. Be patient, grasshopper.)

So why does running the script from the command line return you the same results every time? We will get there. First of all, let's take a look at the

approximate_size () function.

Declare Functions

Python allows you to write functions like most other languages, but it doesn't use separate header files like C ++ or interface / implementation sections like Pascal. When you need a function, simply declare it like this:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

The def keyword begins the function declaration, followed by the function name, followed by the function arguments enclosed in parentheses. Topics are separated by commas. Note also that the function does not define a return data type.

Python functions do not specify the data type of their return value, nor do they specify whether or not they return a value. (In fact, each Python function returns a value: if the function executes a return statement, it will return that value, otherwise it will return None, the null value of Python.) In some languages, functions (which return a value) begin with function and procedures (which do not return a value) begin with sub. There are no procedures in Python.

There are only functions, all functions return a value (even if it is None) and all functions begin with def. The approximate_size () function takes two arguments - size and a_kilobyte_is_1024_bytes - but no argument declares its type. In Python, variables are never explicitly typed. The Python interpreter understands for itself what the type of a variable is and tracks it internally. In Java and other statically typed languages, you must declare the type of the return value and each argument of the function. In Python, types are never declared. Based on the value that is assigned, Python keeps track of the data type internally.

Optional and Named Arguments

Python allows function arguments to have default values, so that if a function is called without an argument, that argument is set to its default

value. In addition, arguments can be specified in any order using named arguments. Let's take another look at that declaration of the approximate_size () function:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

The second argument, a_kilobyte_is_1024_bytes, specifies a default value of True. This means that the argument is optional: you can invoke the function without it and Python will operate as if you had invoked it by passing True as the second parameter.

Now look at the bottom of the script:

```
if __name__ == '__main__': print(approximate_size(1000000000000, False)) ①
print(approximate_size(1000000000000)) ②
```

1. This line calls the approximate_size () function with two arguments. Within the approximate_size () function, a_kilobyte_is_1024_bytes will be worth False, since you explicitly passed False as the second argument.
2. This line calls the approximate_size () function with a single argument. But that's still fine, because the second argument is optional! Since the caller does not specify it, the second argument will take its default value True, as set out in the function declaration.

The values of the arguments of a function can also be passed by name.

```
>>> from humansize import approximate_size >>> approximate_size(4000,
a_kilobyte_is_1024_bytes=False) ①
'4.0 KB' >>> approximate_size(size=4000, a_kilobyte_is_1024_bytes=False) ②
'4.0 KB' >>> approximate_size(a_kilobyte_is_1024_bytes=False, size=4000) ③
'4.0 KB' >>> approximate_size(a_kilobyte_is_1024_bytes=False, 4000) ④
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg >>> approximate_size(size=4000, False) ⑤
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

1. This is an invocation of the approximate_size () function with 4000 as the value for the first argument (size) and False as the value for the second argument called a_kilobyte_is_1024_bytes. (This actually happens to be the second argument, but it doesn't matter, as you'll see in a minute.)

2. This is an invocation of the approximate_size () function with 4000 as the value for the argument called size and False as the value for the argument called a_kilobyte_is_1024_bytes. (As it happens, these named arguments are in the same order as the arguments in the function declaration are listed, but that doesn't matter either.)

3. This is an invocation of the approximate_size () function with False as the value for the argument called a_kilobyte_is_1024_bytes and 4000 as the value for the argument called size. (See? I told you that the order doesn't matter.)

4. This invocation fails, because you pass a named argument followed by an unnamed (positional type) argument, and this never works. Reading the list of topics from left to right, once you use a single named topic, the rest of the topics must also be of the same type.

5. This invocation also fails, for the same reason as the previous one. Are you surprised? After all, you passed 4000 as the value for the argument called size, then "obviously" you meant to pass that False value to the argument a_kilobyte_is_1024_bytes. But Python doesn't work this way. As soon as you use a named argument, all the arguments to the right of that argument must also be named arguments.

Write Readable Code

I won't bore you with a long sermon on the importance of documenting your code. Suffice it to say that the code is written only once but read many times, and that the most important readers of your code are you, six months after writing it (that is, after you have forgotten everything but need to

correct something). Python makes it easier to write readable code, so take advantage of it. In six months you will thank me.

Documentation Strings

You can document a Python function by adding a documentation string (called docstring for brevity). In our program, the approximate_size () function has been equipped with a docstring:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
" 'Convert the size of a file into a readable form
```

.
Arguments with name:

```
size - size of the file in bytes
a_kilobyte_is_1024_bytes –
if True (default), use multiples of 1024
if False, use multiples of 1000 Returns: string
'"
```

Triple quotes are used to represent a string on multiple lines. Everything between the start and end quotes is part of a single string, including carriage returns, whitespace at the beginning of a line, and other quote characters. Triple quotes can be used anywhere, but you will see them exploited especially in the definition of a docstring.

Triple quotes are also an easy way to define a string that contains both quotes and quotes, such as qq /.../ in Perl 5.

Everything between the triple quotes represents the function's docstring, which documents what the function does. A docstring, if it exists, must be the first thing defined in a function (that is, in the line immediately after the function declaration). You don't technically need to equip your function with a docstring, but you should always do it. I know you've heard it in every programming course you've followed, but Python gives you an added incentive: docstring is available at runtime in the form of a function attribute.

Many IDEs for Python use docstrings to provide context-sensitive documentation, so that when you write the name of a function, its docstring appears as a suggestion. This feature can be incredibly useful, but it is only as useful as the docstrings you write.

The Research Path of Import

Before continuing, I want to briefly mention the search path for the libraries. Python looks in several places when you try to import a module. Specifically, look in all directories defined in sys.path. This attribute is simply a list and you can easily view or edit it with the standard list methods. (You will learn more about lists in the chapter on native data types.)

```
>>> import sys
>>> sys.path
['',
'/usr/lib/python31.zip',
'/usr/lib/python3.1',
'/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
'/usr/lib/python3.1/lib-dynload',
'/usr/lib/python3.1/dist-packages',
'/usr/local/lib/python3.1/dist-packages'] >>> sys
<module 'sys' (built-in)> >>> sys.path.insert(0, '/home/mark/diveintopython3/esempi')
>>> sys.path
['/home/mark/diveintopython3/esempi',
'',
'/usr/lib/python31.zip',
'/usr/lib/python3.1',
'/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
'/usr/lib/python3.1/lib-dynload',
'/usr/lib/python3.1/dist-packages',
'/usr/local/lib/python3.1/dist-packages']
```

1. Importing the sys module makes all its functions and attributes available.

2. sys.path is a list of directory names that make up the current search path. (Yours will look different, depending on your operating system, the version of Python you are running and where it was originally installed.) Python will search through these directories (in this order) for a file with a .py extension whose name matches what you are trying to import.

3. In fact, I lied; the truth is more complicated than that, because not all modules are stored as .py files. Some are built-in modules, integrated directly within the Python interpreter. Built-in modules behave exactly like other modules, but their Python source code is not available, because they are not written in Python! (Like the language interpreter himself, these modules are written in C.)

4. You can add a new directory to the Python search path at runtime by adding the directory name to sys.path; subsequently, Python will also look in that directory every time you try to import a module. The effect lasts as long as the Python interpreter is running.

5. Using sys.path.insert (0, new_path), you have entered a new directory as the first element of the sys.path list, then at the beginning of the Python search path. This is almost always what you want. In case of a name conflict (for example, in case the Python interpreter includes version 2 of a particular library but you want to use version 3), this ensures that your modules will be found and used instead of the modules distributed together with the Python interpreter.

Everything is an Object

In case you missed it, I just said that Python functions have attributes and that these attributes are available at runtime. A function, like everything else in Python, is an object.
Launch the interactive Python shell and follow me through this series of instructions:

```
>>> import humansize
>>> print(humansize.approximate_size(4096, True))
4.0 KiB >>> print(humansize.approximate_size.__doc__)
```

Convert the size of a file into a readable form. Arguments with name:

```
size -- dimensione del file in byte
a_kilobyte_is_1024_bytes -- se True (default), usa multipli di 1024
if False, use multiples of 1000
```

Returns: string

1. The first line imports the humansize program as a module - a piece of code that you can use interactively or from a larger Python program. Once a module is imported, you can refer to any function, class, or public attribute belonging to the module. Modules can take advantage of this feature to access the functionality of other modules, and you can also do it in the interactive Python shell. This is an important concept, and you will see it repeated several times later in this book.

2. When you want to use functions defined in imported modules, you must include the module name. So you can't just say approximate_size, but you have to write humansize.approximate_size. If you have used classes in Java, the syntax should be vaguely familiar.

3. Instead of invoking the function, you requested one of its attributes, __doc__. import in Python is like require in Perl. Once you have imported a Python module via import, you can access its functions with the modulo.function syntax; once you have requested a Perl module via require, you can access its functions with the module :: function syntax.

What is an Object?

Everything in Python is an object, and everything can have attributes and methods. All functions have a built-in __doc__ attribute, which returns the docstring defined in the source code of the function. The sys module is an object that has (among other things) an attribute called path. And so on.

However, this does not answer the fundamental question: what is an object? Different programming languages define "object" in different ways. In some, it means that all objects must have attributes and methods; in others, it means that all objects are extensible. In Python, the definition is looser. Some objects have no attributes or methods, but they may have them. Not all objects are extensible. But everything is an object in the sense that it can be assigned to a variable or passed as an argument to a function.

You may have heard the term "first class object" in other programming contexts. In Python, functions are first class objects. You can pass a function as an argument to another function. Modules are first class objects. You can pass an entire module as an argument to a function. Classes are first-class objects, and individual instances of a class are also first-class objects.

This concept is important, so I'll repeat it in case you missed it the first few times: everything in Python is an object. Streaks are objects. Lists are objects. Functions are objects. Classes are objects. Instances of a class are objects. Even modules are objects.

Indent the Code

Python functions neither explicitly begin with begin nor end with end, and there are no curly braces to indicate where the function code begins and ends. The colon (:) and the indentation of the code are the only delimiters.

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
if size < 0:
raise ValueError(‘number should not be negative’)
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000 for suffix in SUFFIXES[multiple]:
size /= multiple
if size < multiple:
return '{0:.1f} {1}'.format(size, suffix)
raise ValueError('number is too big')
```

1. Blocks of code are defined by their indentation. By "code block" I mean functions, if statements, for loops, while loops, and so on. The presence of an indentation begins a block and its absence ends it. There are no curly braces, square brackets, or keywords. This means that white spaces are significant and must be consistent. In this example, the function code is

indented by four spaces. There need not be four spaces, there is only need for the indentation to be consistent. The first line that is not indented indicates the end of the function.
2. In Python, an if statement is followed by a block of code. If the if expression is evaluated as true, the indented block is executed, otherwise it falls back into the else block (if it exists). Notice the lack of parentheses around the expression.
3. This line is inside the if code block. The raise instruction will raise an exception (of the ValueError type), but only if size <0.
4. This is not the end of the function. Completely empty lines are not considered. They can make the code more readable, but are not used to delimit blocks. The function continues on the following line.
5. The for loop also signals the start of a block of code. Code blocks can contain multiple lines, as long as they are indented by the same amount. This for loop contains three lines of code. There is no other special syntax for multi-line blocks of code. Indent and move on with your life.

After some initial protests and several mischievous analogies to Fortran, you will reconcile with this feature of Python and you will begin to see its positive sides. One of the biggest benefits is that all Python programs appear similar, in that indentation is a requirement of language and not a matter of style. This makes it easier to read and understand Python code written by other people. Python uses carriage returns to separate instructions and colons and indentation to separate blocks of code. C ++ and Java use semicolons to separate statements and braces to separate blocks of code.

Exceptions

What is an exception? It is usually a mistake, an indication that something has gone wrong. (Not all exceptions are errors, but don't worry about this for now.) Some programming languages encourage the use of return error codes, which are checked. Python encourages the use of exceptions, which are handled.

When an error occurs in the Shell Python, some details are printed on the exception and how it occurred, and that's all. This is called an unhandled

exception. When the exception was raised, there was no instruction to explicitly notice and address it, so it bubbled up to the level of Shell Python, which spits out some debug information and considers its job finished. In the shell this is not a big problem, but if it happened while your Python program is running, the whole program would crash if the exception is not handled. Maybe this is the behavior you want, maybe it isn't.

Unlike Java, Python functions do not declare which exceptions they might raise. It's up to you to determine what possible exceptions you need to catch. An exception does not necessarily have to result in a complete block of the program, however. Exceptions can be handled. Sometimes an exception appears because your code actually has a bug (like trying to access a variable that doesn't exist), but other times an exception is something you can anticipate.

If you are opening a file, that file may not exist. If you are importing a module, that module may not be installed. If you are opening a connection to a database, that database may not be available, or you may not have the correct security credentials to log in. If you know that a line of code could raise an exception, you should handle the exception using a try ... except block. Python uses try ... except blocks to handle exceptions and the raise statement to throw them. Java and C ++ use try ... catch blocks to handle exceptions and the throw statement to throw them.

The approximate_size () function raises exceptions in two different cases: if the size variable passed contains a larger size than the function is designed to handle, or if it contains a size smaller than zero.

```
if size < 0:
raise ValueError('number should not be negative)
```

The syntax for raising an exception is quite simple. Use the raise statement, followed by the name of the exception and a string containing a message for debugging purposes. The syntax recalls that of the invocation of a function. (In fact, exceptions are implemented as classes and this raise statement actually creates an instance of the ValueError class by passing the string 'the number must not be negative' to its initialization method. But we are putting the cart ahead of the oxen!)

You don't need to manage an exception in the function that raises it. If a function does not handle an exception, the exception is passed to the calling function, then to the function that called that function, and so on "to the top of the execution stack". If the exception is never handled, your program will crash, Python will print a "stack trace" on the error channel and things will end there. Again, perhaps this is the behavior you desire; it depends on what your program does.

Capturing Import Errors

One of Python's default exceptions is ImportError, which is raised when you try to import a module and the operation fails. This can happen for a variety of reasons, but the simplest case is where the form does not exist in your import search path. You can use this exception to include optional features in your program. For example, the chardet library provides automatic recognition of character encodings. Maybe your program wants to use this library if it exists, but continue normally if the user has not installed it. You can do this with a try..except block.

```
try:
import chardet
except ImportError:
chardet = None
```

Later, you can check for the presence of the chardet module with a simple if statement:

```
if chardet:
# do something
else:
# continue anyway
```

The ImportError exception is commonly used even when two modules implement a common API, but one is more desirable than the other because maybe it is faster or uses less memory. You can try to import a module but fall back on a different module if the first import fails. For example, the chapter on XML talks about two modules that implement a common API called ElementTree. The first, lxml, is a third-party module that you need to download and install yourself. The second, xml.etree.ElementTree, is slower but is part of the standard Python 3 library.

```
try:
from lxml import etree
except ImportError:
import xml.etree.ElementTree as etree
```

At the end of this try..except block you have imported some module and called it etree. Since both modules implement a common API, the rest of your program doesn't have to check every time which module was imported. And since the module that was imported is always called etree, there is no need to spread if statements anywhere in the rest of your program to call modules with different names.

Unbinded Variables

Take another look at this line of code in the approximate_size () function:

```
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
```

Never declare the multiple variable, but simply assign it a value. This is fine, because Python allows you to do it. What Python doesn't allow you is to refer to a variable that has never been assigned a value. The attempt to do this will raise a NameError exception.

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

One day you will thank Python for this.

Everything is Case Sensitive

All names in Python are case sensitive: variable names, function names, class names, module names, exception names. If you can get it, set it, invoke it, build it, import it, or lift it, then it's case sensitive.

```
>>> an_integer = 1
>>> an_integer
```

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

And so on.

Run the Scripts

Python modules are objects and have several useful attributes. You can take advantage of this feature to easily test your modules as you write them, including a special block of code that runs when you launch the Python file from the command line. Consider the last few lines of humansize.py:

```
if __name__ == '__main__':
print(approximate_size(1000000000000, False))
print(approximate_size(1000000000000))
```

Like C, Python uses == for comparison and = for assignment. Unlike C, Python does not support online assignment, so there is no possibility of accidentally assigning the value you thought you were comparing. What makes this if statement special? Well, modules are objects, and all modules have the built-in __name__ attribute. The value of a module's __name__ attribute depends on how you are using the module. If you import a module, then __name__ will be the name of the module file, without the path and extension.

```
>>> import humansize
>>> humansize.__name__
'humansize'
```

But you can also run the module directly as a separate program, in which case __name__ will be the special default value __main__. In our example, Python will evaluate the if statement, find a true expression and execute the corresponding block of code. In this case, to print two values.

```
1.0 TB
931.3 GiB
```

And this is your first Python program!

CHAPTER 3. Variables, expressions and instructions

Values and Types

A value, such as a letter or a number, is a basic element that allows a program to function. The values we have seen so far are 1, 2 and "Hello, World!". These values belong to different types: 2 is an integer while "Hello, World!" it is a string, so called because it is composed of a "string" of letters. You (together with the interpreter) can identify the strings because they are enclosed in quotation marks. The print statement also works with integers, the interpreter can be started using the python command.

```
python >>> print(4)
 4
```

If you are not sure which type a specific value belongs to, you can consult the interpreter.

```
>>> type('Hello, World!') <class 'str'>
>>> type(17) <class 'int'>
```

It is not surprising that the strings belong to the str type and the integers belong to the int type. It is less obvious that numbers with a decimal point belong to the type called float. This is due to the fact that these numbers are represented in a format called floating point.

```
>>> type(3.2)
<class 'float'>
```

And what about values like "17" and "3.2"? While looking like numbers are enclosed in quotation marks like strings.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
 <class 'str'>
```

In reality they are real strings. When entering a large integer, such as "1,000,000", you may be tempted to use commas in groups of three digits. This is not an integer allowed in Python, although it appears to be spelled correctly:

```
>>> print(1,000,000)
1 0 0
```

Well, that's not what we expected at all! Python interprets 1,000,000 as if it were a sequence of integers separated by commas, which it displays by inserting a space at the comma. This is the first example of a semantic error that we see: the code is executed without displaying error messages but it does not do the operation for which we thought we had designed it.

Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name assigned to a value. Through an assignment instruction you have the ability to create new variables and assign them a value:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

In this example, three assignments are made: the first assigns a string to a new variable called message, the second assigns the integer 17 to the variable n, the third assigns the (approximate) value of π to pi. You can use the print statement to view the value of a variable:

```
>>> print(n)
 17
>>> print(pi)
3.141592653589793
```

The type of a variable is the type of value to which it is attached.

```
>>> type(message)
<class 'str'
> >>> type(n)
<class 'int'>
>>> type(pi)
```

```
<class 'float'>
```

Variable names and keywords

Normally programmers choose names for their variables that are meaningful and document the use of the variable. These names, of arbitrary length, can contain both letters and numbers but cannot begin with a number. Even if capital letters can be used, it is preferable to start variable names with a lower case letter (we will see why later). The underscore (_) character appears in the name of a variable and is often used in multi-word names, such as my_name or airspeed_of_unladen_swallow. Variable names can begin with the underscore character, but generally we avoid doing so unless the code we are writing is contained in libraries that will be used by others. If you assign an invalid name to a variable, you will cause a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

The name 76trombones is not valid because it starts with a number, more @, instead, because it contains the illegal character '@'. But what's wrong with class? It turns out that class is one of Python's reserved words. The interpreter uses these words to recognize the structure of the program, therefore they cannot be assigned to variables. Python reserves 33 keywords:

and del from None True as elif global nonlocal try assert else if not while break except import or with class False in pass yield continue finally is raise def for lambda return

It is better to keep this list handy: if the interpreter complains about the name of one of your variables and you do not understand why, check if it is present in the list.

Instructions

The instruction is a minimum unit of code that can be executed by the Python interpreter. We have seen two types of instructions: Print used both as an expression instruction and as an assignment. When you type an instruction in interactive mode, the interpreter executes it and, if it exists, displays the result. Generally a script contains a sequence of instructions. If there is more than one instruction, the results are displayed one at a time as the instructions are executed. Eg:

```
print(1)
x = 2
print(x)
produce the output
1
2
```

Remember that assignment statements do not produce output.

Operators and operands

Operators are special symbols that represent calculations such as addition and multiplication. The values to which the operator is applied are called operands. The operators +, -, *, /, and ** perform the addition, subtraction, multiplication, division, and exponentiation operations respectively, as illustrated in the following examples:

```
20+32 hour-1 hour*60+minute minute/60 5**2 (5+9)*(15-7)
```

In the transition between Python 2.x and Python 3.x the division operator has been changed: now the result is represented in floating point:

```
>>> minute = 59
 >>> minute/60
0.9833333333333333
```

In Python 2.x the operator would have divided the two integers and truncated the result by returning an integer value:

```
>>> minute = 59
>>> minute/60
0
```

To achieve the same result in Python 3.x, the integer division (// integer) must be used.

```
>>> minute = 59
>>> minute//60
0
```

In Python 3.x the integer division works much more similarly to what you would expect if you entered the expression on a calculator.

Expressions

The expression is a combination of values, variables and operators. Both a single value or a variable are considered an expression. Therefore the following are all correct expressions (provided that a value has been assigned to the variable x)

```
17
X
x + 17
```

If you type an expression in interactive mode, the interpreter will calculate it and display the result:

```
>>> 1 + 1
2
```

One thing that often creates confusion among beginners is an expression present in a script, alone does not return any results. Exercise 1: Type the following instructions into the Python interpreter and observe their behavior:

```
5
x = 5
x + 1
```

Order of Operations

When more than one operator appears in an expression, the calculation order is governed by the rules of precedence. As for mathematical operators, Python follows mathematical conventions. The acronym PEMDAS is useful for remembering the rules:

- Parentheses: they have the highest precedence and can be used to force the computer to perform an expression in the desired order. Since the expressions in parentheses are evaluated first: 2 * (3-1) is equal to 4 and (1 + 1) ** (5-2) to 8. You can also use parentheses to make the expression more readable, as in (minute * 100) / 60, without this changing the result.
- Power raising has the immediately following precedence level, therefore 2 ** 1 + 1 is equal to 3 and not 4, and 3 * 1 ** 3 is equal to 3 and not 27. Multiplications and Divisions have the same precedence, higher than Additions and Subtractions, which have the same level of precedence. So 2 * 3-1 will result in 5, not 4, and 6 + 4/2 will give 8.0 and not 5.
- Operators with the same precedence are calculated from left to right: The expression 5-3-1 has the result 1 and not 3, because 5-3 is carried out first and only then is subtracted 1 by 2. In case of doubt, always use parentheses in your expressions, to make sure the calculations are performed in the desired order.

Module operator

The module operator is applied to whole numbers and results in the remainder of when the first operand is divided by the second. In Python, the modulo operator is the percentage character (%). The syntax is the same as for the other operators:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

So 7 divided by 3 is 2 with the remainder of 1. The modulo operator can be surprisingly useful. For example, it is possible to check whether one number is divisible by another: if x% y is zero, then x is divisible by y. It is also possible to obtain the value of the rightmost digit or digits from a number. For example, x% 10 returns the rightmost digit of x (in base 10). Similarly, x% 100 returns the last two digits.

Working with strings

The operator + works with strings, without being an addition in a strictly mathematical sense. Instead it performs a concatenation: joins the strings together by connecting the second after the last character of the first. For example:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
>>> second = '150'
 >>> print(first + second)
100150
```

The output of this script is 100150. The * operator also works with strings by multiplying the contents of a string by an integer. For example:

```
>>> first = 'Test '
>>> second = 3
>>> print(first * second)
Test Test Test
```

Ask the user for an input value

Sometimes it may be necessary to ask the user to enter the value of a variable. Python has a function called input that can receive keyboard input1. When using this function, the program stops waiting for the user to type something. When the user presses Enter or Enter, program execution resumes and input returns as string what the user has entered.

```
>>> inp = input()
Some silly stuff
>>> print(inp)
Some silly stuff
```

Before receiving input from the user, it is usually a good idea to display a prompt that informs the user what to enter. It is possible to indicate a string to the input function a ché until it is displayed before the input waiting pause:

```
>>> name = input('What is your name?\n')
What is your name?
```

```
Chuck
>>> print(name)
Chuck
```

The sequence \ n at the end of the prompt indicates a carriage return and is a special character that causes the line to be interrupted. That's why user input appears below the prompt. If you expect the user to type an integer, you can try converting the return value to int using the int () function.

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
 17
>>> int(speed)
17
>>> int(speed) + 5
22
```

But if the user enters something other than a string consisting of digits, an error message is displayed:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

We will see later how to handle this type of error.

Comments

As programs get bigger and more complex, they become more difficult to read. Formal languages are condensed and it is often difficult to examine a piece of code and understand what or why it does something. For this reason, it is a good idea to add some notes to your programs to explain in natural language what the program is doing. These notes, which in Python begin with the # symbol, are called comments:

```
# compute the percentage of the hour that has elapsed
 percentage = (minute * 100) / 60
```

In this case, the comment appears alone on one line. You can also insert comments at the end of a line:

```
percentage = (minute * 100) / 60 # percentage of an hour
```

Anything between "#" and the end of the line is ignored and has no e etto ect on the program. Comments are even more useful when documenting not so obvious code features. While it is reasonable to assume that code readers can understand what this does, it is far more useful to explain why. This code comment is superfluous and unnecessary:

```
v = 5 # assign 5 to v
```

This comment, however, contains useful information that is not included in the code:

```
v = 5 # velocity in meters/second.
```

Assigning suitable names to variables can reduce the need for comments, but long names can make complex expressions difficult to read, so there is a tradeoff.

Choose mnemonic variable names

As long as you follow the simple rules of naming the variables avoiding reserved words, you will have ample freedom of action in choosing their name. At first, this choice can be confusing when you read or write a program. For example, the following three scripts, although identical in terms of functionality, seem very different when you read them and try to understand how they work.

```
a = 35.0
b = 12.50
c = a * b
print(c)
hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

Python interprets all three in exactly the same way as we humans read and understand these programs quite differently. The purpose of the second

example is quickly understood since the developer has chosen the name of the variables thinking about what data will be stored in each of them. We call these wisely chosen variable names "mnemonic variable names".

The word mnemonic2 literally means "memory aid". We choose mnemonic names for variables mainly to help us remember why we created them. While all of this sounds great, the names of mnemonic variables can hinder the ability of a novice developer to analyze and understand a code.

This is because novice developers have not yet memorized the reserved words (there are only 33) and sometimes variables with too descriptive names begin to seem part of the language and not just well-chosen variable names. Take a quick look at the following Python sample code which runs repeatedly (looped) on some data. We'll get back to the loops soon, but for now, try to understand what it means:

```
for word in words:
print(word)
```

What is happening? Which words (for, word, in, etc.) are reserved words and which are only variables? Can Python understand the notion of words fundamentally? Beginner programmers have a hard time distinguishing which parts of the code should remain unchanged, like this example, and which parts of the code are simply choices made by the developer. The following code is equivalent to the previous one:

```
for slice in pizza: print(slice)
```

It is easier for the novice programmer to read this script to know which parts are reserved words of Python and which parts are simply variable. It is quite clear that Python does not know the notion of pizza and slice (slice) and that a pizza is made up of a set of one or more slices. But if our program is really about reading data and searching for words in the data, pizza and slices are by no means mnemonic variable names. Choosing them as variable names distracts us from the purpose of the program.

After a short period of time, you will know the most common reserved words and they will begin to catch your eye: The parts of the code de fi ned by Python (for, in, print, and :) are bold, contrary to the variables chosen by

the developer (word and words). Many text editors recognize Python syntax and color reserved words differently by highlighting them with respect to variables. After a little practice in reading Python you will be able to quickly understand which is a variable and which is a reserved word.

Debug

At this point, the most likely syntax error you can make is choosing an illegal variable name, such as class and yield, which are keywords, or odd ~ job eUS $, which contain illegal characters. If you enter a space in the variable name, Python will assume that they are two operands without an operator:

```
>>> bad name = 5
SyntaxError: invalid syntax
>>> month = 09
File "<stdin>", line 1
month = 09 ^
SyntaxError: invalid token
```

As for the syntax errors, it must be said that the error messages are not of great help. The most common messages, SyntaxError: invalid syntax and SyntaxError: invalid token, are both not very explanatory. The most common runtime error is "use before def;": that is, the use of a variable before assigning it a value. This can happen if a variable is written incorrectly:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Variable names are case-sensitive: LaTeX is not the same as latex. At this point, the most likely cause of a semantic error is the order of operations. For example, to calculate 1 / 2π, you may be tempted to write:

```
>>> 1.0 / 2.0 * pi
```

Since the division is done first, you will get π / 2 which is not the same thing! Since Python has no way of knowing what I really wanted to write, you will not receive an error message but only an incorrect result.

Conditional execution

Boolean expressions

A Boolean expression is an expression that can only take on true or false values. In the following examples the == operator is used for the comparison of two operands and will produce True if they are the same or False otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
{}
```

True and False are not considered strings but are special values belonging to the bool type:

```
>>> type(True)
<class 'bool'>
>>> type(False)
 <class 'bool'>
```

The == operator is one of the comparison operators; the others are:

```
x != y # x is not equal to y
x > y # x is greater than y
x < y # x is less than y
x >= y # x is greater than or equal to y
x <= y # x is less than or equal to y
x is y # x is the same as y
x is not y # x is not the same as y
```

Although these operations are likely to be familiar to you, the symbols used by Python are different from the mathematical ones. A common mistake is to use a single equal sign (=) instead of a double equal sign (==): Remember that = assigns a value while == compares two values. There is currently no operator similar to = <or =>.

Logical operators

There are three logical operators: and, or and not. The semantics (meaning) of these operators is similar to their meaning in the English language. For example: x> 0 and x <10 is TRUE only if x is greater than 0 and less than 10. n% 2 == 0 or n% 3 == 0 is TRUE if one of the conditions occurs: that is, if the number it is divisible by 2 or 3. Finally, the not operator negates a Boolean expression, so not (x> y) is true if x> y is false; that is, if x is less than or equal to y. In practice, the operands of the logical operators should be Boolean expressions, even Python is not very rigorous in this regard. Any non-zero number is interpreted as "true":

```
>>> 17 and True
True
```

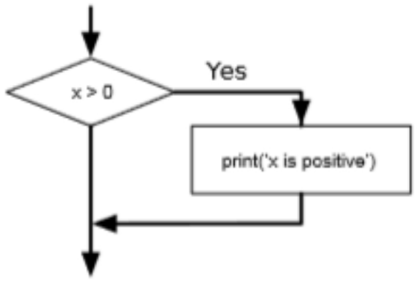
While this flexibility may come in handy, there are a few details that could be confusing. Unless you know what you are doing, you should avoid such an event.

Conditional execution

To write useful programs, we almost always need to check the conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the if statement:

```
if x > 0 :
print('x is positive')
```

The Boolean expression after the if statement is called a condition. At the end of the if statement, the colon must be placed (:), the following line / lines of code must be indented (indented).



If statements have the same structure as function definitions or for1 loops. The instruction consists of a header row ending with a colon (:) followed by an indented block. Instructions like this are called compound because they span multiple lines. While there is no limit to the number of instructions that can appear in the block, at least one must be present. For example, it is useful to have a block without instructions (always as a placeholder for code not yet written). In that case, you can use the pass statement, which does not produce any effect ect.

```
if x < 0 :
pass # need to handle negative values!
```

If you enter an if statement in the Python interpreter, the prompt will change from three chevron to three points to signal that you are in the middle of a block of instructions, as shown below:

```
>>> x = 3
>>> if x < 10:
... print('Small')
 ... Small
>>>
```

When using the Python interpreter, it is necessary to leave an empty line at the end of a block, otherwise Python will return an error:

```
>>> x = 3
>>> if x < 10:
... print('Small')
... print('Done')
File "<stdin>", line 3 print('Done')


^ SyntaxError: invalid syntax
```

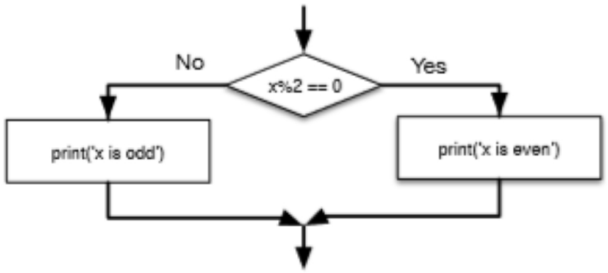
When writing and executing a script, an empty line is not necessary at the end of a block of instructions, although this can improve the readability of the code.

Alternative execution

A second form of the if statement is the alternative execution in which two possibilities have been foreseen and the condition determines which should be performed. The syntax looks like this:

```
if x%2 == 0 :
print('x is even')
 else :
print('x is odd')
```

In the operation x divided by 2, if the rest is 0, we know that x is even and the program displays a message to that effect. If the condition is false, the second set of instructions is executed.



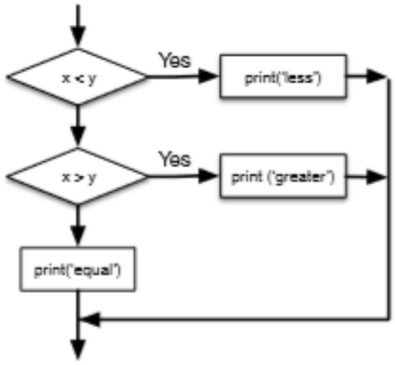
Chained Conditions

Sometimes it is necessary to foresee more than two possibilities and consequently it is necessary to insert more than two branches. One way to express such a calculation is to use chained conditions:

```
if x < y:
print('x is less than y')
elif x > y:
print('x is greater than y')
else:
print('x and y are equal')
```

elif is an abbreviation for "else if". Again, only one branch will run. There is no limit to the number of elif instructions. Various else conditions can be used, as long as they are inserted at the end.

```
if choice == 'a':
print('Bad guess')
elif choice == 'b':
print('Good guess')
elif choice == 'c':
print('Close, but not correct')
```



Exception handling using Try and Except

Previously we examined a segment of code in which we used the input and int functions to read and analyze an integer entered by the user. We also saw that this could be insidious:

```
>>> prompt = "What...is the airspeed velocity of an unladen swallow?\n"
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
 >>>
```

When the Python interpreter executes these instructions, a new prompt will appear, as if Python had thought "oops" and had moved on to the next instruction. However, if you insert this code into a Python script and this error occurs, the script immediately stops displaying a traceback, refusing to execute the next statement. This is an example script to convert a temperature from Fahrenheit to Celsius:

```
inp = input('Enter Fahrenheit Temperature: ')
```

```
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
 print(cel)
```

If we run this code and supply it with invalid values, it will simply fail by showing an unfriendly error message:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.22222222222222


python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
File "fahren.py", line 2, in <module>
fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

To handle these types of expected and unexpected errors, there is a conditional function built into Python called "try / except". The purpose of try and except is related to knowing in advance that some sequences of instructions may have problems. Therefore you can add some alternative instructions to be executed in case of error. These additional statements (except block) will be ignored if no errors occur. You might think of the try and except function in Python as an "insurance policy" on a sequence of statements. For example, we can rewrite the code of our temperature converter as follows:

```
inp = input('Enter Fahrenheit Temperature:')
try:
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)
except:
print('Please enter a number')
```

After inserting a value, Python executes the sequence of instructions in the try block. If no errors are found, skip the except block and proceed. If an exception occurs in the try block, Python blocks the try run and executes the instructions in the except block.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222


python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

Handling an exception with a try statement is called "catching an exception". In this example, the condition except shows an error message. In general, catching an exception gives you the opportunity to solve the problem or try again, or at least end the program elegantly.

CHAPTER 4. Functions

Functions are a very useful tool in Python (and in general in any programming language). Splitting a program into functions may seem unnecessary effort, but instead it is very beneficial. Focus on the following points, which expose the ventures in dividing a program into functions:

- A function allows you to have a set of instructions, in this way the program is much more readable (never underestimate this aspect, especially if it is your job), therefore it is much easier to find errors;
- Make the program much shorter, instead of writing the same things a hundred times, you can simply create a function and use it a hundred times, saving dozens of lines to write;
- It gives you the opportunity to write (and test) the functions individually, inserting them in the program only when you are sure of their functioning;
- If you write a function, nothing prevents you from copying it to another program, do you understand how long you can buy?

Many programming languages choose for us the programming paradigm to use: procedural, object-oriented or functional. Python gives us the opportunity to choose through the def statement:

```
>>> def par (x):
if x <2:
return 1
return x*par (x-1)
```

As you can see the par function accepts a single parameter, named n.

Let's try to write a function that returns the factorial number of any number passed as a parameter.

```
>>> def par (x):
```

```
if x <2:
return 1
return x*par (x-1)
```

We pay attention to the string we entered immediately after the function definition: it is the documentation string.

Let's try to type now in IDLE fact followed by the round parenthesis as in the figure:

```
>>> de fact (n):
“”” This function return factorial number
If n<2:
Return 1
Return n*fact (n -1)
>>> fact (
```

We can see the help window (the so-called "helper") which is automatically shown to us and which includes the list of parameters necessary for the function.

Values returned by functions in Python

In Python, contrary to what happens in other languages, when a function ends without the return statement, the value None is still returned. The same happens if a return is made that is not followed by a value. In the following example, functions f1, f2 and f3 all return None:

```
def f1 ():
pass
def f2 ():
return
def f3 ():
return None
```

We can verify it in IDLE:

```
>>> print (f1 (), f2 (), f3())
None none None
```

In case the value returned by a function is None, IDLE does not automatically show it:
we have to explicitly request it with the print function.

We probably have often had to make the function return more than one value.
In these cases, with languages other than Python, various escamotages had to be used: such as passing parameters by reference, returning values to an array and so on.

With Python we just need to return and contextually assign two or more values.
Let's clarify with an example:

```
>>> def eat_nut():
Return “Eat”, “Nut”
>>> a,b = eat_nut ()
>>>a
‘Eat’
>>> b
‘Nut’
>>>
```

Online assignment of multiple variables is really a very useful feature of Python and can also be used with sequences such as lists and tuples:

```
>>> list = [1,2]
>>> elem1, elem2, = list
>>> elem1
1
>>>tuple = ( ‘a’, ‘b’, ‘c’)
>>>elem1, elem2, elem3 = tupla
>>> elem3
‘c’
>>>
```

Passing parameters

In the fact function of a few paragraphs ago, we saw a very simple example, which included the passage of a single parameter. For a function we can obviously define all the parameters we want:

```
def minimum (n1, n2, n3):
“””” this function has the minimum in the parameter value””””
If n1 < n2 and n1 < n3:
Return n1
If n2 < n3:
Return n2
Return n3
```

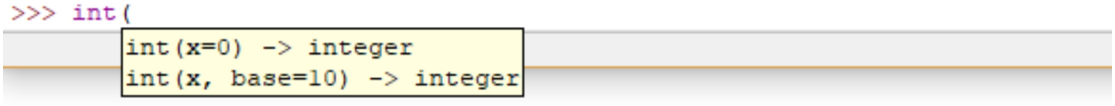
The parameters can also be indicated by name instead of by position.
The previous function can then be called in the following two ways:

```
minimo(23, 9, 64)
minimo(n2 = 9, n3 = 64, n1 = 23)
```

But there are other features and modes for passing parameters that allow us to define functions in Python in an extremely versatile and powerful way.

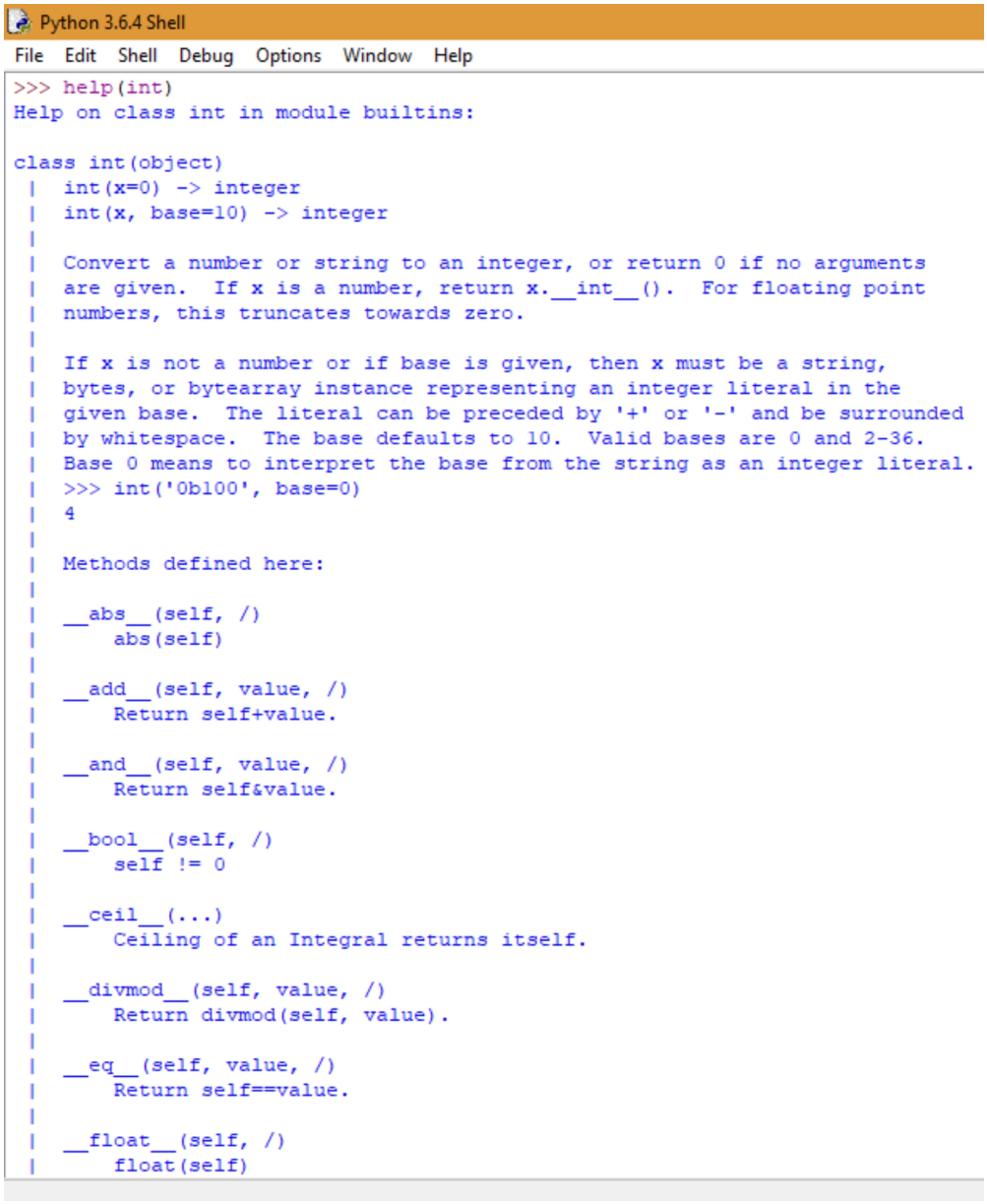
Optional parameters

For each parameter we can define an optional default value: in the absence of the respective parameter, the function will use the default value. Let's try to type the sequence "int (" in IDLE and wait, as you can see in the following figure, for the helper to appear:



We see that the int function accepts one or two parameters: x and base.
Given the second suggestion, we can guess that the base parameter is an optional parameter.

In this case, the value assumed by the optional parameter is specified.
If not, it would be sufficient to consult the documentation string in full using the help function:



From the documentation we would have discovered that the conversion, in the absence of the optional base parameter, is performed in the decimal

system. Therefore the base default would have been 10. We verify that in IDLE it is really like this:

```
>>> int (“13”)
13
>>>
```

Now let's try to explicitly specify a base:

```
>>> int (“13”, 8)
11
>>> int (“13”, 16)
19
>>>
```

The default is 10 and we can also pass it explicitly:

```
>>> int (“13, 10)
13
>>>
```

Just to remind us of this possibility, let's try to pass the parameters by name, reversing their order of definition:

```
>>> int (base=8, x=”13”)
11
>>>
```

Functions are objects

As we have seen, it is simple to define functions in Python. But a fundamental thing we have yet to say is that in Python a function is in effect an object. We can assign it to a variable:

```
>>> def stamp (what):
```

```
print (what)


>>> f = stamp
>>> f(“hello”)
Hello
>>>
```

Or even pass it as a parameter to another function:

```
>>> def stamp(what):
print (what)


>>> def do(funz, arg):
funz (arg)


>>> do(stamp, “hello”)
hello
>>>
```

In the next example we will try to create a dictionary of functions:

```
>>> def throw(what):
print (“we throw”, what)


>>> def keep(what):
Print (“we keep”, what)


>>> functions = dict()
>>> functions [“ugly”] = throw
>>>functions [“beautiful”]= keep
```

Now let's use the functions we just assigned to our dictionary:

```
>>> functions [“ugly”] (“spam”)
We throw spam
```

```
>>> functions [“beautiful”] (“book”)
We keep book
```

If we have a little more experience in programming, we can certainly imagine many situations in which this versatility would have come in handy.

CHAPTER 5. Loops OK

There are two types of loops in Python:

- the for loop: iterates through each element of an iterable;
- while: loop iterates as long as a condition is true.

For Cycle

The for loop allows us to iterate over all the elements of an iterable and execute a certain block of code. An iterable is any object capable of returning all the elements one after the other, such as lists, tuples, sets, dictionaries (return the keys), etc.

Let's see a simple example of a for loop:

```
>>> # print the square of each seq number
>>> seq = [1, 2, 3, 4, 5]
>>> for n in seq:
...print ('The square of', n, 'is', n ** 2)
...
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
```

We can note that:

- after the colon there is a block of indented code (which can also be made up of several lines);
- the loop iterates over all the elements of the sequence, assigns them to the variable n, and executes the block of code;
- in this example the variable n will assume the values of 1, 2, 3, 4, and 5 and for each value it will print the square;
- once the code block has been executed for all values, the for loop ends.

- the for loop is introduced by the keyword for, followed by a variable, the keyword in, an iterable, and finally the colon (:);

The following example shows how it is possible to use an if inside a for loop:

```
>>> # determines whether the seq numbers are even or odd
>>> seq = [1, 2, 3, 4, 5]
>>> for n in seq:
... print ('The number', n, 'is', end = '')
... if n% 2 == 0:
... print ('even')
... else:
... print ('odd')
...
The number 1 is odd
The number 2 is even
The number 3 is odd
The number 4 is even
The number 5 is odd
```

Again the code block is executed 5 times, one for each value of the sequence.
Those familiar with other languages will have noticed that in Python the for loop does not use indexes that are manually incremented (as in C, for example), but is rather more similar to constructs such as foreach.

Range

Since it often happens to want to work on sequences of numbers, Python provides a built-in function called range that allows you to specify an initial or start value (included), a final or stop value (excluded), and a step, and which returns a sequence of integers:

```
>>> range (5) # returns a range object with start equal to 0 and stop equal to 5
range (0, 5)
>>> list (range (5)) # converting it to a list we can see the values
```

```
[0, 1, 2, 3, 4]
>>> list (range (5, 10)) # with 2 arguments you can specify the start and stop
[5, 6, 7, 8, 9]
>>> list (range (0, 10, 2)) # with 3 arguments you can also specify the step
[0, 2, 4, 6, 8]
```

This function is particularly useful when combined with the for loop:

```
 >>> for n in range (1, 6):
... print ('The square of', n, 'is', n ** 2)
...
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
```

range can also be used in combination with the for loop if we want to repeat a block of code a fixed number of times:

```
>>> # print 'Python' 3 times
>>> for x in range (3):
... print ('Python')
... Python Python Python
```

In this case, a behavior more similar to the "traditional" for (like that of C) is obtained and the variable x is not used.

While Loop

```
>>> # remove and print numbers from seq until only 3 remain
>>> seq = [10, 20, 30, 40, 50, 60]
>>> while len (seq)> 3:
... print (seq.pop ())
...
60
50
40
>>> seq
```

```
[10, 20, 30]
```

We can note that:

- the while loop is introduced by the keyword while, followed by a condition (len (seq)> 3) and a colon (:);
- after the colon there is a block of indented code (which can also be made up of several lines); The while loop executes the block of code as long as the condition is true;
- in this case it removes and prints the elements of seq as long as there are more than 3 elements in seq;
- once the sequence is left with only 3 elements, the condition len (seq)> 3 becomes false and the cycle ends. Some other languages also include a construct called do-while, which performs at least one iteration before checking the condition.

In Python this construct does not exist, but obtaining an equivalent result is very simple:

```
>>> # ask the user to enter numbers as long as you guess
>>> n = 8
>>> while True:
... guess = int (input ('Enter a number from 1 to 10:'))
... if guess == n:
... print ('You guessed it!')
... break # guessed number, break the cycle
... else:
... print ('Retry you will be luckier')
... Please enter a number from 1 to 10: 3
Try again you will be luckier
Please enter a number from 1 to 10: 5
Try again you will be luckier
Please enter a number from 1 to 10: 8
You guessed!
```

In this example, the while loop is used to ask for numbers until the user guesses the correct number (8). The condition used is simply True, and since it can never become false, an infinite loop is created. The cycle is actually interrupted when the user guesses the number, using an if and the keyword break.

Break and Continue

Python has 2 constructs that can be used in the for and while loops:

- break: interrupts the cycle;
- continue: interrupts the current iteration and proceeds to the next.

For example, we can use a for loop to search for an element in a list and stop the search as soon as the element is found:

```
>>> seq = ['alpha', 'beta', 'gamma', 'delta']
>>> for elem in seq:
... print ('I'm checking', elem)
... if elem == 'range':
... print ('Item found!')
... break # element found, break the cycle
... I am checking alpha
I am checking beta
I am checking gamma
Item found!
```

As soon as the cycle reaches the "gamma" element, the if condition becomes true and the break interrupts the for loop. From the output it can be seen that 'delta' is not controlled.

```
>>> seq = ['alpha', 'beta', 'gamma', 'delta']
>>> for elem in seq:
... if len (elem) == 5:
... continue # proceed to the next element
... print (element)
...
Beta
```

In this example, however, we use continuous to "jump" the words that have 5 letters. From the output you can see that the if condition is true for "alpha", "gamma", and "delta", and in these cases the iteration proceeds immediately with the next element without printing. Only in the case of 'beta' (which has 4 letters), the continue is not performed and the element is printed.

for-else and while-else

A peculiarity of Python is the ability to add an else to the for and while. The block of code in the axis is performed if the cycle ends all the iterations. If, on the other hand, the cycle is interrupted by a break, the else is not performed. The syntax is similar to the one we have already seen with the if: the else must be indented at the same level as the for / while, it must be followed by a colon (:) and an indented block. Let's see an example where we give the user 3 attempts to guess a number:

```
>>> n = 8
>>> for x in range (3):
... guess = int (input ('Enter a number from 1 to 10:'))
... if guess == n:
... print ('You guessed it!')
... break # guessed number, break the cycle
... else:
... print ('Attempts finished. You didn't guess')
...
```

Note that even if the axis follows the if, the indentation corresponds to that of the for: it is therefore a for-else, not an if-else. If the user is able to guess the number, the if condition is fulfilled and the break is performed. In this case the for else is not executed because the for loop has been interrupted:

```
Please enter a number from 1 to 10: 3
Please enter a number from 1 to 10: 8
You guessed!
```

If, on the other hand, the user is unable to guess the number in 3 attempts, the for loop ends all the iterations and the for else is performed.

```
Please enter a number from 1 to 10: 3
Please enter a number from 1 to 10: 5
Please enter a number from 1 to 10: 7
Attempts over. You didn't guess
```

CHAPTER 6. Dictionaries, List and Tuples OK

Dictionaries

In Python the dictionaries (dictionary), or simply dict, are unordered collections of objects that belong to the so-called types of compound data (or containers) exactly like lists, tuples and sets, but unlike these, dictionaries do not show as elements of simple values, but of pairs made up of keys and values. This feature is due to the fact that the dictionaries have been designed to easily find values based on their keys when they are known to the developer.

The syntax for the creation of the dictionaries is simple, in fact one can be defined by delimiting with the square brackets of the pairs formed each by a key associated with the respective value through the ":" operator and separated by a comma. The values of the pairs of a dictionary can be associated with any type of data and be present more than once, the keys instead must be unique and their type immutable.

```
# Dictionaries definition
nome_dict = {1: 'homer', 2: 'bart'}
>>> nome_dict
{1: 'homer', 2: 'bart'}
```

In the example above, a dictionary called "name_name" has been defined, consisting of two pairs of keys and values: the first pair has "1" as key and "homer" as value, the second "2" as key and "bart" "as a value. Keys don't have to be numeric just as values don't have to be strings:

```
# Definition of a dictionary with string keys
name_nict = {'name': 'homer', 'surname': 'simpson'}
>>> dict_name
{'first name': 'homer', 'last name': 'simpson'}
# Definition of a dictionary with numeric values
>>>ict_name = {'first': 1, 'second': 2}
>>> dict_name
{'second': 2, 'first': 1}
# Definition of a dictionary with different types of values and keys
dict_name = {'name': 'marge', 1: [4,6,8]}
>>> dict_name {'name': 'marge', 1: [4, 6, 8]}
```

Python also has an alternative dictionary definition syntax that uses the native dict () function.

```
# Definition of a dictionary with dict ()
>>> dict_name = dict ({1: 'php', 2: 'java'})
>>> dict_name
{1: 'php', 2: 'java'}
# Definition of a dictionary with dict ()
# through a sequence of elements made up of pairs
>>> dict_name = dict ([(1, 'python'), (2, 'javascript')])
>>> dict_name
{1: 'python', 2: 'javascript'}
```

Access the elements of a dictionary

Since the dictionaries contain pairs of keys and values, the easiest way to access a value is to refer to its key; for this purpose, a syntax is used which involves using the name of the dictionary followed by a key inserted in square brackets:

```
# Access the elements of a dictionary
>>>ict_name = {'name': 'montgomery', 'surname': 'burns', 'years': 102}
>>> name_dict ['surname']
'Burns'
>>> name_nict ['years'] 102
```

The get () method offers an alternative way to access the elements of a dictionary, it accepts the key associated with a certain value as a parameter:

```
# Access the elements of a dictionary with get ()
>>> name_nict = {'name': 'apu', 'surname': 'Nahasapeemapetilon', 'years': 38}
>>> dict_name
{'years': 38, 'surname': 'Nahasapeemapetilon', 'name': 'apu'}
>>> name_dict.get ('surname')
'Nahasapeemapetilon'
>>>
```

The main difference between the first methodology and the one based on get () lies in the fact that the method does not return anything if a nonexistent key is passed to it as an argument, with "dict_name ['nonexistent_key_name']" an error would instead be generated .

Change the elements of a dictionary

To modify an element of a dictionary it is sufficient to associate a new value to the key of a pair:

```
# Modify an element of a dictionary
>>> name_nict = {1: 'euro', 2: 'dollar', 3: 'pound'}
>>> dict_name
{1: 'euro', 2: 'dollar', 3: 'pound'}
>>>ict_name [2] = 'US dollar'
>>> dict_name
{1: 'euro', 2: 'US dollar', 3: 'pound'}
```

In all dictionary manipulation operations (for example modification and removal) it is always good to keep in mind that unlike what happens with lists and tuples, in dictionaries we do not have an automatic indexing from "0" to "n" operated by the Python interpreter, but by user-defined keys; therefore, the latter must always be used to access the values.

Adding an item in a dictionary

To add an element to a dictionary you must specify a new key associating a value to it using the syntax used in the following example:

```
# Add an element to a dictionary
>>>ict_name = {'Europe': 'euro', 'USA': 'dollar', 'UK': 'pound'}
>>> dict_name
{'Europe': 'euro', 'USA': 'dollar', 'UK': 'pound'}
>>> name_nict ['Switzerland'] = 'free'
>>> dict_name
{'Europe': 'euro', 'USA': 'dollar', 'UK': 'pound', 'Switzerland': 'franc'}
```

When inserting an additional key / value pair into a dictionary, it is necessary to pay attention to the name used for the new key, this is because if you indicate the name of a key already present in the dictionary, its value will be updated.

Removing an item from a dictionary

The pop () method allows you to remove a key / value pair from a dictionary by accepting the key of the pair you want to delete as a parameter; the Python engine will take care of showing on the screen the value that will be removed together with the key:

```
# Removing an item from a dictionary
>>>ict_name = {'a': 1, 'b': 2, 'c': 3, 'd': 4, e: 'f'}
>>> dict_name
{'a': 1, 'd': 4, 'b': 2, 'c': 3, 'f': 6, 'e': 5}
>>> name_dict.pop ('b')
2
>>> dict_name
{'a': 1, 'd': 4, 'c': 3, 'f': 6, 'e': 5}
```

The popitem () method works similarly to pop () but with the difference that it does not accept any parameter, in practice, by calling this method the Python interpreter arbitrarily chooses the key / value pair to be eliminated:

```
# Removing an item from a dictionary with popitem ()
>>> name_dict.popitem ()
('a', 1)
>>> dict_name
{'d': 4, 'c': 3, 'f': 6, 'e': 5}
>>>
```

clear () is instead the method to be used to delete all the contents of a dictionary through a single instruction:

```
# Clear the contents of a dictionary with clear ()
>>>ict_name.clear ()
>>> dict_name
{}
```

Finally, the () function works like the pop () method when the key of a pair is passed as an argument, but it can also be used to permanently delete a dictionary:

```
# Delete a dictionary with del ()
```

```
>>> of the name_nict
>>> dict_name
Traceback (most recent call last):
  File "<pyshell # 26>", line 1, in <module>
    nome_dict
NameError: name 'name_dict' is not defined
```

Attempting to retrieve a dictionary deleted with del () the Python interpreter will inform the developer that it has not been defined.

List

In Python it is possible to define a list, to which a name can be attributed, inserting elements in square brackets ("[..]"). Those who already work with other languages for programming or development will certainly know arrays, or "vectors", which are variables intended to contain further variables; syntactically and conceptually the lists can remember arrays, but their functioning has some peculiarities that make them different.

Basically a list can exist but be empty, that is, not present any element inside it:

```
# Example of empty list
list_name = []
```

If a list is not empty, it will be necessary to separate the elements that compose it by a comma. It is therefore possible to define lists that contain elements associated with a single type of data:

```
# Example of a list containing only numeric values
list_name = [15, 25, 35, 45, 55]
# Example of a list containing only strings
list_name = ['Homer', 'Bart', 'Lisa']
```

Just as you can create lists that present elements of different nature:

```
# Example of list containing elements of different types
```

```
# numbers and strings
list_name = [65, 'Homer', 7.3]
```

Finally, the creation of nested lists is also allowed, i.e. lists that have other lists among their elements.

```
# Example of nested list
list_name = ['Homer', 3.7, 10, [29, 39, 49]].
```

Automatic generation of lists of integers

range () is a native Python function (or more properly an "immutable sequence type") designed to automatically generate a list based on a range of values or a numeric value passed as an argument; it is particularly useful when you need to define lists made up of a large number of numerical elements.

In the case of a parameter expressed as an interval, we will have an instruction like the following:

```
# Using range () to generate
# a list based on an interval
>>> range (1.8)
```

will lead to the generation of a list consisting of 7 elements:

```
[1, 2, 3, 4, 5, 6, 7]
```

These elements will be the result of the call to the range () function which will evaluate the two arguments of the range by returning a list containing all the integer values starting from the first, included in the list, up to the second, excluded from the list instead.

The two arguments of the interval may be followed by a third argument called step; it specifies the interval between successive values, which is why if, for example, you want to obtain a list made up of only the odd numbers covered between "1" and "8" you could operate in this way:

```
# Using range () to generate
```

```
# a list based on an interval
# with steps equal to 2
>>> range (1,8,2)
# list generated
# [1, 3, 5, 7]
```

When, on the other hand, we pass the function an integer as in the example below:

```
# Using range () to generate
# a list based on an integer
>>> range (8)
```

you will get a list populated like this:

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

The elements present will therefore be 8, placing the "0" as an element and (not) initial value.

Indexing of the elements of a list

By default and lists in Python, internal elements are indexed numerically; indexing will start from "0", so a list consisting of three elements will have "0", "1" and "2" as indexes. It follows that a specific element of a list can be called up via its index, as in the following example:

```
# Access to an element of a list
# through its index
# definition of the elements in the list
>>> list_name = ['a', 'b', 'c', 'd', 'e']
# access to element with index "3"
>>> list_name [3]
```

In this case, the element called will be "d", ie the fourth inserted in the list, this is because "a" is associated with the index "0", "b" to "1", "c" to "2" and di consequence "d" has as index "3".

Access to the elements in the list could also take place through negative indexing, where the last element will have the index "-1", the penultimate

"-2" and so on. A call like the following will therefore have the result of allowing access to the "a" element.

```
# Access to an element of a list
# through the negative index
# definition of the elements in the list
>>> list_name = ['a', 'b', 'c', 'd', 'e']
# access to element with index "-5"
>>> list_name [-5]
```

It is also possible to access the elements of a list on the basis of a range of values; be careful that the two components of the range will correspond to the index numbers, which is why an expression like this:

```
# Access to multiple elements of a list
# through an interval
# definition of the elements in the list
>>> list_name = ['a', 'b', 'c', 'd', 'e']
# access to elements with interval "0: 2"
>>> list_name [0: 2]
```

will allow access to the elements "a", "b", ie those whose index goes from "0" to "2" (excluded). The colon symbol (":"), previously used for the definition of the interval, is called slicing operator. It can also be used for other purposes, such as defining the index from which to start for data access:

```
# Access to multiple elements of a list
# starting from a specific index
# definition of the elements in the list
>>> list_name = ['a', 'b', 'c', 'd', 'e']
# access to the first element in the list ('a') via negative index
# the last 4 elements will be excluded
>>> list_name [: - 4]
# access from the third element to the last
# 'c', 'd' and 'e'
>>> list_name [2:]
# access to all items in the list
>>> list_name [:]
```

As described in the next chapter, Python has constructs that allow you to perform more advanced operations than simply accessing values, such as those dedicated to manipulating lists.

Tuples

As anticipated, tuples are constructs which, as with lists, can contain a collection of elements; the substantial difference between the lists and the tuples, however, lies in the fact that while in the former the elements that the components must be replaced, in the tuples they become immutable.
The syntax for defining tuples requires that the elements to be inserted into them are delimited by round brackets and associated with a name by assignment (operator "=").
Basically, as already seen for the list, tuples can exist, and therefore have been defined, but at the same time be empty and therefore do not have any element inside:

```
# Example of empty tuple
tuple_name = ()
```

Always pay attention to the rule that to assign a single element to a tuple requires that it be followed by a comma:

```
# Definition of a tuple with a single element
>>> tuple_name = ('Homer')
>>> type (double_name)
<class 'str'> # without the comma is not identified as a tuple
>>> tuple_name = ('Homer',)
>>> type (double_name)
<class 'tuple'>
```

In the event that a tuple is not empty, you will be asked to separate the elements that make it up with a comma, you can therefore create tuples intended to contain elements associated with a single type of data:

```
# Example of a tuple containing only numeric values
triple_name = (16, 26, 36, 46, 56)
# Example of a tuple containing only strings
tuple_name = ('Homer', 'Abraham', 'Lisa')
```

In the same way, tuples can be defined that present elements associated with different types of data.

```
# Example of a tuple containing elements of different types
```

```
# numbers and strings
triple_name = (66, 'Marge', 6.1)
```

Finally, nested tuples can be generated, that is, tuples that have other tuples among their elements.

```
# Example of nested tuple
triple_name = ('Apu', 2.8, 15, (21, 31, 41))
```

If desired, you can also insert a list as an element of a tuple.

```
# nesting with insertion of a list in the tuple
tuple_name = ('Homer', [7, 6, 5], ('a', 'b', 'c'))
```

The syntax of tuples also supports their creation without the use of round brackets and, in this case, we are talking about an operation called tuple packing:

```
# Tuple packing
triple_name = 5, 8.7, 'Bart'
```

Indexing of the elements of a tuple

As already observed for lists, tuples also require that their elements be indexed numerically; indexing will therefore start from "0", which is why a tuple consisting of only three elements will have "0", "1" and "2" as indices. A certain element of a tuple can then be recalled through the reference index:

```
# Access to an element of a tuple
# through its index
# definition of the elements of the tutple
>>> tuple_name = ('z', 'y', 'x', 'w', 'v')
# access to element with index "2"
>>> tuple_name [2]
```

In this case the element called will be "x", ie the third inserted in the tuple, in fact "z" is associated with the index "0", "y" to "1" and consequently "x"

will have "2" as index. Note how the index was specified in square and non-round brackets, taking up the same syntax as the lists.
An alternative procedure for accessing the elements of the tuple involves exploiting negative indexing, in this case the last element will have the index "-1", the penultimate "-2" and so on, exactly as for the lists. The following call will, for example, allow access to the "y" element.

```
# Access to an element of a tuple
# through the negative index
# definition of the elements of the tuple
>>> tuple_name = ('z', 'y', 'x', 'w', 'v')
# access to element with index "-4"
>>> tuple_name [-4]
```

You can also access the elements of a list by defining a range of values; the two components of the range will correspond to the index numbers, so an expression like the one proposed below:

```
# Access to multiple elements of a tuple
# through an interval
# definition of the elements of the tuple
>>> tuple_name = ('z', 'y', 'x', 'w', 'v')
# access to elements with range "0: 3"
>>> list_name [0: 3]
```

will allow access to the elements "z", "y" and "x", that is to say those whose index goes from "0" to "3" excluding the latter. The colon symbol (":") also used in the lists for the definition of the interval, that is the already known slicing operator, can also be adopted for different purposes, for example with the aim of defining the index from which to start. to access the elements of the tuple:

```
# Access to multiple elements of a tuple
# starting from a specific index
# definition of the elements of the tuple
>>> tuple_name = ('z', 'y', 'x', 'w', 'v')
# access to the last two elements of the tuple ('z' and 'y') via negative index
# the first 3 elements will be excluded
>>> tuple_name [: - 3]
# access from the second element to the last
# 'y', 'x', 'v' and 'w'
>>> tuple_name [2:]
# access to all items in the list
```

```
>>> tuple_name [:]
```

If you want to access an element of a tuple nested within another tuple, the reference indices must also be nested:

```
# Access to an element of a nested tuple
>>> tuple_name = ('Apu', 2.8, 15, (21, 31, 41))
>>> tuple_name [3] [1]
```

The result of the call will be equal to "31", ie the second element (with index "1") of the tuple which in turn represents the fourth element, with index "3", of the tuple in which it is nested.

Classes

In this lesson we will see in more detail how to define classes, methods and attributes, and how to use them. We will also see how to create instances and subclasses Define classes The simplest class we can define in Python is:

```
>>> class Test:
... pass
```

As we can see from the example, to define a class just use the class keyword, followed by the name we want to give to the class (in this case Test), followed by a colon (:), followed finally by a block of indented code (in this case there is only the pass).

This syntax is similar to the one used to define functions, with the difference that the keyword class is used instead of def and that the list of arguments after the name is not present.

It is also important to note that, by convention, class names generally use CamelCase, except for some basic types such as int, str, list, dict, etc. Once the Test class is defined, we can verify that the name Test refers to the class. We can therefore use the class to create different instances, simply using the () to "call" it:

```
>>> Test
```

```
# Test refers to the class <class '__main __. Test'>
>>> Test ()
# Test () returns a new instance <__ main __. Test object at 0x7f87ed6a26d8>
>>> Test ()
# Test () returns a new instance <__ main __. Test object at 0x7f87ed6a26a0>
```

Copy Each call returns a new and separate instance, as can be seen from the different ID of the two instances created in the example (0x7f87ed6a26d8 and 0x7f87ed6a26a0). Note that in Python 2 it is important to define classes using the syntax class Test (object): ... to make them inherit from object. In Python 3 all classes automatically inherit from object.

Methods

In the previous lesson we also saw that the methods are similar to functions defined within the class:

```
>>> # let's define a Test class
>>> class Test:
... # let's define a method that prints a message
... def method (self):
... print ('method called')
...
>>> inst = Test () # we create an instance of Test
>>> inst.method ()
```

From this example we can see several things:

- the definitions of the methods are found indented within the class;
- the syntax used to define the methods is the same as that used to define the functions;
- the methods must define an additional parameter which is conventionally called self;
- to call a method just use the instance.method () syntax.

Self

As noted in the previous example, the most important difference between the definition of methods and functions is the presence of self. self is an argument that refers to the instance, and even if the methods must explicitly declare it, it is not necessary to pass it explicitly. Let's look at another example to better understand how self works:

```
>>> # let's define a Test class
>>> class Test:
... # let's define a method that prints self
... def method (self):
... print ('self is:', self)
...
>>> inst = Test () # we create an instance of Test
>>> inst # we show the instance id
<__ main __. Test object at 0x7fef42c03f28>
>>> inst.method () # we verify that self corresponds to the instance
self is: <__ main __. Test object at 0x7fef42c03f28>
```

The reason why it is not necessary to explicitly pass self is that the expression inst.method () is simply syntactic sugar for Test.method (inst):

```
>>> Test.method # Test.method is a function defined in the class
<function Test.method at 0x7fef42c27840>
>>> inst.method # inst.method is an instance-related method
<bound method Test.method of <__ main __. Test object at 0x7fef42c03f28 >>
>>> Test.method (inst) # we can call Test.method and pass inst
self is: <__ main __. Test object at 0x7fef42c03f28>
>>> inst.method () # we can call inst.method directly
self is: <__ main __. Test object at 0x7fef42c03f28>
```

Both ways produce the same result, but normally the second way (inst.method ()) is used: when we call inst.method (), Python actually automatically goes back to the inst class and executes Test.method (inst). For this, each method defined in the class must accept self as the first argument. Also note that self has nothing special or different from other arguments (unlike other languages such as Java which uses the keyword this to refer to the instance).

Since self refers to the instance, we can use it to access other attributes and methods defined within the class simply by making self.attribute or self.method ().

Initialize Instances

The classes also support several "special" methods which are identified by the presence of two underscores before and after the name. These methods are not called directly by making inst .__ method__, but are typically called automatically in special situations One of these special methods is __init__, automatically called each time an instance is created:

```
>>> # let's define a Test class
>>> class Test:
...
# let's define an __init__ that prints a message
...
def __init __ (self):
...
print ('New instance created!')
...
>>> Test ()
# when we instantiate, __init__ is called New instance created!
<__ main __. Test object at 0x7fef42c1eeb8>


>>> Test ()
# when we instantiate, __init__ is called New instance created!
<__ main __. Test object at 0x7fef3ff76390>
```

Copy __init__ also has another peculiarity: the arguments passed during the creation of the instance are received by __init__. This allows us to automatically create different instances based on past topics:

```
>>>
# let's define a Dog class
>>> class Dog:
...
# let's define an __init__ that accepts a name
...
def __init __ (self, name):
...
# let's create an instance attribute for the name
```

```
...
self.name = name
...
>>>
# let's create two instances of Dog
>>> rex = Dog ('Rex')
>>> fido = Dog ('Fido')
>>> rex.name
# we check that the name of the first one is Rex 'Rex'
>>> fido.name
# and that the name of the second is Fido
'Fido'
```

It is also important to note that __init__ is not equivalent to the constructors present in other languages, since it does not create the instance, but only initializes it.

Attributes

In these last two lessons we have seen that attributes are values associated with the instance (or class) and we have also seen some examples of declaration and use of attributes.

Attributes can be grouped into two categories:

- instance attributes;
- class attributes.

As you can guess from the names, the instance attributes are tied to a specific instance, while the class attributes are tied to the class. Instance attributes are generally more common and are declared by instance.attribute = value.
When an instance attribute is declared within a method (for example the __ init__), self.attributo = value is used, since self refers to the instance:

```
>>> # let's define a Dog class
>>> class Dog:
...
# let's define an __init__ that accepts a name
...
```

```
def __init __ (self, name):
...
# let's create an instance attribute for the name
...
self.name = name
...
# let's define a method that accesses the name and prints it
...
def print_name (self):
...
print (self.name)
...
>>>
# let's create an instance of Dog
>>> dog = Dog ('Rex')
>>>
# we access the instance attribute "name"
>>> dog.name 'Rex'
>>>
# let's call a method that prints the name
>>> dog.print_name () Rex
>>>
# let's change the value of the instance attribute
>>> dog.name = 'Fido'
>>>
# check that the name has been changed
>>> dog.name 'Fido'
>>> dog.print_name ()
Fido
```

It is also possible to add or remove attributes from instances, but generally not recommended, since it is preferable to have the same attributes (albeit with different values) on all instances of the same class:

```
>>>
# let's create an instance of Dog
```

```
>>> rex = Dog ('Rex')
>>>
# let's check the attributes and methods of the instance
>>>
# using dir () (special methods have been omitted)
>>> dir (rex) [..., 'name', 'print_name']
>>>
# let's add a new attribute to the instance
>>> rex.job = 'police officer'
>>>
# check that the attribute has been added
>>> dir (rex) [..., 'job', 'name', 'print_name']
>>>
# we access the new attribute
>>> rex.job 'police officer'
>>>
# we remove the attribute using "del"
>>> of rex.job
>>>
# check that the attribute has been removed
>>> dir (rex) [..., 'name', 'print_name']
```

Class attributes are class-related values, which are common and accessible by all instances. To declare class attributes, there are two ways: using class.attribute = value or using attribute = value in the body of a class declaration:

```
>>>
# let's define a Dog class
>>> class Dog:
...
# let's define a class attribute
...
scientific_name = 'Canis lupus familiaris'
...
# let's define an __init__ that accepts a name
...
def __init __ (self, name):
...
# let's create an instance attribute for the name
...
self.name = name
...
>>>
# we create two instances of Dog
>>> rex = Dog ('Rex')
```

```
>>> fido = Dog ('Fido')
>>>
# we check that each instance has a different name
>>> rex.name 'Rex'
>>> fido.name 'Fido'
>>>
# we access the class attribute from Dog
>>> Dog.scientific_name 'Canis lupus familiaris'
>>>
# we access the class attribute from the instances
>>> rex.scientific_name 'Canis lupus familiaris'
>>> fido.scientific_name 'Canis lupus familiaris'
>>>
# let's change the value of the class attribute
>>> Dog.scientific_name = 'Canis lupus lupus'
>>>
# check the change from the instance
>>> rex.scientific_name 'Canis lupus lupus'
```

In this example we see that the scientific name ('Canis lupus familiaris'), common to all instances of Dog, is declared during the definition of the class, and is accessible both from the instances and from the class itself. Each instance then has a unique name (e.g. 'Rex' and 'Fido'), which is defined in the __ init__ and which is accessible only from the instances.

Modules

In Python a module is essentially a file that contains code written in this language. The modules are characterized by the extension ".py", therefore the file "instruction.py" will correspond to the module name "instruction".

The purpose of the modules is twofold: they contain instructions and definitions that can be reused as many times as you want without having to rewrite them, just "import" the modules where necessary; moreover, they are particularly convenient for easily managing large quantities of code.

Creating a form

The procedure for creating a module is summarized in writing the code it is intended to contain and saving the file with the ".py extension. As for the

syntax to be used, the rules are those seen so far, so a module could contain, for example, a code like this, intended to carry out a simple multiplication through a function ("mul ()") to which pass 2 factors as parameters:

```
# Define the code of a module
def mul (x, y):
   "" "Simple application that multiplies
   2 values and returns the result "" "


   product = x * y
   return product
```

Once the code is saved in a file, which we will call for example "app.py", we will have a module that can be imported into any program.

Importing a form

To import a module you must use the keyword import followed by the name of the module and not that of the file that represents it, therefore omitting the extension ".py"; once this is done, the instructions contained therein will become available for the application in which it was imported. So, in the case of the previously created module, we will have to use a syntax like the following:

```
# Importing a form
>>> import app
```

After the import, to use the instructions contained in the module we will have to perform an operation which technically consists in introducing the names of the functions defined in "app" in the current symbol table. It is a matter of making the functions of the module usable, which is possible by concatenating the name of the module to the desired function by means of the already known operator ".".
In our case the "app" module presents only the "mul ()" function which we will call in this way:

```
# Importing a form and calling the function
>>> import app
```

```
>>> app.mul (6.5) 30
```

An efficient way to manage modules is to rename them in order to save time in typing the code. "App" could for example be renamed to "a" like this thanks to the as clause:

```
# Import and rename a module
>>> import app as a
```

Be careful, however, because once a module is renamed the new name must also be used in the function call, otherwise Python will produce an error no longer recognizing the initial name of the module:

```
# Call functions of a renamed module
>>> import app as a
>>> app.mul (8.4)
Traceback (most recent call last):
File "<pyshell # 16>", line 1, in <module> app.mul (8.4) NameError: name 'app' is not defined >>>
a.mul (8.4) 32
```

Manage calls to instructions

A module can contain numerous instructions and definitions, therefore it can happen that there is no need to import it in full, recalling only some elements. To do this we will use the from keyword followed by the module name and "import" followed in turn by the name of the construct we want to import. If our module contained multiple instructions and we wanted to import only the "mul ()" function we should therefore proceed as follows:

```
# Import only one function of a module
>>> from app import mul
>>> mul (3,5) 15
```

Note how in this case it is possible to use the function without the need to concatenate its name to that of the module. In case you want to import multiple instructions of a module excluding others, just use the construct seen above, where the names of the chosen instructions will be separated with a comma:

```
# Multiple importation of instructions from a form
>>> from app import mul, second_instruction, third_instruction
```

Standard modules in Python

For the convenience of the developers, Python provides a large number of standard modules, basically the modules provided natively by the language to carry out the most varied operations.

These modules can be imported using the same syntax that can be used for user-defined modules.

To offer an example we can consider the calendar module, as the name suggests it is a solution that allows you to manage calendars through dedicated methods, also natively provided by Python.

By default, following the so-called "European convention" "calendar" requires that the first day of the week be Monday (associated with the index "0") with Sunday as the last day (index "6"). To check what has been said, you can import the form and then call the firstweekday () method which shows the index for the first day of the week as output:

```
# Importing the calendar module and viewing the index
# of the first day of the week
>>> import calendar
>>> calendar.firstweekday ()
0
```

If you want to change the default setting, you can use another method, setfirstweekday (), which accepts the day of the week index with which you want to start the week as a parameter. In the event that the choice should fall on Sunday, following a usual convention in countries like the United States, we should proceed as follows:

```
# Import the calendar module and modify the index
# of the first day of the week
>>> import calendar
```

```
>>> setfirstweekday (6)
>>> calendar.firstweekday ()
6
```

Just like user-defined modules, standard modules can also be renamed. In the same way it is possible to import only specific parts of a module or all the definitions and instructions that compose it.

CHAPTER 7. Object Programming

Management of larger programs

At the beginning of this book, we saw four basic programming models used to write programs:

- Sequential code
- Conditional code (if instructions)
- Repetitive code (cycles)
- Stores and reuses (functions).

In the previous chapters we explored the use of simple variables and structures for data management such as lists, tuples and dictionaries. During program development, we design the structures with which we store data and write the code necessary to manipulate it.

There are many ways to write programs and, at this point, you have probably written some "not particularly elegant" and other "more elegant" scripts. Even if your scripts are quite small in size, you are starting to understand how there is a bit of "art" and "aesthetics" in developing code. As programs reach millions of lines, it becomes increasingly important to write code that is easy to interpret. If you are working on a million-line program, you will never be able to keep the entire program in mind at the same time.

We need to find ways to break the program into multiple pieces so that solving problems, fixing a bug or adding a new feature is as simple as possible. In a sense, object-oriented programming is a way of organizing the code so that we can focus our attention on 500 lines of code while momentarily ignoring the other 999,500.

How to get started

As with many aspects of programming, you need to learn the basic concepts of object-oriented programming before you can use it effectively. So consider this chapter as a way to study some basic terms and concepts illustrated through some simple examples with the aim of laying the foundations for future learning. Although in the rest of the book we will adopt object-oriented programming in many programs we will not develop new ones. What we want to achieve with this chapter is a basic knowledge of how objects are made, how they work and, above all, how to exploit the possibilities offered by Python libraries .

Use objects

You will realize that we actually used the objects throughout the course: Python provides us with many objects already integrated into it. Here is a simple script whose first few lines should seem very simple and familiar to you.

```
stuff = list ()
stuff.append ( 'python')
stuff.append ( 'Chuck')
stuff.sort ()
print (stuff [0])
print (stuff .__ getitem __ (0))
print (list .__ getitem __ (stuff, 0))
```

Rather than focusing on what you could achieve through these few lines of code, let's see what is really going on from an object-oriented perspective. Don't worry if the first time you read the next few paragraphs it seems that we don't make much sense since I haven't explained the meaning of many of the terms yet. The first line is building a list object, the second and third lines call the append () method, the fourth line calls the sort () method and the fifth line is getting the element in position 0. The sixth line calls the __getitem __ () method in the stuff list with parameter zero.

```
print (stuff .__ getitem __ (0))
```

The seventh row is an even more detailed way of recovering the element that occupies the zero position in the list.

```
print (list .__ getitem __ (stuff, 0))
```

In this script, we call the __getitem__ method of the list class, we pass it in the list (stuff) with the element we want to retrieve from the list as the second parameter. The last three lines of the program are completely equivalent, but it is easier to use the syntax with square brackets to search for an element in a specific position of a list. We can take a look at the potential of an object by studying the output of the dir () function:

```
>>> stuff = list()
>>> dir (stuff)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

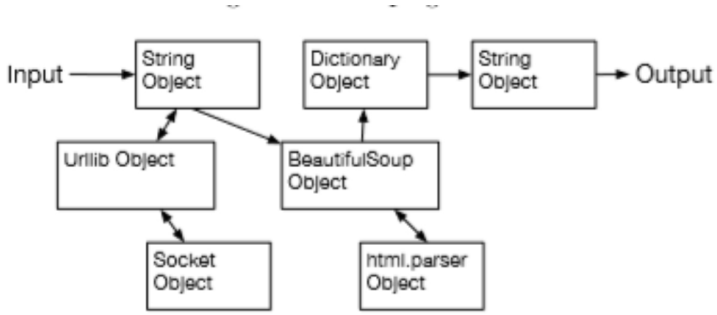
More precisely dir () lists the methods and attributes of an object. The rest of this chapter will provide you with a more precise de fi nition of all the terms already reported, therefore, after completing this chapter, re-read the previous paragraphs to verify what you have really learned.

The first scripts

A program in its simplest form requires one or more inputs, does some processing and produces an output. the following script for converting floor numbers to an elevator is very short but complete and has all three of these steps.

```
usf = input ('Enter the US Floor Number:')
wf = int (usf) - 1
print ('Non-US Floor Number is', wf)
```

If we focus a little more on this program: we can see that the "external world" and the internal world of the program coexist. The input and output operations represented the way in which the program interacts with the outside world. Within the program we have the code and data necessary to perform the task for which it was designed.



Some well-defined interactions with the "outside" world take place within the program, which are generally not something we focus on. When we write about the code, we only care about the details "within the program". One way to think about object-oriented programming is to want to try to separate our program into multiple "zones".

Each "zone" is composed of code and data (as if it were a program in its own right) and has well-defined interactions with the outside world and with the other zones within the program. If we take into consideration the link extraction script in which we used the BeautifulSoup library we can see a program consisting of multiple objects that interact with each other to perform a task:

```
import urllib.request, urllib.parse, urllib.error
 from bs4 import BeautifulSoup
```

```
import ssl


# Ignore SSL certificate errors
ctx = ssl.create_default_context ()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
url = input ('Enter -')
html = urllib.request.urlopen (url, context = ctx) .read ()
soup = BeautifulSoup (html, 'html.parser')
# Retrieve all of the anchor tags
tags = soup ('a')
for tag in tags:
print (tag.get ('href', None))
```

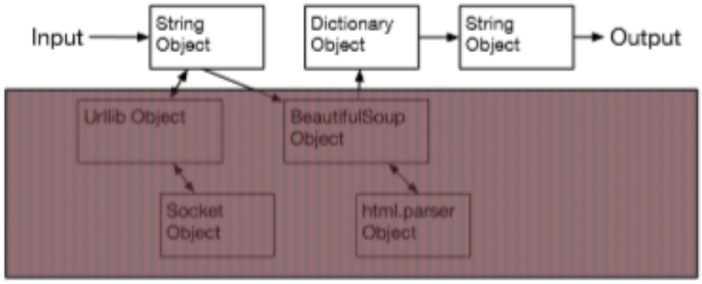
The URL is requested as a string, then passed in urllib for the recovery of data from the web. The urllib library makes the network connection by leaning on the socket library. The string downloaded from urllib is delivered to BeautifulSoup for analysis.

BeautifulSoup uses the html.parser1 object to return an object to which the tags method is applied () to generate a dictionary of tag objects, which are browsed through the get () method to display the possible 'href' attribute. We can represent the interaction of the objects of this program just described with the following image.

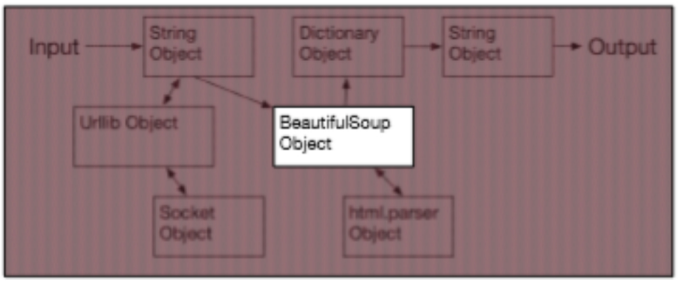
At this moment, the most important thing is not to fully understand the functioning of this program, but rather to see how this set of objects has been structured and how the exchange of information is orchestrated. It is also important to note that when you studied the functioning of a script present in the first chapters of this book, you were able to fully understand what was happening without even realizing that the program was "managing the movement of data between the objects present". Back then, for you it was just lines of code that did the job.

Break down a problem - encapsulation

One of the advantages of the object-oriented approach is that it can reduce the complexity of a script: even if you need to know how to take advantage of urllib and BeautifulSoup, you don't need to know how these libraries work internally. This allows you to focus on the part of the problem that you need to solve by letting go of other parts of the program.



This ability to focus on the part of a program that interests us ignoring the rest of the program is also useful for the developers of the objects themselves: the programmers who take care of the BeautifulSoup development do not need to know or worry about how you recovered the HTML page from analyze, which parts you want to read or what you intend to do with the data you extracted.



There is another word that is commonly used to convey the idea that we are ignoring the internal functioning of the objects we use: "encapsulation". In other words, it means that we can know how to use an object without really knowing what really happens inside it.

Our first Python object

Simplifying to the maximum, an object is a code portion with its own smaller data structures of the entire program. Defining a function means indicating a well-defined portion of code by a name that we can subsequently invoke according to our needs using only the name that we have assigned to it. An object can contain a number of functions (which we call "methods") and the data used by those functions. We define given the elements that make up the "attributes" of the object. The class keyword is used to define the data and code that make up each of the objects. class also includes the class name and begins with an indented block of code in which we include attributes (data) and methods (code).

```
class PartyAnimal:
x = 0
def party(self) :
self.x = self.x + 1
print("So far",self.x)
an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)
```

Each method that looks like a function starts with the def keyword and consists of a block of indented code. The previous example has an attribute (x) and a method (party). The methods have a special first parameter called by self convention. Just as the def keyword does not cause the function code to execute, the class keyword does not create an object. Rather, the class keyword defines a model that indicates which data and code will be contained in each PartyAnimal type object.

You might think that the classroom is a cookie cutter and that the objects created using the classroom are cookies2. When preparing the biscuits, you know that the icing should not be put on the mold but rather on the biscuits and you always have the possibility to put a different icing on each biscuit. But now let's continue to analyze the sample code. Look at the first line of executable code:

```
an = PartyAnimal()
```

This is the point where we tell Python where to build (eg create) an object or "the instance of the class called PartyAnimal". It looks like a function call to the class itself and Python builds the object with the correct data and methods and returns the object which is then assigned to the variable an. In a sense, all of this is quite similar to the line we have used since the beginning:

```
counts = dict()
```

Here we are telling Python to create an object using the template dict (already present in the language), return the dictionary instance and assign it to the counts variable.
When we want to use the PartyAnimal class to build an object, the variable an allows us to point to that object, access the code and data that that particular instance of a PartyAnimal object contains. Each Partyanimal object / instance contains an x variable and a party method / function which is called in this line:

```
an.party()
```

When the party method is called, the first parameter (called by self convention) points to the particular instance of the PartyAnimal object which is called within the party. Within the party method, we can see the line:

```
self.x = self.x + 1
```

This syntax using the 'dot' operator indicates 'the x inside self'. So each time party () is called the internal value x is increased by 1 and then displayed. To help you understand the di enza erence between a global function and a method within a class / object, in the following line you can see another way to call the party method within the object an:

```
PartyAnimal.party(an)
```

In this variant we are accessing the code from the class and explicitly passing the pointer of the object an as the first parameter (for example self within the method). You can think of an.party () as an abbreviation of the previous line. When the program is executed, the following output will be generated: So far 1 So far 2 So far 3 So far 4 The object was created and the

party method is called four times, increasing and displaying the value of x inside the object an.

Classes as types

As we have seen, in Python all variables have a type that we can examine via the integrated dir function. these features are extended to the classes we create.

```
class PartyAnimal:
x = 0
def party(self) :
self.x = self.x + 1
print("So far",self.x)
an = PartyAnimal()
print ("Type", type(an))
print ("Dir ", dir(an))
print ("Type", type(an.x))
print ("Type", type(an.party))
```

which can produce the following output:

```
Type <class '__main__.PartyAnimal '>
Dir ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party', 'x']
Type <class 'int'>
Type <class 'method'>
```

You can notice that we created a new type using the class keyword and in the output of dir that both the integer attribute x and the party method are available in the object.
Life cycle of the object

In the previous examples, we have defined a class (template) and used it to create an instance (object) which we then used until the end of the program, when all the variables were eliminated. Usually we don't think much about the creation and destruction of variables but it often happens that, when our

objects become more complex, we have to act inside the object to fix things while the object is being built and possibly clean things before it is eliminated. If we want our object to become aware of these moments of construction and destruction, we must add methods that have a special denomination:

```
class PartyAnimal:
x = 0
def __init__(self):
print('I am constructed')
def party(self) :
self.x = self.x + 1
print('So far',self.x)
def __del__(self):
print('I am destructed', self.x)


an = PartyAnimal()
an.party()
an.party()
an = 42 print('an contains',an)
```

This program will result in the following output:

```
I am contructed
So far 1
So far 2
I am destructed 2
 an contains 42
```

As soon as Python starts building our object, call our __init__ method to give us the opportunity to set some initial or default values to pass to the object. When Python encounters the line:

```
an = 42
```

In fact, "trash our object" so that we can reuse the variable an to store the value 42. The destructor code (__del__) is called just when our object an is "destroyed". We cannot prevent our variable from being destroyed but we

can do any necessary cleaning before our object no longer exists. During object development it is quite common to add a constructor to an object to set its initial values and it is relatively equally rare that you need to set a destructor.

Inheritance

Another powerful feature of object-oriented programming is the ability to create a new class by extending an existing one. When we extend a class, we call the original class 'parent class' and the new 'child class'. In this example we will move our PartyAnimal class to its file:

```
class PartyAnimal:
x = 0
name = ''
def __init__(self, nam):
self.name = nam
print(self.name,'constructed')
def party(self) :
self.x = self.x + 1
print(self.name,'party count',self.x)
```

So we have the possibility to 'import' the PartyAnimal class into a new file and extend it as follows:

```
from party import PartyAnimal
class CricketFan(PartyAnimal):
points = 0
def six(self):
self.points = self.points + 6
self.party()
print(self.name,"points",self.points)


s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
print(dir(j))
```

In defining the CricketFan object, we indicated that we are extending the PartyAnimal class: all variables (x) and methods (party) of the PartyAnimal class are inherited from the CricketFan class. You can see that within the six method in the CricketFan class we can call the party method from the PartyAnimal class. The variables and methods of the parent class are united in the child class. During the execution of the program we can see that s and j are independent instances of PartyAnimal and CricketFan. Object j has additional capabilities with respect to object s.

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
Jim points 6
['__class__', '__delattr__', ... '__weakref__', 'name', 'party', 'points', 'six', 'x']
```

In the dir output for object j (instance of the CricketFan class) you can see that both have both the attributes and methods of the parent class and the attributes and methods that were added when the class was extended to create the CricketFan class.

CHAPTER 8. Error Handling

A Python program terminates as soon as it encounters an error. In Python, an error can be a syntax error or an exception. In this chapter, you will see what an exception is and how it differs from a syntax error. After that, you will learn about raising exceptions and making assertions. Then, you'll finish with a demonstration of the try and except block.

Exceptions versus Syntax Errors

Syntax errors occur when the parser detects an incorrect statement. Observe the following example:

```
>>> print( 0 / 0 ))
  File "<stdin>" , line 1
    print ( 0 / 0 ))
                 ^
SyntaxError : invalid syntax
```

The arrow indicates where the parser ran into the syntax error. In this example, there was one bracket too many. Remove it and run your code again:

```
>>> print( 0 / 0)
Traceback (most recent call last):
  File "<stdin>" , line 1 , in <module>
ZeroDivisionError : integer division or modulo by zero
```

This time, you ran into an exception error. This type of error occurs whenever syntactically correct Python code results in an error. The last line of the message indicated what type of exception error you ran into.
Instead of showing the message exception erro r , Python details what type of exception error was encountered. In this case, it was a ZeroDivisionErro r . Python comes with various built-in exceptions as well as the possibility to create self-defined exceptions.

Raising an Exception

We can use raise to throw an exception if a condition occurs. The statement can be complemented with a custom exception.

If you want to throw an error when a certain condition occurs using raise, you could go about it like this:

```
x = 10
if x > 5 :
  raise Exception ( 'x should not exceed 5. The value of x was: {} ' . format(x))
```

When you run this code, the output will be the following:

```
Traceback (most recent call last):
 File "<input>" , line 4 , in <module>
Exception : x should not exceed 5. The value of x was: 10
```

The program comes to a halt and displays our exception to screen, offering clues about what went wrong.

The AssertionError Exception

Instead of waiting for a program to crash midway, you can also start by making an assertion in Python. We asser t that a certain condition is met. If this condition turns out to be Tru e , then that is excellent! The program can continue. If the condition turns out to be Fals e , you can have the program throw an AssertionErro r exception.

Have a look at the following example, where it is asserted that the code will be executed on a Linux system:

```
import sys
assert ( 'linux' in sys . platform), "This code runs on Linux only."
```

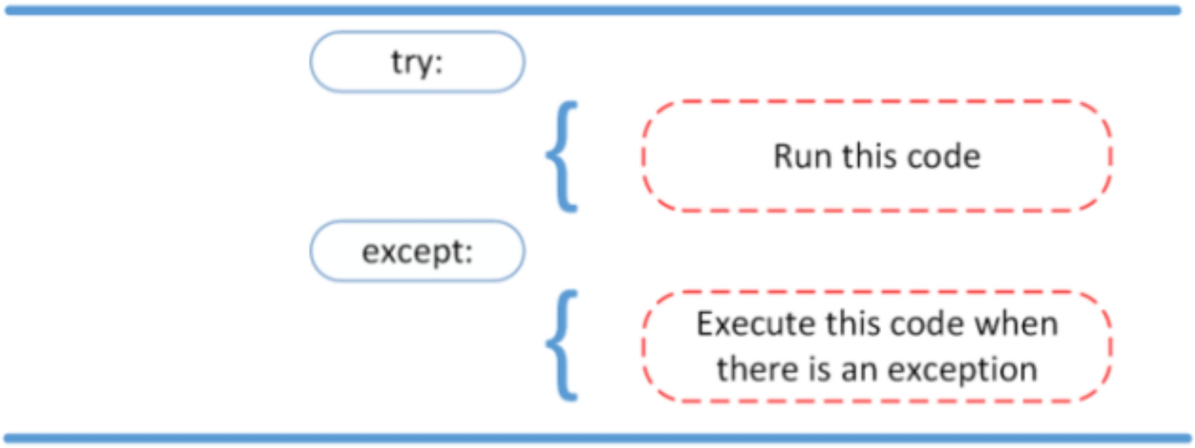
If you run this code on a Linux machine, the assertion passes. If you were to run this code on a Windows machine, the outcome of the assertion would be False and the result would be the following:

```
raceback (most recent call last):
  File "<input>" , line 2 , in <module>
AssertionError : This code runs on Linux only.
```

In this example, throwing an AssertionError exception is the last thing that the program will do. The program will come to halt and will not continue. What if that is not what you want?

The try and except Block: Handling Exceptions

The `try` and `except` block in Python is used to catch and handle exceptions. Python executes code following the `try` statement as a "normal" part of the program. The code that follows the `except` statement is the program's response to any exceptions in the preceding `try` clause.



As you saw earlier, when syntactically correct code runs into an error, Python will throw an exception error. This exception error will crash the program if it is unhandled. The `except` clause determines how your program responds to exceptions.

The following function can help you understand the `try` and `except` block:

```
def linux_interaction():
    assert ( 'linux' in sys . platform), "Function can only run on Linux systems."
    print ( 'Doing something.' )
```

The `linux_interaction( )` can only run on a Linux system. The `assert` in this function will throw an `AssertionError` exception if you call it on an operating system other then Linux.
You can give the function a `try` using the following code:

```
try :
    linux_interaction()
except :
    pass
```

The way you handled the error here is by handing out a pass. If you were to run this code on a Windows machine, you would get the following output:

You got nothing. The good thing here is that the program did not crash. But it would be nice to see if some type of exception occurred whenever you

ran your code. To this end, you can change the pass into something that would generate an informative message, like so:

```
try :
    linux_interaction()
except :
    print ( 'Linux function was not executed' )
```

Execute this code on a Windows machine:

```
Linux function was not executed
```

When an exception occurs in a program running this function, the program will continue as well as inform you about the fact that the function call was not successful.

What you did not get to see was the type of error that was thrown as a result of the function call. In order to see exactly what went wrong, you would need to catch the error that the function threw.

The following code is an example where you capture the AssertionErro r and output that message to screen:

```
try :
    linux_interaction()
except AssertionError as error:
    print (error)
   print ( 'The linux_interaction() function was not executed' )
```

Running this function on a Windows machine outputs the following:

```
Function can only run on Linux systems.
The linux_interaction() function was not executed
```

The first message is the AssertionErro r , informing you that the function can only be executed on a Linux machine. The second message tells you which function was not executed.

In the previous example, you called a function that you wrote yourself. When you executed the function, you caught the AssertionErro r

exception and printed it to screen.
Here's another example where you open a file and use a built-in exception:

```
try :
    with open ('file.log' ) as file:
      read_data = file. read()
except :
    print ('Could not open file.log' )
```

If file.log does not exist, this block of code will output the following:

```
Could not open file.log
```

This is an informative message, and our program will still continue to run. In the Python docs, you can see that there are a lot of built-in exceptions that you can use here. One exception described on that page is the following:

```
Exception FileNotFoundError
Raised when a file or directory is requested but doesn't exist. Corresponds to errno ENOENT.
```

To catch this type of exception and print it to screen, you could use the following code:

```
try :
    with open ( 'file.log' ) as file:
      read_data = file . read()
except FileNotFoundError as fnf_error:
    print (fnf_error)
```

In this case, if file.log does not exist, the output will be the following:

```
[Errno 2] No such file or directory: 'file.log'
```

You can have more than one function call in your tr y clause and anticipate catching various exceptions. A thing to note here is that the code in the tr y clause will stop as soon as an exception is encountered.
Look at the following code. Here, you first call the linux_interaction( ) function and then try to open a file:

```
try :
    linux_interaction()
    with open ( 'file.log' ) as file:
```

```
        read_data = file . read()
except FileNotFoundError as fnf_error:
    print (fnf_error)
except AssertionError as error:
    print (error)
   print ( 'Linux linux_interaction() function was not executed' )
```

If the file does not exist, running this code on a Windows machine will output the following:

```
Function can only run on Linux systems.
Linux linux_interaction() function was not executed
```

Inside the `try` clause, you ran into an exception immediately and did not get to the part where you attempt to open file.log. Now look at what happens when you run the code on a Linux machine:

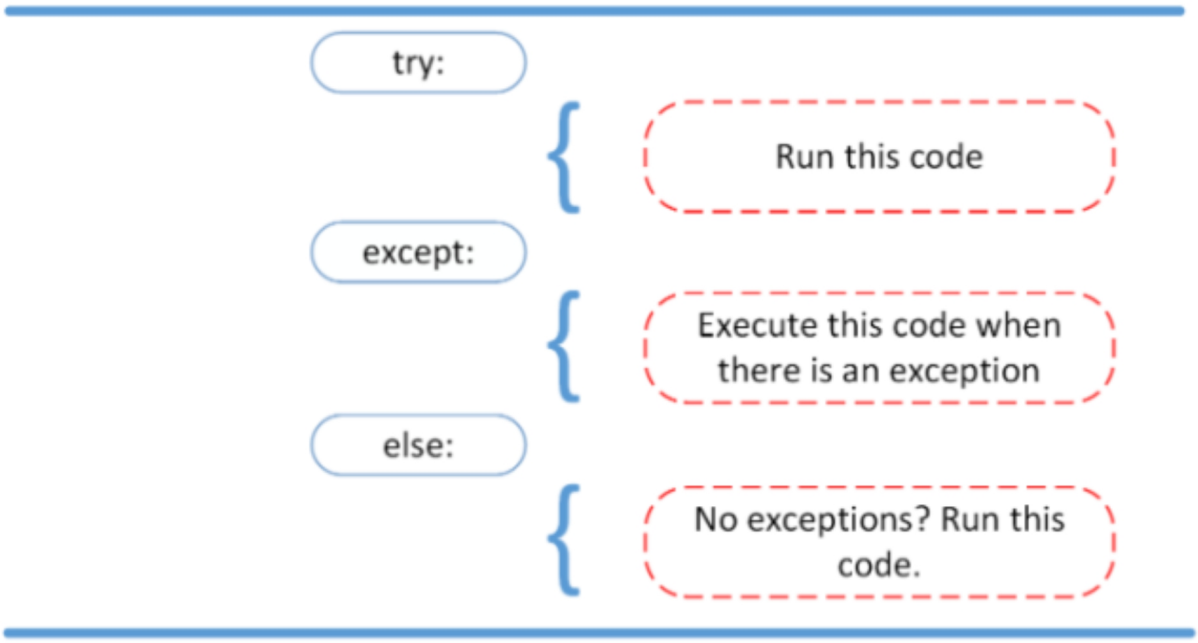
```
[Errno 2] No such file or directory: 'file.log'
```

Here are the key takeaways:

- A `try` clause is executed up until the point where the first exception is encountered.
- Inside the `except` clause, or the exception handler, you determine how the program responds to the exception.
- You can anticipate multiple exceptions and differentiate how the program should respond to them.

The else Clause

In Python, using the `else` statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.



Look at the following example:

```
try :
    linux_interaction()
except AssertionError as error:
    print (error)
else :
    print ( 'Executing the else clause.' )
```

If you were to run this code on a Linux system, the output would be the following:

```
Doing something.
Executing the else clause.
```

Because the program did not run into any exceptions, the `else` clause was executed.

You can also `try` to run code inside the `else` clause and catch possible exceptions there as well:

```
try :
    linux_interaction()
except AssertionError as error:
    print (error)
else :
    try :
```

```
    with open ( 'file.log' ) as file:
        read_data = file . read()
except FileNotFoundError as fnf_error:
    print (fnf_error)
```

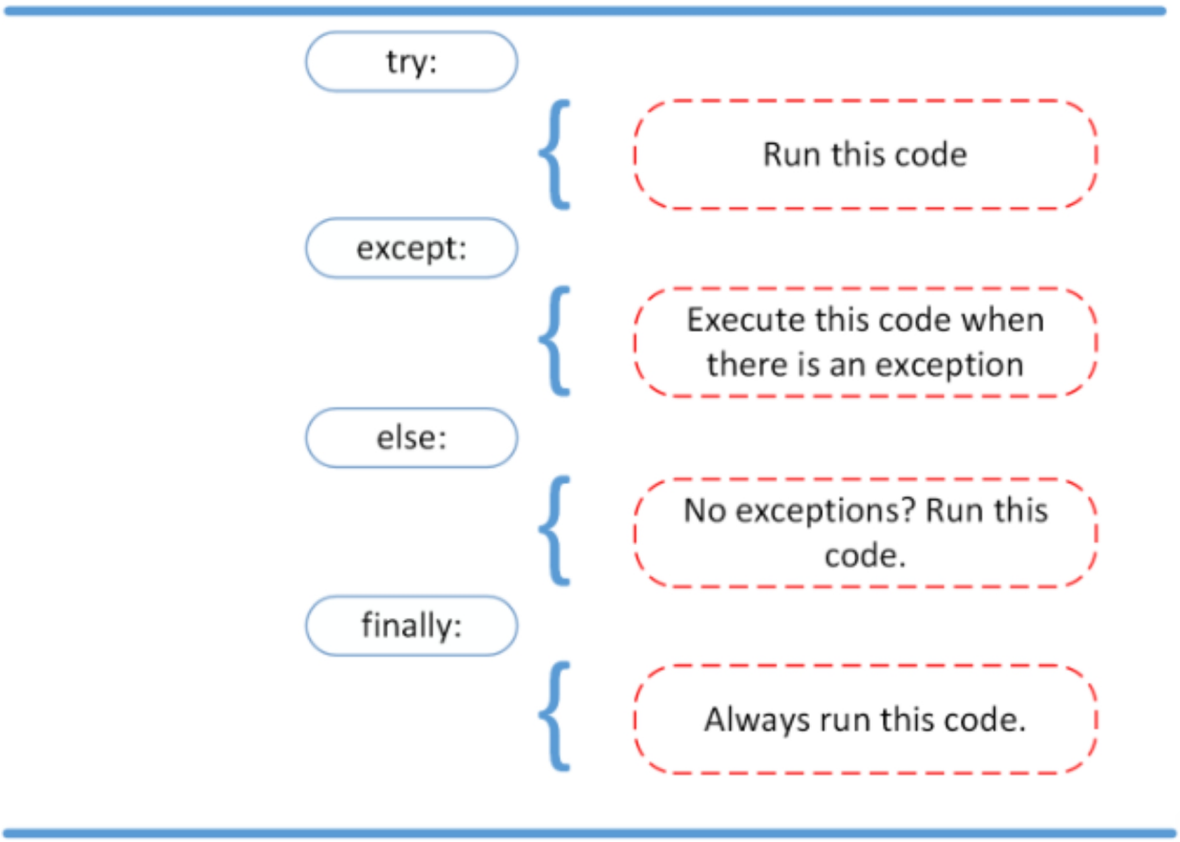
If you were to execute this code on a Linux machine, you would get the following result:

```
Doing something.
[Errno 2] No such file or directory: 'file.log'
```

From the output, you can see that the linux_interaction () function ran. Because no exceptions were encountered, an attempt to open file.log was made. That file did not exist, and instead of opening the file, you caught the FileNotFoundErro r exception.

Cleaning Up After Using finally

Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using the finall y clause.



Have a look at the following example:

```
try :
    linux_interaction()
except AssertionError as error:
    print (error)
else :
    try :
        with open ( 'file.log' ) as file:
            read_data = file . read()
    except FileNotFoundError as fnf_error:
        print (fnf_error)
finally :
   print ( 'Cleaning up, irrespective of any exceptions.' )
```

In the previous code, everything in the finally clause will be executed. It does not matter if you encounter an exception somewhere in the try or else clauses. Running the previous code on a Windows machine would output the following:

```
Function can only run on Linux systems.
```

Cleaning up, irrespective of any exceptions.

Summing Up

After seeing the difference between syntax errors and exceptions, you learned about various ways to raise, catch, and handle exceptions in Python. In this article, you saw the following options:

- `raise` allows you to throw an exception at any time.
- `assert` enables you to verify if a certain condition is met and throw an exception if it isn't.
- In the `try` clause, all statements are executed until an exception is encountered.
- `except` is used to catch and handle the exception(s) that are encountered in the try clause.
- `else` lets you code sections that should run only when no exceptions are encountered in the try clause.
- `finally` enables you to execute sections of code that should always run, with or without any previously encountered exceptions.

Conclusion

The basics of Python programming have been explained in this book. First of all I tried to explain why programming is useful and why (in my opinion) we should all be able to do it. After introducing this concept, in my fundamental opinion, I explained how to install Python on your computer and I introduced you to the basic concepts.

The notions that you have learned in this book are notions extendable to any programming language, cycles, if, objects, error handling are present in many other languages: C ++, Java, SQL, JavaScript, all use these foundations.

Naturally it will change the syntax and the potential of the language, but with a basic knowledge, like the one I have provided you with in this book, you will be able to learn much faster.

I would like to conclude this book by inviting you not to stop at this level of knowledge, but to learn with curiosity as much as possible about the world of programming, because they are useful skills, which you can use both in the professional world and in the personal sphere. Scheduling, automating processes helps us in daily life, allowing us to do many things in a short time.

Happy Coding!