

DEUSDEV

PYNTRODUCTION TO PYTHON



LEARN THE PYTHON BASIC CONCEPTS

Table Of Contents

- Table Of Contents
- Chapter 1: Installation & Setup
 - Step 1: Download the Python installer
 - Step 2: Run the exe file
 - Step 3: Check the Python installation
 - Conclusion
- Chapter 2: Using the terminal and Python interpreter
 - First steps: The Python interpreter
 - Getting help in the Python interpreter
 - Basic Operations in Python
 - Operations with numbers
 - Operations with strings
 - Python variables
 - Conclusion
 - Exercises
- Chapter 3: Using text editors and IDEs
 - Notepad
 - What is an Integrated Development Environment (IDE)?
 - Sublime Text
 - PyCharm (by JetBrains)
 - Visual Studio Code (VS Code)
 - Which text editor should I use?
 - Conclusion
- Chapter 4: Data Types
 - What you will learn
 - Numeric Data Types
 - Integer
 - Float
 - Complex
 - Conversion from one type to another: casting
 - Integer to float
 - Float to integer
 - Integer to complex
 - Float to complex
 - Complex to integer
 - Text Data Type
 - Sequence Data Types
 - List
 - Tuple

- Range
 - Mapping Data Type: dictionary
 - Set Data Types
 - Set
 - Frozen set
 - Boolean Data Types
 - Binary Data Types
 - Bytes
 - Bytearray
 - Memoryview
 - None Data Type
 - Conclusion
 - Exercises
- Chapter 5: Working with strings
 - Python strings: the basics
 - What are Python strings?
 - What is the length of a string?
 - String index notation
 - What is the index of a string?
 - String slice notation
 - Python strings are immutable
 - Format strings
 - Conclusion
 - Exercises
- Chapter 6: Working with lists
 - Python lists
 - What is the length of a list?
 - List index notation
 - Python list slice notation
 - Lists can contain other lists
 - Python lists are mutable
 - Python Lists vs Strings
 - Conclusion
 - Exercises
- Chapter 7: Working with dictionaries
 - Python dictionaries: the basics
 - What is the length of a dictionary?
 - Reading items from a dictionary
 - Editing key-value pairs
 - Adding new key-value pairs
 - Updating key-value pairs
 - Removing elements from a dictionary

- Dictionary key/value data types
 - Converting dictionaries to lists
 - Conclusion
 - Exercises
- Chapter 8: Conditional statements: if, elif, else
 - Revisiting the boolean data type
 - Boolean values and variables
 - Boolean operators: and, or, not
 - Simple decisions: the if statement
 - Flowchart diagrams
 - Python indentation
 - The else statement
 - Nested if-else statements
 - The elif statement
 - Conclusion
 - Exercises
- Chapter 9: Loops, for and while, break and continue
 - For loops
 - Iterating over a tuple
 - Iterating a list
 - Iterating a string
 - Iterating a dictionary
 - Combining for and if statements
 - The range function
 - Nested for loops
 - While loops
 - Infinite loops
 - Control statements
 - The break statement
 - The continue statement
 - The else statement in loops
 - Conclusion
 - Exercises
- Chapter 10: Functions
 - What is a function? The mathematical definition
 - Functions in Python
 - Functions with more than one parameter
 - Functions which returns more than one value
 - Functions with arbitrary data types
 - Default arguments
 - Built-in functions
 - The print function

- The len function
 - The input function
 - All built-in functions
- Conclusion
- Exercises
- Chapter 11: Built-in Methods
 - Python methods
 - String methods
 - **upper**
 - **lower**
 - **count**
 - **find**
 - **index**
 - **replace**
 - **center**
 - **format**
 - List methods
 - **append**
 - **insert**
 - **pop**
 - **remove**
 - **sort**
 - Dictionary methods
 - keys
 - values
 - items
 - update
 - pop
 - Conclusion
 - Exercises
- Chapter 12: Modules
 - Python modules
 - Creating and using a custom module
 - pycache files
 - The random module
 - random.random()
 - random.uniform(a, b)
 - random.randrange(start, stop, step)
 - random.randint(a, b)
 - random.choice(seq)
 - random.shuffle(seq)
 - The math module

- `math.ceil(x)`
 - `math.factorial(n)`
 - `math.gcd()`
 - `math.exp(x)`
 - `math.sin(x)`
 - `math.pi`
 - The numpy module
 - `numpy.array()`
 - `numpy.arange()`
 - `numpy.linspace()`
 - Conclusion
 - Exercises
- Chapter 13: Practice Projects
 - Rock, paper, scissors game
 - Countdown timer and clock

Chapter 1: Installation & Setup

The first step to start programming using Python is to make an installation in your computer. There are websites that offer different ways to work with Python without the need of having anything installed, but in this article you are going to make Python available in your personal computer.

Installing Python is very straightforward, and it takes just a couple of minutes in most cases. Once you have Python installed in your machine, you can start using the most basic but powerful features of this great programming language.

In this article I'm going through one of the various methods available to install Python: downloading the executable installer from the official website. Let's get started.

Step 1: Download the Python installer

The first step is to find the official Python website and look for the installer.

1. Go to the Python official website: <https://www.python.org/>. The Python website looks like the following image.

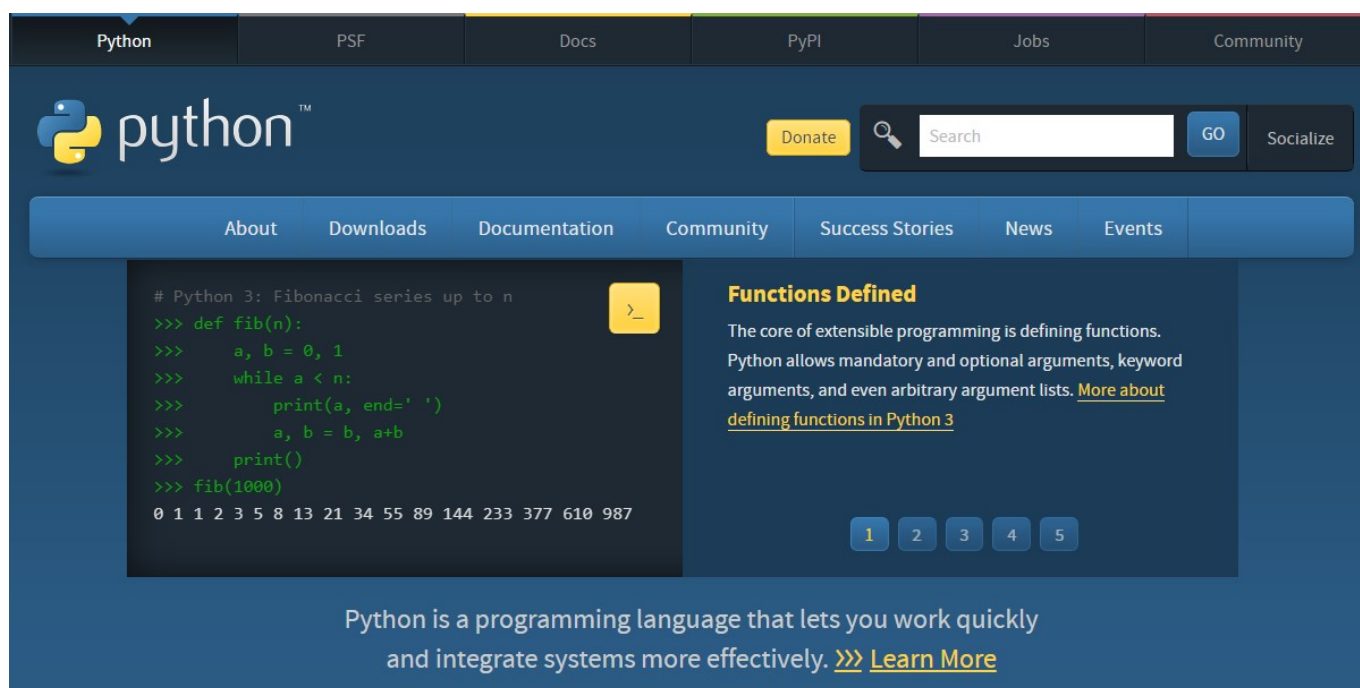


Figure 1. The Python official website.

2. Within this main website, find the Downloads section and click on it. If you are on Windows, the website will show the download version for that operating system. Each operating system has a different download source. You should find the one for your current operating system (Windows, Linux, macOS, etc).

If you are a sufficient curious person, you may have already scrolled down the page and noticed that there are different Python versions you can download and install. You can install any Python version you want, and in fact you can have more than one Python version installed in your computer. This is outside the scope of this tutorial. Instead, we will focus on working with the latest stable Python release.

3. Go ahead and download the latest release for your operating system. For my case and for the time of writing this article, I have to download the Python 3.11.1 version for Windows. When you click on the link, it will download an exe file with a size of about 27 Mb.

Step 2: Run the exe file

1. Open the exe file. There is a chance that a pop up window will show up asking for permissions. You will see a window like the following (the version of the installable will be different than the one in the image):

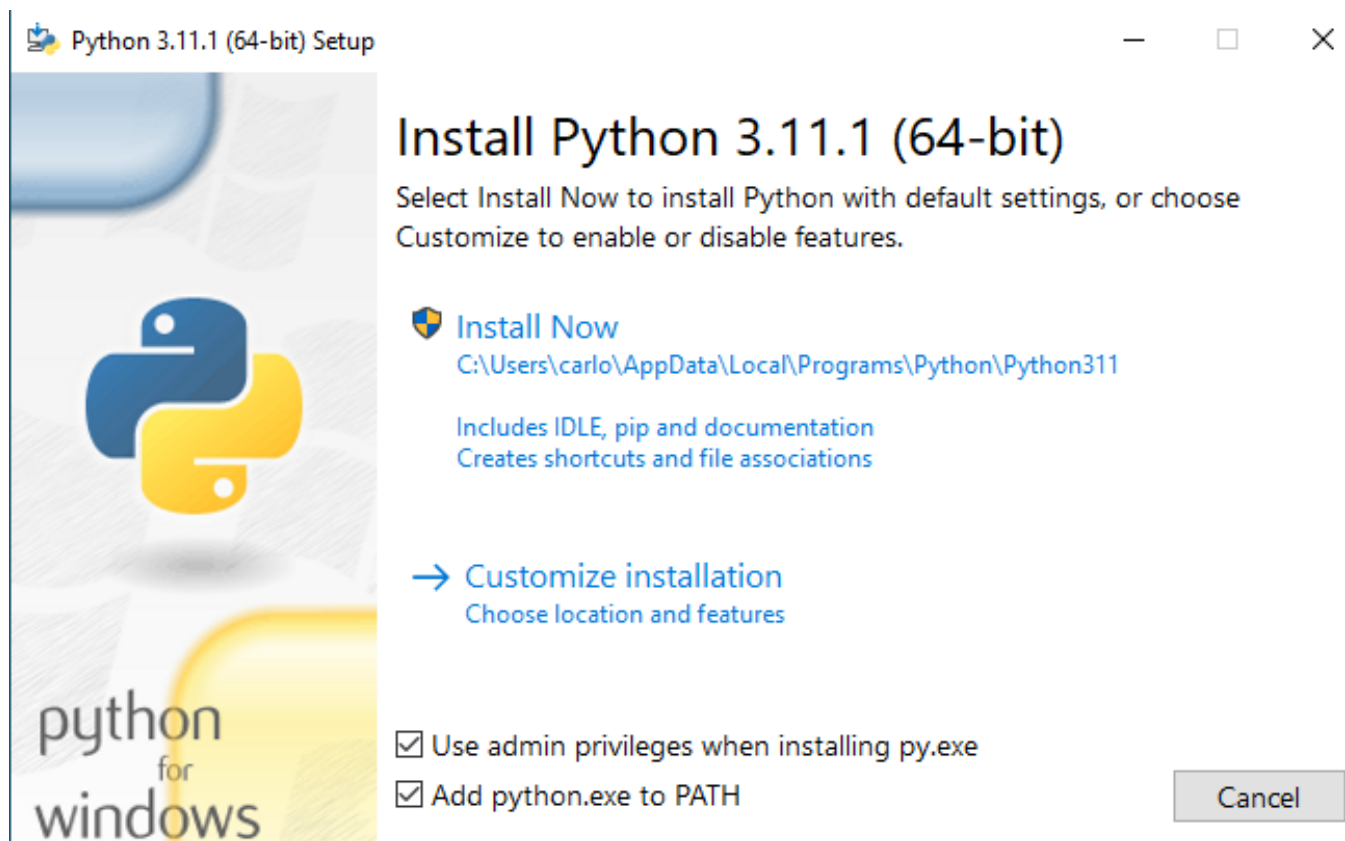


Figure 2. The first Python installation screen.

- The Customize installation section gives a set of options to choose, like including pip in the installation (this is included by default).
- I recommend you mark the option that says Add Python to PATH. I'm not going to discuss the implications of doing this or not, but I encourage you to investigate on this to know at least what that is. For more information about PATH, you can read the discussion on this link: <https://discuss.python.org/t/could-we-add-python-to-system-path-by-default/3067>

2. Click install now. Wait for the files to finish installing.

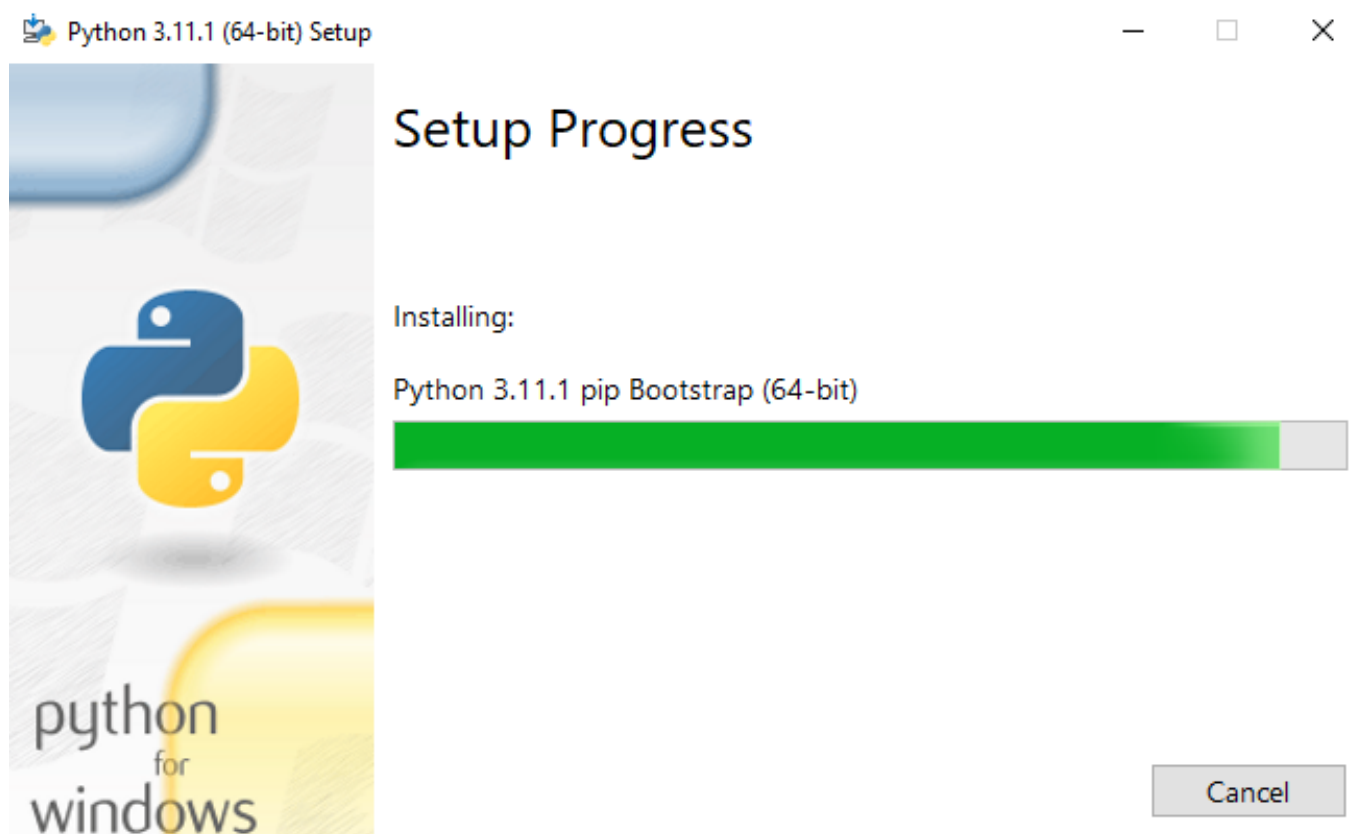


Figure 3. Python installation in progress.

3. Once the installation finishes, this window will show. As you can see there are a couple of useful links like official tutorials and documentation.

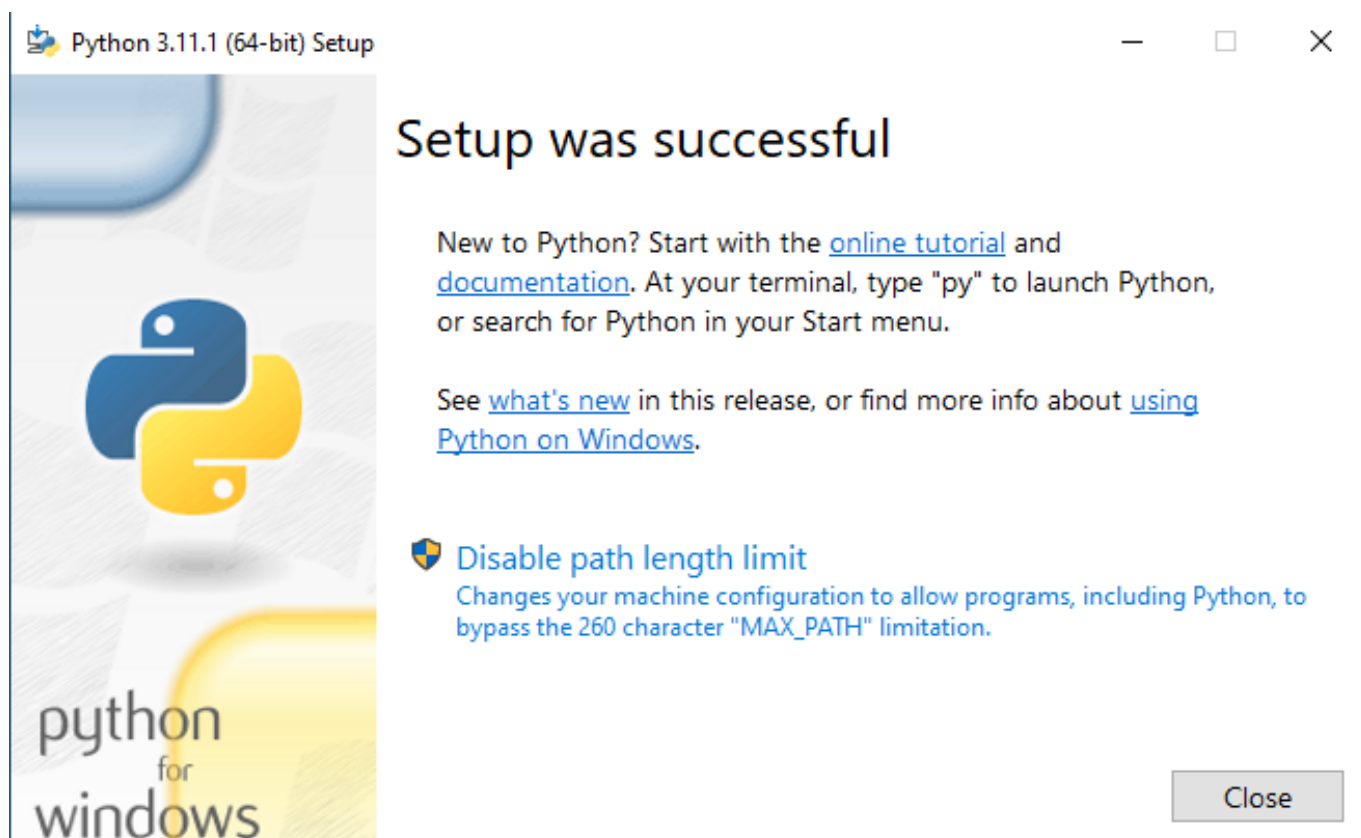


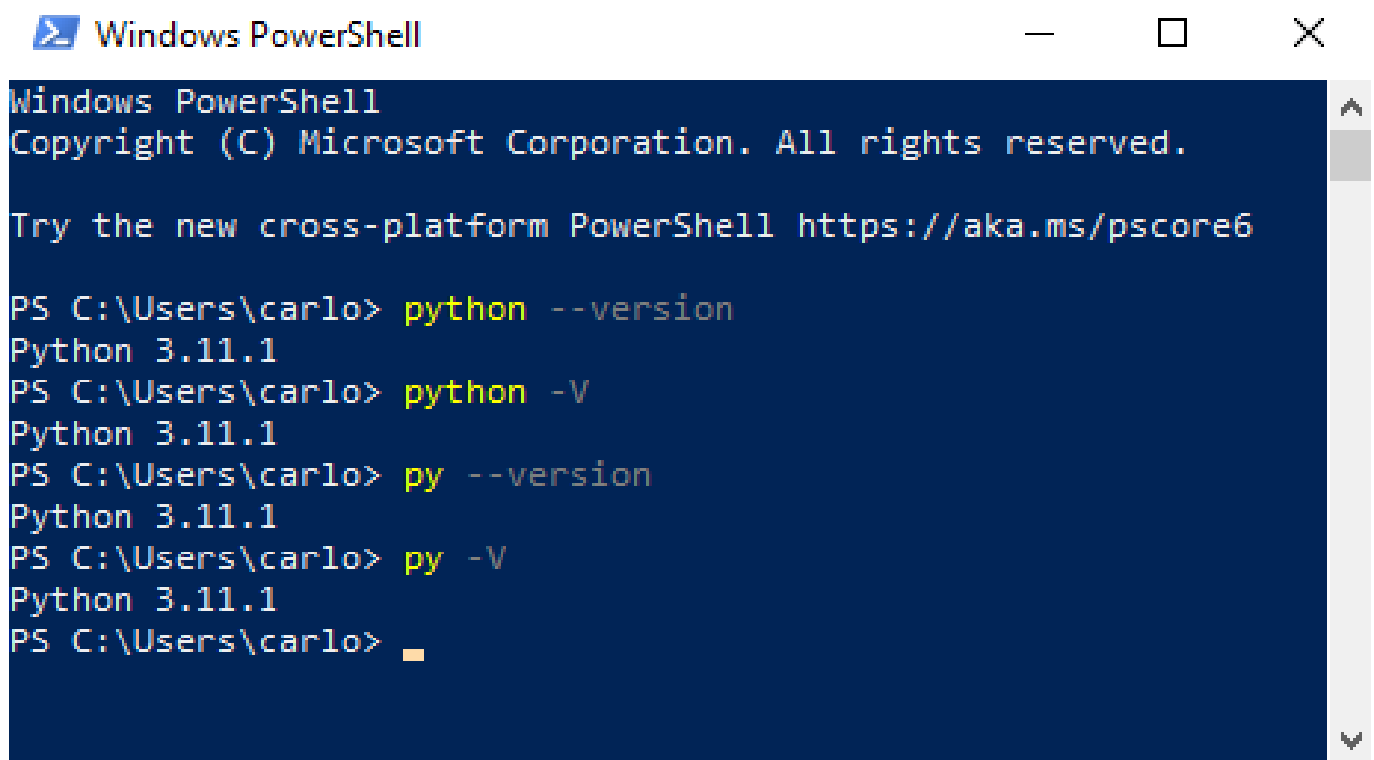
Figure 4. Python installation finishing.

Step 3: Check the Python installation

Python is now installed on your computer, but you may want to make sure that everything went fine with the process. You can check your Python installation by running a first Python command on the terminal.

1. If you are on Windows, open [Windows PowerShell](#) (you may have to install this depending on your system).
2. Write `python --version` (double hyphen) and hit Enter. Alternatively you can write `python -V` (with a single hyphen). This command is equivalent. You can also write `py --version` or `py -V`.

In any case, you should see the current Python version as the output:

A screenshot of a Windows PowerShell window. The title bar says 'Windows PowerShell' with standard window controls. The terminal text is as follows:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\carlo> python --version
Python 3.11.1
PS C:\Users\carlo> python -V
Python 3.11.1
PS C:\Users\carlo> py --version
Python 3.11.1
PS C:\Users\carlo> py -V
Python 3.11.1
PS C:\Users\carlo> _
```

Figure 5. First Python code in Windows Powershell.

As you can see the terminal prints the current Python version installed on your machine. This means everything went good and you are ready to use Python to write your programs.

Conclusion

In this article you learned how to install Python on your computer. Although I assume you are using Windows, you can also install Python on your preferred operating system.

You can find much more information in [this link](#). This is the official Python documentation on how to use Python on Windows. If you are on a mac, you can follow [this link](#) instead.

In the next article you are going to learn how to work with Python and write your first programs.
See you next time!

Chapter 2: Using the terminal and Python interpreter

Python can be used directly from the terminal, without the need of using [text editors](#) like Notepad, Sublime or VsCode. Using the terminal is a good practice when making the first steps to learn Python. In this article you are going to learn how to run basic Python commands using the terminal (PowerShell).

In the [previous article](#) you went through the installation and setup of Python. You already used PowerShell to check the Python version by using `python --version`. Check it out if you missed it.

First steps: The Python interpreter

Let's start using PowerShell right now. Go ahead and open a new terminal (PowerShell). The first thing you will see on the screen is something like the following:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\yourusername>
```

As you can see, you have an initial message with Copyright and some extra information. If you are using an operating system different than Windows, you may see your terminal in some other format.

The last line is the place where you can enter different commands. **yourusername** will be your computer user name. At the end of that line you should see a blinking underscore, meaning it is ready to take a written input from the user. For example, you could write `pwd` to print the current working directory:

```
PS C:\Users\yourusername> pwd

Path
----
C:\Users\yourusername
```

When you write `pwd` and hit Enter, the terminal shows the current working directory on the screen. There are a lot of useful commands you can try besides `pwd`. But let's work with the one

we are interested in: `python`. Go ahead and write `python` and hit Enter:

```
PS C:\Users\yourusername> python
Python 3.11.2 (tags/v3.11.2:878ead1, Feb  7 2023, 16:38:35) [MSC v.1934
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The first thing you see after the command you just entered is the Python version and some extra information (never mind that now). The next thing you see is a couple of suggestions to try next. They are help, copyright, credits and license.

Lastly you may have noticed that there are three greater-than symbols `>>>` before the blinking underscore. This is how the Python interpreter looks like. This means that you are ready to enter Python commands. This is called [interactive mode](#).

Getting help in the Python interpreter

Go ahead and try `help`, one of the suggested commands. Write the command and hit Enter.

```
>>> help
Type help() for interactive help, or help(object) for help about
object.
```

Seems like you need to include a set of parentheses after `help` according to the message you just got. Let's do that now.

```
>>> help()

Welcome to Python 3.11's help utility!

If this is your first time using Python, you should definitely check
out
the tutorial on the internet at https://docs.python.org/3.11/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility
and
return to the interpreter, just type "quit".
```

To get a list of available modules, keywords, symbols, or topics, type "modules", "keywords", "symbols", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose name or summary contain a given string such as "spam", type "modules spam".

```
help>
```

Writing `help()` (with parentheses at the end) on the Python prompt and hitting Enter shows something very interesting! First of all it is giving you a warm welcome to the Python help utility. Right below it suggests to check out the official tutorial for the current version.

Read everything. Now is the time for me to suggest you a couple of things you should do that I did not when learning Python. Read everything in detail even if you don't understand it. For example if you keep reading the message given by the Python interpreter, you will find terms like *module*, *keyword* and *string*. You should not have to know their meaning yet, but is good to have those words ringing in your head. You will learn those in given time.

Lastly, notice that the prompt changed again. Now it says `help>` where you can write your search. Let's try writing "modules" as suggested, just to see what happens.

```
help> modules
```

```
Please wait a moment while I gather a list of all available modules...
```

```
test_sqlite3: testing with version '2.6.0', sqlite_version '3.39.4'  
C:\Users\yourusername\AppData\Local\Programs\Python\Python311\Lib\site-  
packages\_distutils_hack\__init__.py:33: UserWarning: Setuptools is  
replacing distutils.
```

```
warnings.warn("Setuptools is replacing distutils.")
```

<code>__future__</code>	<code>_testmultiphase</code>	<code>genericpath</code>	<code>runpy</code>
<code>__hello__</code>	<code>_thread</code>	<code>getopt</code>	<code>sched</code>
<code>__phello__</code>	<code>_threading_local</code>	<code>getpass</code>	<code>secrets</code>
<code>_abc</code>	<code>_tkinter</code>	<code>gettext</code>	<code>select</code>
<code>_aix_support</code>	<code>_tokenize</code>	<code>glob</code>	<code>selectors</code>
<code>_ast</code>	<code>_tracemalloc</code>	<code>graphlib</code>	<code>setuptools</code>
<code>_asyncio</code>	<code>_typing</code>	<code>gzip</code>	<code>shelve</code>
<code>_bisect</code>	<code>_uuid</code>	<code>hashlib</code>	<code>shlex</code>
<code>_blake2</code>	<code>_warnings</code>	<code>heapq</code>	<code>shutil</code>
<code>_bootsubprocess</code>	<code>_weakref</code>	<code>hmac</code>	<code>signal</code>
<code>_bz2</code>	<code>_weakrefset</code>	<code>html</code>	<code>site</code>
<code>_codecs</code>	<code>_winapi</code>	<code>http</code>	<code>smtpd</code>

_codecs_cn	_xxsubinterpreters	idlelib	smtplib
_codecs_hk	_zoneinfo	imaplib	sndhdr
_codecs_iso2022	abc	imghdr	socket
_codecs_jp	aifc	imp	
socketserver			
_codecs_kr	antigravity	importlib	sqlite3
_codecs_tw	argparse	inspect	sre_compile
_collections	array	io	
sre_constants			
_collections_abc	ast	ipaddress	sre_parse
_compat_pickle	asynchat	itertools	ssl
_compression	asyncio	json	stat
_contextvars	asyncore	keyword	statistics
_csv	atexit	lib2to3	string
_ctypes	audioop	linecache	stringprep
_ctypes_test	base64	locale	struct
_datetime	bdb	logging	subprocess
_decimal	binascii	lzma	sunau
_distutils_hack	bisect	mailbox	symtable
_elementtree	builtins	mailcap	sys
_functools	bz2	marshal	sysconfig
_hashlib	cProfile	math	tabnanny
_heapq	calendar	mimetypes	tarfile
_imp	cgi	mmap	telnetlib
_io	cgitb	modulefinder	tempfile
_json	chunk	msilib	test
_locale	cmath	msvcrt	textwrap
_lsprof	cmd	multiprocessing	this
_lzma	code	netrc	threading
_markupbase	codecs	nntplib	time
_md5	codeop	nt	timeit
_msi	collections	ntpath	tkinter
_multibytecodec	colorsys	nturl2path	token
_multiprocessing	compileall	numbers	tokenize
_opcode	concurrent	opcode	tomllib
_operator	configparser	operator	trace
_osx_support	contextlib	optparse	traceback
_overlapped	contextvars	os	tracemalloc
_pickle	copy	pathlib	tty
_py_abc	copyreg	pdb	turtle
_pydecimal	crypt	pickle	turtledemo
_pyio	csv	pickletools	types

_queue	ctypes	pip	typing
_random	curses	pipes	unicodedata
_sha1	dataclasses	pkg_resources	unittest
_sha256	datetime	pkgutil	urllib
_sha3	dbm	platform	uu
_sha512	decimal	plistlib	uuid
_signal	difflib	poplib	venv
_sitebuiltins	dis	posixpath	warnings
_socket	distutils	pprint	wave
_sqlite3	doctest	profile	weakref
_sre	email	pstats	webbrowser
_ssl	encodings	pty	winreg
_stat	ensurepip	py_compile	winsound
_statistics	enum	pyclbr	wsgiref
_string	errno	pydoc	xdrlib
_strptime	faulthandler	pydoc_data	xml
_struct	filecmp	pyexpat	xmlrpc
_symtable	fileinput	queue	xxsubtype
_testbuffer	fnmatch	quopri	zipapp
_testcapi	fractions	random	zipfile
_testconsole	ftplib	re	zipimport
_testimportmultiple	functools	reprlib	zlib
_testinternalcapi	gc	rlcompleter	zoneinfo

Enter any module name to get more help. Or, type "modules spam" to search for modules whose name or summary contain the string "spam".

Looks overwhelming! So, it is listing all of the available modules you have (you will see what [Python modules](#) are later in this course). You may have a different list with fewer modules. Let's try "keywords" to see what you get.

```
help> keywords
```

Here is a list of the Python keywords. Enter any keyword to get more help.

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return

as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield

A shorter list now. Let's try entering a keyword from this list to see what help you get.

```
help> for
The "for" statement
*****
```

The "for" statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for_stmt ::= "for" target_list "in" expression_list ":" suite
          ["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the "expression_list". The suite is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see Assignment statements), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a "StopIteration" exception), the suite in the "else" clause, if present, is executed, and the loop terminates.

A "break" statement executed in the first suite terminates the loop without executing the "else" clause's suite. A "continue" statement executed in the first suite skips the rest of the suite and continues with the next item, or with the "else" clause if there is no next item.

The for-loop makes assignments to the variables in the target list. This overwrites all previous assignments to those variables including those made in the suite of the for-loop:

```
for i in range(10):
    print(i)
```

```
i = 5          # this will not affect the for-loop
               # because i will be overwritten with the
next
               # index in the range
```

Names in the target list are not deleted when the loop is finished, but if the sequence is empty, they will not have been assigned to at all by the loop. Hint: the built-in function "range()" returns an iterator of integers suitable to emulate the effect of Pascal's "for i := a to b do"; e.g., "list(range(3))" returns the list "[0, 1, 2]".

Related help topics: break, continue, while

Very nice! We get a lot of helpful information about this *for* keyword. If you read it now you may not understand many things, but we will cover this important keyword later on this course. Let's be patient!

Let's quit the help section to try some other Python commands (you can write **quit** or do CTRL+C).

```
help> quit
```

You are now leaving help and returning to the Python interpreter. If you want to ask for help on a particular object directly from the interpreter, you can type "help(object)". Executing "help('string')" has the same effect as typing a particular string at the help> prompt.

```
>>>
```

Remember that the three **>>>** signs means you are now on the Python interpreter again. When quitting the help interpreter you get a message indicating that you can use the help directly on the Python interpreter. Go ahead and give it a try.

Basic Operations in Python

You want to learn Python and you may have already guessed that Python can make calculations fast. You can make many different operations with different data types in Python. In this section you are going to learn the very basics of Python operations.

Operations with numbers

Let's see how to make some simple math with the Python interpreter. Let's start simple and try addition, subtraction, multiplication and division.

```
>>> 1+1
2
>>> 3-1
2
>>> 2*3
6
>>> 12/3
4.0
```

The symbols for addition and subtraction are `+` and `-`, as you may have already guessed. The symbol for multiplication is an asterisk `*`, and for division you use a single forward slash `/`. The four operations you tried on this example are giving correct results.

If you look closer on the last example, the answer is shown as **4.0** instead of just **4**. This has something to do with [floating point numbers](#) (more on this in the future).

You can also do more complex calculations, like combined operations. If you need to group operations, use parentheses `()`. You can even use multiple parentheses if needed.

```
>>> 1-8+3
-4
>>> 1-(8+3)
-10
>>> 1-8+3/4
-6.25
>>> 1/8+(3/4)*5
3.875
>>> (1/8+(3*5-1))
14.125
```

There are other useful operations you can use in Python. The most common are modulus (remainder), exponentiation and floor division operators.

```
>>> 10%3
1
>>> 2**3
8
```

```
>>> 10//3
3
```

Each of these operations can be summarized as:

- **modulus** (%) returns the remainder of the division between two numbers.
- **exponentiation** (**, double asterisk) returns the result of multiplying the number on the left by itself a number of times given by the number on the right.
- **floor division** (//) returns the integer part of the division between two numbers.

Operations with strings

Another type of data you can play with before moving on are *strings*. A string is a sequence of characters surrounded by single, double or triple quotes. Within the Python interpreter, try writing different strings and hit Enter to see the output.

```
>>> 'hello in single quotes'
'hello in single quotes'
>>> 'hello in double quotes'
File "<stdin>", line 1
    'hello in double quotes'
    ^^^^^
SyntaxError: invalid syntax
>>> "hello in double quotes"
'hello in double quotes'
>>> '''hello in triple quotes'''
'hello in triple quotes'
```

The first thing we tried is writing some text between single quotes and hitting Enter. The terminal outputs the same thing as expected. Next we tried the same string using double quotes, but the double quotes are used with two single quotes, which Python doesn't like (it gives a **SyntaxError**. More on errors later). To use double quotes you have to use the special key (it is often done with SHIFT+2 on the keyboard). And lastly triple quotes, done with three single quotes.

A curious thing to note is that no matter if you use single, double or triple quotes, the output is shown with single quotes. What if you need to include quotes in the string? You can use single quotes for our Python string, and use double quotes inside, or the other way around.

```
>>> 'hello, this "word" has quotes'
'hello, this "word" has quotes'
```

```
>>> "hello, this 'word' has quotes"
"hello, this 'word' has quotes"
```

So, the title of this section contains the word "operations". Can you "operate" with strings like you did with numbers? You can concatenate strings with the `+` operator:

```
>>> 'hello'+'world'
'helloworld'
>>> 'hello'+ ' '+'world'
'hello world'
```

You can also use the `*` operator to repeat the same string a given number of times:

```
>>> 'hello'*3
'hellohellohello'
```

As you can see, Python strings can also be manipulated with some of the basic operations you learned so far.

Python variables

The last topic on this section will be about the basics of using variables in Python. Variables are a way to store values for later use. For example you may need to use your name as a string multiple times in your program or script.

Let's begin by declaring the variable `myName` and assigning it the value of the string 'Carlos'. Then write `myName` and hit Enter to print the value inside of it:

```
>>> myName = 'Carlos'
>>> myName
'Carlos'
```

Declaring and assigning values to variables in Python is as simple as that. You don't have to worry about which data type the variable will be.

You can now concatenate a string 'Hello' with 'Carlos' using the variable `myName`:

```
>>> 'Hello ' + myName
'Hello Carlos'
```

The same thing can be done with numbers. Suppose you want to compute the discriminant of a quadratic equation. For this you can use three variables for each parameter **a**, **b** and **c**:

```
>>> a = 3
>>> b = -7
>>> c = 11
>>> b**2 - 4*a*c
-83
```

Using variables in Python is a great way to write better, reusable and more readable code. Variables are used everywhere as you will see in future chapters of this course.

To quit the Python interpreter you can use **quit()** (don't forget the parentheses like I did before):

```
>>> quit
Use quit() or Ctrl-Z plus Return to exit
>>> quit()
PS C:\Users\yourusername>
```

Conclusion

In this article you learned the basics of Python by using the terminal (PowerShell) and the Python interpreter. Here is a list of what you have learned:

- How to get help from the Python interpreter with the use of the **help()** command.
- How to make simple operations with numbers.
- How to make operations with strings.
- How to define and make simple calculations with variables.

After closing the Python interpreter, all the work you did is lost. What about if you needed to continue the work you've done at another time? For this you can use text editors.

In the next chapter you will learn more advanced ways to work with Python. You will use the topics learned on this article, so be sure you are comfortable with them. See you next time!

Exercises

This set of exercises are here to help you better understand the topics covered on this article.

1. Open the terminal (PowerShell).
2. Check that Python is correctly installed by showing the version on the screen.
3. Enter the Python interpreter.
4. Find the help document about STRINGS. There are two ways of getting to it: one from the help interpreter, and the other directly from the Python interpreter.
5. Make the following calculation using the Python interpreter: $6/2(2+1)$. Is the result correct or not? How can you be sure about the order of operations in a general case?
6. Make the following calculation with the Python interpreter: $\text{sqrt}(-4)=(-4)^{0.5}$. A number with an exponent of 0.5 is the same as taking its square root. What can you guess about the output Python is giving? You will learn more about this in future chapters.
7. Use your own name to print a custom welcoming message on the screen with the use of strings.
8. Combine your name and a friend's to print a message containing both. Use two variables to hold your names.
9. Use variables to calculate how many seconds had been gone within the present day. Make a variable to hold the hours, another for minutes, and another for seconds. For example if the current time is 16:33:21 then you can assign 16 to hours, 33 to minutes and 21 to seconds.
10. Quit the Python interpreter.

Chapter 3: Using text editors and IDEs

Although Python can be used directly from a terminal like the Windows PowerShell, in many cases the programs can get so long and complex that the use of a text editor is necessary. Even if the program is not too long in extension, using a text editor can come in handy in many cases.

In this article you are going to see how to run Python programs written in text editors, and which editors are the most common in the industry.

Notepad

The first text editor you can try is one of the most common and popular in Windows: Notepad. This text editor is installed in Windows by default, so you don't need to install any extra software for this case. Go ahead and open the Notepad software. Let's also open the Windows PowerShell too, because the terminal is needed to execute the program you will write in the text editor.

In Notepad, write some Python code to test. I will just write `1+1` to start testing something really simple.

```
1+1
```

Let's save the file and give it a name. For example you can give the file a name like **firstPythonProgram.txt**. Note that the extension `.txt` is the default when working with notepad. But in order for Python to be able to run the program, the file needs to have a different file extension. In the case of Python programs the extension is **.py**, so the file should be named as **firstPythonProgram.py**. You can save the file on the Desktop or inside a new folder named **DeusDevExercises**, or the name you wish.

Changing file extensions on Windows. Changing a file extension on Windows is not trivial, specially if you don't have much experience working with computer programs. If you don't know how to change the file extension in Windows you can read [this article](#).

To run the program you wrote on the text file, go to the terminal and write `python` followed by the name of the file. This will not open the Python interpreter like it used to when writing just `python`. This command will instead execute the program written in the file. Open a Windows PowerShell (terminal) and run the following:

```
PS C:\Users\youruser> cd Desktop/DeusDevExercises
PS C:\Users\youruser\Desktop\DeusDevExercises> python
```



```
firstPythonProgram.py
PS C:\Users\youruser\Desktop\DeusDevExercises>
```

First you need to *change directory* to where the text file is located. Use the **cd** (*change directory*) command to move to a different directory, for example **Desktop/DeusDevExercises** if you created the folder in the Desktop (it is possible that you need to look for a folder named **OneDrive** before **Desktop** depending on the version of Windows).

As you can see, after running **python firstPythonProgram.py** nothing happened. This is because working with files is different than working directly on the Python interpreter. With the Python interpreter the output is shown automatically, but when working with files you have to explicitly tell the script to *print* the output to the screen.

This is where the **print()** function comes in. Edit the Python file to look like the following:

```
print(1+1)
```

Execute the script again in PowerShell, and you will see the output printed on the screen.

```
PS C:\Users\youruser\Desktop\DeusDevExercises> python
firstPythonProgram.py
2
```

Amazing! Let's write a more complex script, using a variable to print a custom message with your own name:

```
myName = 'Carlos'
print('Hello ' + myName)
```

Run the script on the terminal:

```
PS C:\Users\youruser\Desktop\DeusDevExercises> python
firstPythonProgram.py
Hello Carlos
```

As you can see the scripts are working as expected. This way of working with programs is very helpful in case you need to edit the variable **myName** for example. You only need to edit that line

and the rest of the script remains unchanged.

You can use Notepad for all your programs, but there are other excellent editors that are specifically designed to work with different programming languages like Python. These are called **IDEs**.

What is an Integrated Development Environment (IDE)?

An Integrated Development Environment (IDE) is a software application specifically designed to work in software development. An IDE usually has different features which makes coding easier, and that other common text editors like Notepad doesn't.

IDEs consist of some sort of text editor, building automation and debuggers. Some IDEs include extra features like syntax highlighting (colored code), autocomplete, specific code language integration, and many more.

You will learn all of this concepts along the way. For now let's see some examples of popular IDEs and try to pick one or two to keep working with Python.

Sublime Text

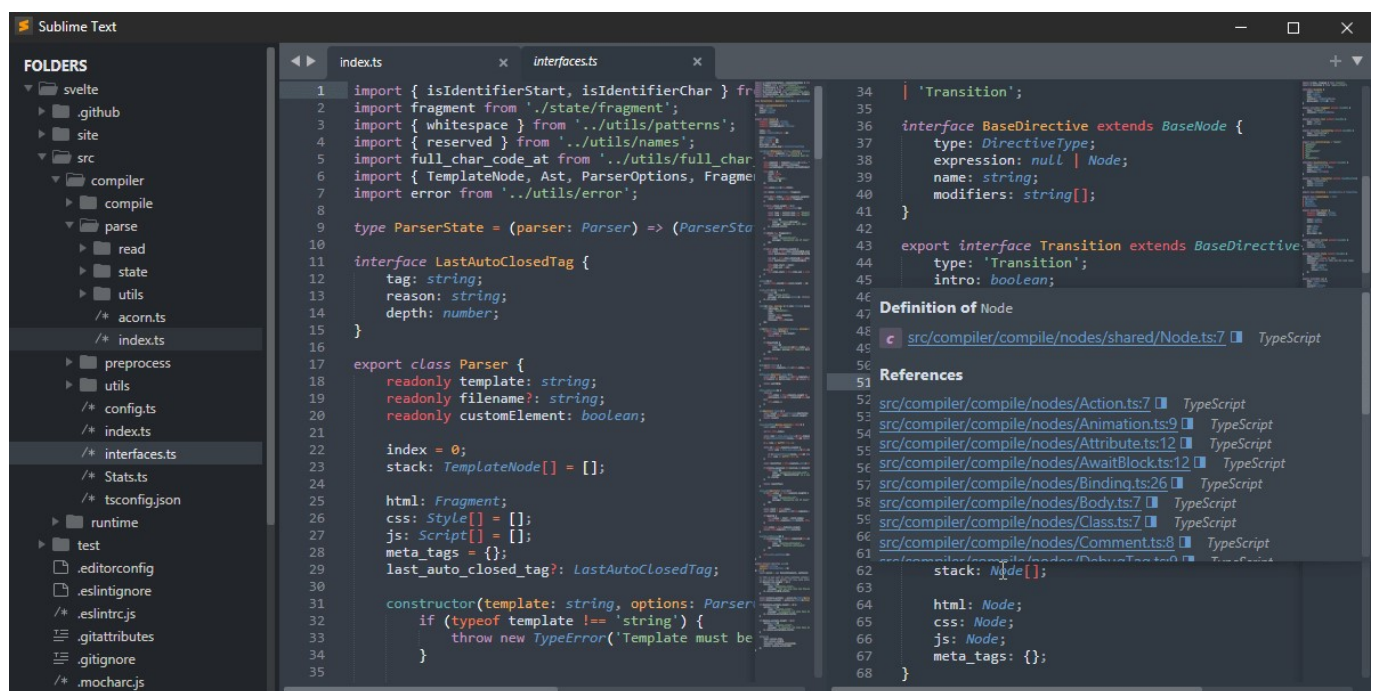


Figure 1. Sublime text editor

Sublime text is a text editor similar to Notepad. It has many features that help developers code in a more comfortable way. Some of them are:

- Files and folder organization: you can open multiple files, and even a complete folder with files and subfolders inside of it.
- Colored code according to the language: sublime has many different color combinations to improve code readability based on the programming language used (Python, C, Ruby,

etc).

- Packages: you can use different packages or plugins to make your sublime text editor more customizable.

PyCharm (by JetBrains)

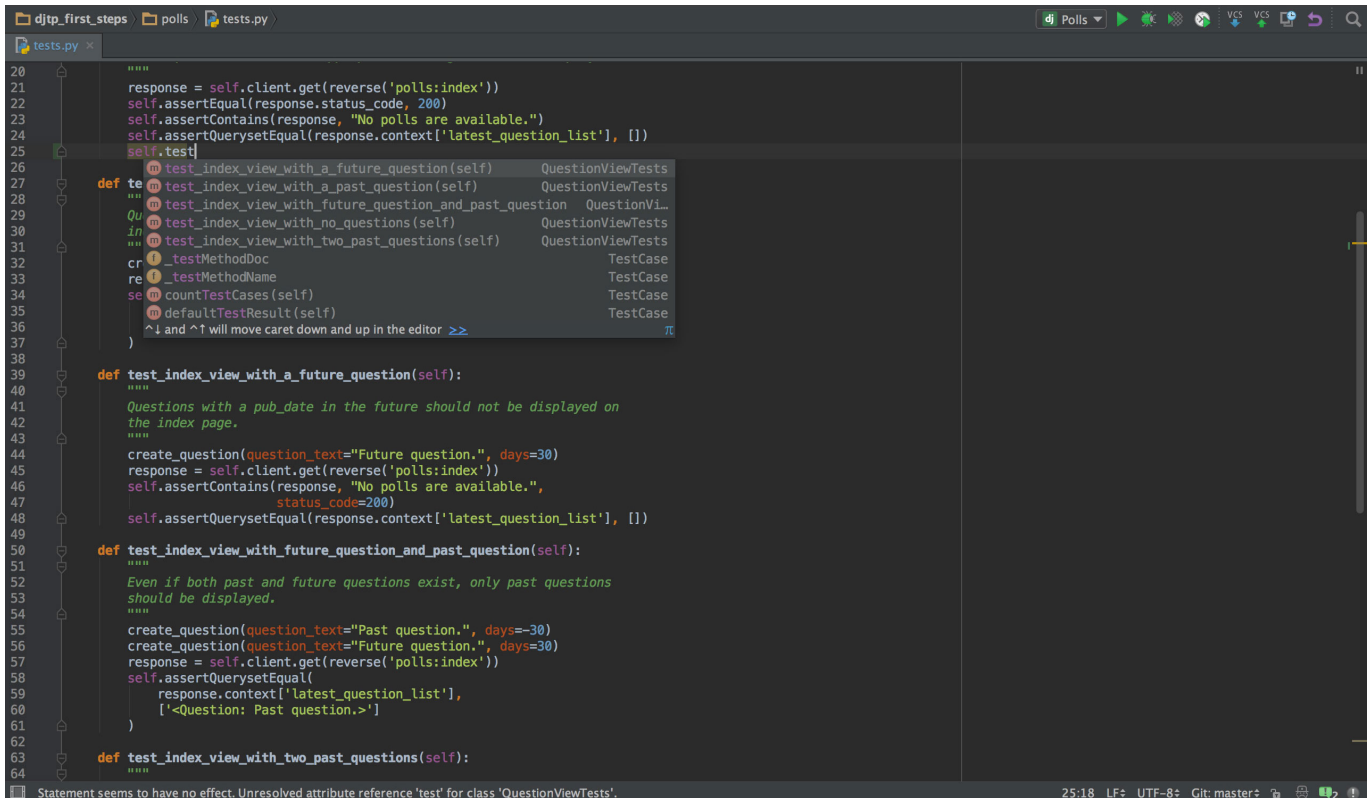


Figure 2. PyCharm text editor

Another very popular IDE is [PyCharm](#). This IDE is specific for Python, which makes it a very good choice if you intend to work solely with this programming language. Some key of its features are:

- Syntax highlighting to improve code readability.
- Intelligent code autocompletion and suggestions.
- Folder and files structure for ease of navigation.

Visual Studio Code (VS Code)

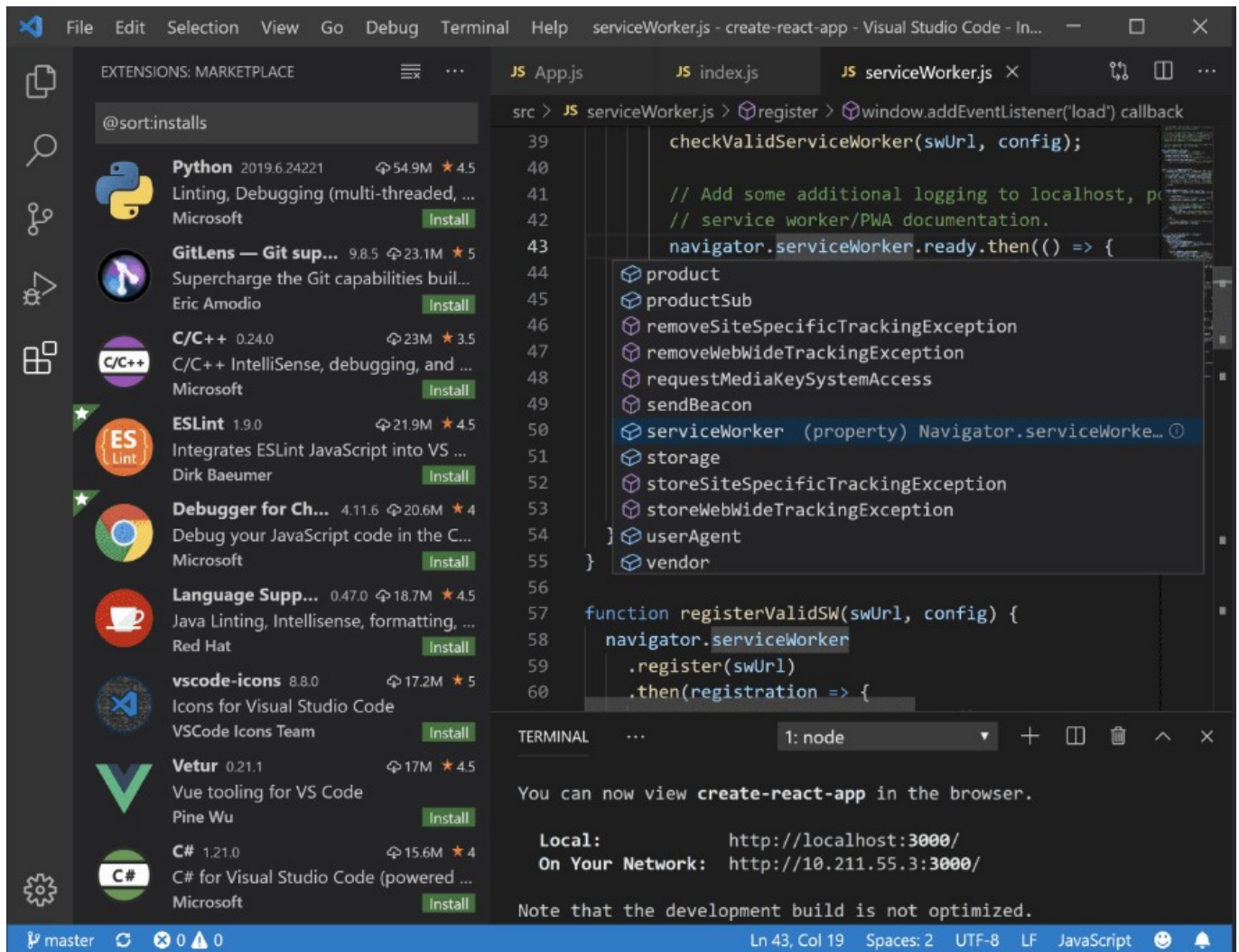


Figure 3. VS Code text editor

One of the most popular and used code editors is [VS Code](#). This code editor has many features that makes working with code a lot easier compared with common text editors like Notepad. Some of them are:

- Files and folder organization (similar to Sublime and PyCharm)
- Colored code according to language (similar to Sublime and PyCharm)
- VS Code works with something called *extensions*. For example, you can download the Python extension when working with Python programs.
- Customization: you can customize the editor to your needs, being the layout, colors and more.
- IntelliSense: VS Code has autocomplete and suggestions that makes coding very easy and faster.
- Integrated terminal. You can run your Python code directly from the IDE.

Which text editor should I use?

There are many options to choose from. I have given only three with some detail, but you can search for others and see which one fits your needs and likes better.

Other popular options are: IDLE, Jupyter Notebook, Atom, Spyder, Thonny, Vim, etc.

I can recommend you to try at least three options so you can see for yourself which code editor you feel more comfortable with.

Conclusion

In this article you learned how to work with text editors and IDEs. Writing your Python code in a text editor is a great practice for your future projects. Keep practicing!

Chapter 4: Data Types

Python can be used for a wide range of tasks involving different types of data. It can be used to make large and complicated computations using numbers, automating text operations, storing data in the form of lists, and many more.

There are a set of different types of data to work with each specific task. For numeric tasks you use numbers (integers, decimals), for text operations you use data in form of strings, and so on. In this article you will learn the basics of how to work with different kinds of data types in Python.

What you will learn

In this article you will learn the following topics:

- What are the different available Python built-in data types
- How to check the type of a variable or value using the `type()` function
- How data types are casted into other data types

Numeric Data Types

Numbers are everywhere, and Python is not the exception. There are different kinds of numbers you can work with. Let's enter the Python interpreter in a terminal to understand how each type of number work.

Integer

Integer numbers are those which don't have decimal part. They can be positive, negative and zero. Let's begin by assigning an integer number to a new variable called `x`.

```
>>> x = 1
>>> x
1
```

If you don't know what a variable is or how to work with them, check out a [previous tutorial](#). Now the variable `x` has the *value* of 1 assigned to it.

What you can do now is to use a Python function called `type()`. This function takes as argument the variable (or the value), and returns the *data type*. Write `type` followed by the name of the variable inside parentheses, and then hit Enter:

```
>>> type(x)
<class 'int'>
```

As you can see Python prints `<class 'int'>`, which means the variable is an **int** (short for integer). You will learn about functions in [another tutorial](#) of this course.

Float

You can repeat the same process with a real number. Let's try the number 3.14. If you assign this value to the variable called `y` and check for its type, this is what you obtain:

```
>>> y = 3.14
>>> type(y)
<class 'float'>
```

Python tells you that the data type for 3.14 is called **float**. This is a floating point number, which is used to represent real numbers.

Complex

Python also has implemented the complex data type to represent complex numbers, which have real and imaginary parts. To make a complex number, write it as a sum of the real part and the imaginary part. The imaginary part has to be accompanied by a *j* (the imaginary unit):

```
>>> z = 1 + 0.5j
>>> type(z)
<class 'complex'>
```

Keep in mind that the imaginary unit in Python is written with a *j* instead of an *i*.

Given a complex number, there are different methods you can use to extract information such as the real part, the imaginary part, the complex conjugate, etc. You will learn about [methods](#) in another tutorial.

Conversion from one type to another: casting

There are a set of functions that convert one type of variable (number) to another. They are `int()`, `float()` and `complex()`. Functions in Python have to be called using parentheses, and inside them you put the variable or value.

Integer to float


```
>>> x = 1
>>> float(x)
1.0
```

The result of calling the function `float()` on the variable `x` (which contains the integer value of 1) is 1.0. Python represents float numbers with a decimal point. So when the value doesn't have any decimal digits it adds a 0.

In some situations the decimal point is represented with a comma instead of a point. In Python you always have to use the decimal point to represent decimals, because the comma is reserved for other situations like using lists and sets.

Float to integer

```
>>> y = 3.14
>>> int(y)
3
```

In this case the result of the function `int()` applied to a float is just the integer part. Keep in mind that this is not the same as rounding the number. Take for example the float 9.99 and apply the function `int()`:

```
>>> int(9.99)
9
```

The result is still 9, and not 10 as it would have been if you rounded the number 9.99.

Integer to complex

```
>>> x = 1
>>> complex(x)
(1+0j)
```

When using the `complex()` function on an integer, the result is a complex number where the real part is the original value, and the imaginary part is 0. Note that Python explicitly prints 0j indicating that you are dealing with a complex number.

Float to complex


```
>>> y = 3.14
>>> complex(y)
(3.14+0j)
```

This is similar to the previous example, just in this case the real part is a floating point number instead of an integer.

Complex to integer

```
>>> z = 1 + 0.5j
>>> int(z)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to int
```

We just got a **TypeError**! Well, this makes sense because there is no way to convert directly from a complex number to an integer. You could however take the real part, or the imaginary part, or even compute the modulus, and then convert that result to an integer. Either way you need to specify what you want to do and tell it to Python.

Text Data Type

Python represents textual data using the *string* data type. You have already worked with strings in a previous tutorial. Let's check the type of a sample string:

```
>>> s = 'Hello World!'
>>> type(s)
<class 'str'>
```

The output tells us that the data type is called **str** (short for string). You can also convert strings to numbers. For example:

```
>>> s = '7'
>>> int(s)
7
>>> float(s)
7.0
```

```
>>> complex(s)
(7+0j)
```

Of course you can't convert a string that is not a number to a number type like an integer:

```
>>> s = 'Not a number'
>>> int(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Not a number'
```

Python throws an error (a `ValueError` in this case) claiming that the string is invalid to use with the `int()` function.

You can also convert a numeric value to a string by using the `str()` function:

```
>>> str(3)
'3'
>>> str(3.14)
'3.14'
>>> str(3+4j)
'(3+4j)'
```

Sequence Data Types

Another kind of data type that is very used in Python programs are *sequence* data types. This data types are used to hold sequence of values like numbers, strings and even combinations of both. You will learn about sequence data types with more detail in another tutorial.

List

The first sequence data type you will deal with is **list**. Let's make an example with a list of integer values. To define a list, enclose the elements inside square brackets `[]`, and separate each value with a comma `,`.

```
>>> my_list_numbers = [1, 2, 3]
>>> type(my_list_numbers)
<class 'list'>
```

In this example I have added a space between the comma and the next value. This is not necessary but it can improve readability. When calling the `type()` function on the list just defined, Python tells it is of a **list** data type.

Tuple

Another sequence data type is **tuple**. Tuples are similar to lists, with the difference that tuples are *immutable*, while lists are *mutable*. You will deal with these concepts in more detail in another tutorial. For now, let's define a tuple in a similar way as the list, but in this case we use parentheses `()` instead of square brackets:

```
>>> my_tuple_strings = ('John', 'Carol', 'Eva', 'Charles')
>>> type(my_tuple_strings)
<class 'tuple'>
```

Range

Another widely used sequence type is **range**. `range()` is a function that can take one or more integer parameters. The details of the range function are left for another tutorial. For now let's just define it with one integer parameter and check its type:

```
>>> my_range_var = range(6)
>>> type(my_range_var)
<class 'range'>
```

Python tells us that the data type is *range* as expected.

Mapping Data Type: dictionary

The **dictionary** data type is widely used in Python. For example, it is used in applications that work with APIs (Application Programming Interfaces), and it resembles the JSON format. An example of such an application is a [Weather App](#).

To create a dictionary, enclose the data inside curly brackets `{ }` (braces). Inside the curly brackets you have to set **key: value** pairs separated by commas. Use a colon `:` to separate the key and the value. For example, you can define the keys to be the name of a person and the country he lives in, while the values for each key can be anything which makes sense:

```
>>> my_dictionary = {"name": "Charles", "country": "Argentina"}
>>> type(my_dictionary)
<class 'dict'>
```

By calling **type** on the dictionary variable Python tells it is of class **dict** (short for dictionary).

The keys in a dictionary have to be unique. If you try to assign different values with the same key, Python will only keep the last one:

```
>>> my_dictionary = {"name": "Charles", "name": "Einstein"}
>>> my_dictionary
{'name': 'Einstein'}
```

Error but no error. As you may have noticed, Python didn't throw an error in the last example. When working with Python programs you will make a lot of errors (and that is completely fine). Sometimes Python will help you correct those errors by throwing error messages. But there are some special situations like in the last example, where Python doesn't have a problem with what you are doing. This is one of the hardest aspects of programming: how do you know you are making a mistake if nobody tells you?

You can, however, have repeated values for different keys:

```
>>> my_dictionary = {"name": "Charles", "teacher": "Charles"}
>>> my_dictionary
{'name': 'Charles', 'teacher': 'Charles'}
```

Set Data Types

Set

Sets are similar to lists. The items are stored inside curly brackets and separated with commas.

```
>>> my_set = {'John', 'Carol', 'Eva', 'Charles'}
>>> type(my_set)
<class 'set'>
```

The items in a set have to be unique as opposed to a list, where the items can be repeated. If you try to repeat the same item in a set, Python will not keep more than one of it:

```
>>> s = {'a', 'b', 'c', 'b', 'c', 'd'}
>>> s
{'c', 'b', 'd', 'a'}
```

The set data type is very useful when you need to retrieve the *distinct values* that are present, for example, in a list.

Frozen set

Frozen Sets are similar to sets, only in this case the items are unchangeable.

```
>>> my_frozen_set = frozenset({'John', 'Carol', 'Eva', 'Charles'})
>>> type(my_frozen_set)
<class 'frozenset'>
```

You will be dealing with modifying sets, lists and others in later tutorials.

Boolean Data Types

Sometimes you need to check whether an expression is valid or not. For example you would check an equality between two variables, like two names belonging to the same dictionary. If there are two values with the same name, you would return **True**. In case they are different, return **False**.

The data type that deals with this kind of values is referred to as **boolean** data type. Boolean values can have two possible values: **True** or **False**. Is important to remember the capital T for True, and the capital F for False.

```
>>> a = True
>>> type(a)
<class 'bool'>
>>> b = False
>>> type(b)
<class 'bool'>
```

You will learn how to properly work with boolean variables in another tutorial.

Binary Data Types

There is a set of data types that deal with **bytes**. Let's leave the details about this data types for another tutorial. Here are the three data types for reference.

Bytes

```
>>> my_bytes_var = b"DeusDev"
>>> type(my_bytes_var)
<class 'bytes'>
```

Bytearray

```
>>> my_byte_array = bytearray(123)
>>> type(my_byte_array)
<class 'bytearray'>
```

Memoryview

```
>>> my_memory_view = memoryview(bytes(123))
>>> type(my_memory_view)
<class 'memoryview'>
```

None Data Type

The last data type listed in this tutorial is the **NoneType** data type. The value of this data type is called **None** (capital N). This is used to assign a null value to a variable.

You may be wondering why would you even need this kind of data at all. You will see how this works and why is useful in later tutorials. For now let's see just one example:

```
>>> my_none_var = None
>>> type(my_none_var)
<class 'NoneType'>
```

Conclusion

In this article you have gone through the basics of several data types supported by Python. Some of them are used a lot and others not so much, but is good to have a basic understanding of each data type and what they mean before moving on.

The next step is to learn different functions and methods each data type has to offer. For example, you may need to round a float number to two decimals, capitalize a string, add values to a list, retrieve some information from a dictionary, etc.

Exercises

Here is a list of exercises intended for you to practice some of the topics covered in this article. You have learned about each Python data type using the terminal. I recommend you try this exercises in a new Python file and run the script with the terminal. If you don't know how to do this, refer to the article about how to use [text editors and IDEs](#).

1. Create a new variable called `my_int` and assign any integer value to it. You can try a positive integer, a negative integer and even the number 0. Check that the variable is actually an integer by printing its type.
2. Create a variable and assign a float number to it. Call the variable however you like. Keep in mind that variable names usually have meaningful names according to its value. Check its type.
3. Create a complex variable and check its type. Remember that complex numbers have real and imaginary parts, each of which can be either integer or float. Try creating a complex variable with no imaginary part, but the type being complex anyway.
4. Convert the float variable to an integer, and assign the new value to a new variable. Check the type of the new variable (it should be int).
5. Create a variable called `my_greeting` and assign a string like 'Hello World' to it. Print its contents and type. You can try enclosing your string with single quotes, double quotes, triple quotes. You can even try triple double quotes around it.
6. Create a list. Remember that lists are enclosed with square brackets, and each item is separated with a comma. Try using numbers, strings, and even combinations of both, meaning you can use integers, floats and strings in the same list. Check its type.
7. Create a dictionary. Let your imagination fly! I recommend you try different options for the keys and values (integers, strings, complex, even lists). The more you try, the more you will likely to get errors. Don't worry, you will learn by force! If you want a suggestion, try to make a dictionary such that it generates this error: *TypeError: unhashable type: 'list'*
8. Back to exercise #1, you created a variable called `my_int` and assigned a value to it. If you lost that variable for some reason, recreate it. Now create another variable called `my_boolean` and assign the following to it: `bool(my_int)`. Print the contents of `my_boolean`, and check its type. As you can see you can cast different values to boolean variables. You will see more on this in another tutorial.

Chapter 5: Working with strings

Python strings are everywhere. Most of your Python programs will work with strings in one way or the other. For example, you can use strings to work with large textual data, analyze it, retrieve specific information from text files, etc. You can also use strings to print part of the information that comes from a program, debugging and more.

In this article you are going to learn what are the most common use cases of Python strings, how to retrieve basic parts of a string and different ways in which you can show information using the `print()` function.

You can [click here](#) to get a Python Strings Cheat Sheet for free to help you practice with the exercises!

Python strings: the basics

In a previous article you learned the basic data types Python has to offer. One of them are strings, which is the topic of the present article.

What are Python strings?

Strings are sequences of **characters** which are enclosed in either single, double or triple quotes.

Open a new terminal and enter the Python interpreter (if you don't know how to do this, refer to the article on using the [terminal and Python interpreter](#)). Create a new variable called `my_string` and assign the string value "Hello World!":

```
>>> my_string = "Hello World!"
>>> my_string
'Hello World!'
>>> print(my_string)
Hello World!
```

If you write the name of the variable and hit Enter, the terminal will output the value assigned to the variable. Note that the terminal includes single quotes around *Hello World!*. But if you use the `print()` function to show the value instead, the terminal now shows the value *Hello World!* without quotes.

Python strings can even be empty. This can be useful in some cases as you will see in later chapters. To create an **empty string**, open and close quotes with nothing in between:


```
>>> empty_string = ""
>>> empty_string
''
>>> print(empty_string)

>>>
```

You can see that the same thing happened as with the previous example. If you define the empty string with double quotes, writing the name of the variable and hitting Enter will show the empty string with single quotes. If you use the `print()` function instead, the quotes are not shown, and in this case there is an empty space in the screen as the output.

What is the length of a string?

Every string has different properties. One of the most commonly used property is the number of characters, or the **length** of the string.

To obtain the length of a string, use the `len()` function. Let's continue with our example string "Hello World!". If you manually count the number of characters of this string (including the whitespace between both words) you will find there are 12 of them. Let's check this with the `len()` function:

```
>>> my_string = 'Hello World!'
>>> len(my_string)
12
```

There are other functions to work with strings. You will learn more about them in later chapters.

String index notation

You can retrieve specific characters of a string by using the **index notation**. To access a specific character from a string, write the name of the variable followed by square brackets. Inside the square brackets enter the **index** corresponding to the position of the character.

```
>>> my_string[index]
```

What is the index of a string?

The index is an **integer** value indicating the position of the character in the string. Index values start at 0 for the first position, then 1 for the second position, 2 for the third, and so on.

H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

Note that the index of the last element is not equal to the number of characters in the string, but one less. This is because the index starts with 0 instead of 1. In this example the number of characters (the length of the string) is 12, while the index of the last character is 11.

For example, if you want to access the first character of the string (in this case it is the letter H) use the index value of 0 inside the square brackets:

```
>>> my_string = 'Hello World!'
>>> my_string[0]
'H'
```

If you want to retrieve the fifth element, this corresponds to the index 4:

```
>>> my_string = 'Hello World!'
>>> my_string[4]
'o'
```

Index can also be negative. For the case of negative indexes you have to think in reverse: -1 is the last character in the string, -2 is the previous to last, etc.

H	e	l	l	o		W	o	r	l	d	!
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

In this case, the first character of the string can be retrieved using a negative index with a value equal to the length of the string, which is -12:

```
>>> my_string = 'Hello World!'
>>> my_string[-12]
```

```
'H'
```

What would happen if you try to access an element whose index goes out of range?

```
>>> my_string = 'Hello World!'
>>> my_string[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> my_string[-13]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

As expected, the Python interpreter throws an `IndexError` indicating that the index is *out of range*.

String slice notation

You can do more than select just one specific character from a string. The square bracket notation lets you retrieve a specified *range* of values from a string. For example, you may want to select the "World" part of the string (without the last exclamation character).

In order to do that you need to use two index values: the *starting* and *stopping* positions separated by a colon:

```
>>> my_string[start:stop]
```

This will return a string ranging from the `start` index all the way through `stop-1`. The second index `stop` is not included in the output. One way to remember this rule is to think of the number of characters returned as the difference between `stop` and `start`.

The rules are:

- First index `start` is included
- Second index `stop` is not included
- number of characters = `stop - start`

To retrieve the "World" part of your string, you need to go from index 6 through index 10 (included).



So the first index will be 6, and the second index will be 11 (one more).

```
>>> my_string = 'Hello World!'
>>> my_string[6:11]
'World'
```

The difference between 11 and 6 is 5, which is the number of characters in the string "World".

You can leave the *start* or *end* parameters empty. If you omit the *start* parameter, it will default to zero, while omitting the *end* parameter will default to the last character of the string.

```
>>> my_string[:5]
'Hello'
>>> my_string[6:]
'World!'
```

You can also retrieve a substring and skipping certain characters from it. For example, say you want the substring "World" but only the characters that have an even index. In this case they are 6, 8 and 10. The output should be "Wrd".



There is a third parameter in slice notation which is the **step**. This is also an integer parameter and indicates how many indices to avoid on each pass (the default value of the step is 1).

```
>>> my_string = 'Hello World!'
>>> my_string[6:11:2]
```

```
'Wrd'
```

The *step* parameter can also be negative. A negative step makes the output of the substring go backwards. For example, if you want the string "World" become "dlroW", use a step value of -1, a start of 10 and a stop of 5:

```
>>> my_string = 'Hello World!'
>>> my_string[10:5:-1]
'dlroW'
```

Python strings are immutable

You may be wondering if you can modify the elements of a string. For example in the string "Hello World!", the character "W" needs to change to lowercase: "w". You could try to assign the character "w" to the string at index position 6 (where the "W" character is located):

```
>>> my_string[6] = 'w'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Python throws an error saying that a *string object does not support item assignment*. So one possible solution is to simply create a new variable and make the desired string using the old variable. For example:

```
>>> my_new_string = my_string[:6] + 'w' + my_string[7:]
>>> my_new_string
'Hello world!'
```

Format strings

There are different ways in which a string can be constructed given some variables. For example, say you want to make a string with a welcoming message for a given user and inform their age.

```
>>> user = 'John'
>>> age = 23
```

To make an automatic welcoming text string given these variables, you can make use of *format strings*. The welcoming message can be something like "Welcome John ! You are 23 years old". Instead of directly writing the strings *John* and *23*, let's use the variables just defined. This can be useful in cases where the variable changes, and the welcoming message needs to change automatically without the need to write everything from scratch.

To make a format string, use curly braces `{}` where the variables should be located. At the end of the string use a dot followed by "format". The format method takes two parameters in this case: the `user` and the `age`. Each of these variables will be replaced where the curly braces are placed, in order.

```
>>> user = 'John'
>>> age = 23
>>> 'Welcome {} ! You are {} years old.'.format(user, age)
'Welcome John ! You are 23 years old.'
```

This is a very basic example of Python format strings. There is a lot more you can do with this, but that is something for another time. For more information you can read [this article](#).

Conclusion

In this article you have learned the basics of Python strings. Python strings are one of the most important data types, and many programs use them in one way or another. Here is a list of the topics covered on this article:

- What is a string
- How to compute the length of a string
- String index and slice notation
- Immutability of strings
- Format strings: the basics

Exercises

Once you read about the basics of Python strings, get ready to practice with this set of exercises. Don't give up!

1. In your own words, explain what is a Python string, and what is it used for. Think of different use cases where you may want or need to work with Python strings.
2. Create a new string and save it to a variable. The contents of the string can be anything you like, such as *'Hello! I am learning about Python strings.'*
3. What is the length of the string you created in exercise #2? Count the characters manually and check your answer by using the `len()` function.

4. Use the square bracket index notation to retrieve the first and last characters of the string. Use non-negative index values for this exercise.
5. Repeat exercise #4, but this time use only negative index values.
6. Use the slice notation to retrieve a substring. For example, if the string was *'Hello! I am learning about Python strings.'* you could retrieve the word *Python*. Use non-negative index values for this exercise.
7. Repeat exercise #6 using only negative index values.
8. Use the slice notation to return the complete string in reverse order.

Chapter 6: Working with lists

All programming languages have the feature of organizing data in efficient ways. In a previous article you learned quite a lot about the string data type. Strings can contain lots of information, but they are not usually chosen as a way of organizing tabular like data.

There is a popular data type to manage related data, which is the **list** data type. This is very useful if you need to store a list of names for a birthday reunion, or a list of products your company offers, a list of bar code numbers, etc.

In this article you will learn how to create and work with Python lists in different ways. You have learned about lists in a previous article, but here you will learn a lot more.

Python lists

To define a list in Python use square brackets: `[]`. Inside the square brackets, place the different values you want to store, separated with commas. For example, say you want to create a list with the numbers 1, 2, 3, 4 and 5:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> numbers
[1, 2, 3, 4, 5]
```

In the example I included a space after each comma. This is not strictly necessary, but it improves readability. Check the type of the variable `numbers`:

```
>>> type(numbers)
<class 'list'>
```

As you can see, the type of the variable `numbers` is `list`. In this example you used integer values to create a list. You can use another data type for the list elements, such as floats or strings:

```
>>> floats = [1.1, 3.14, 2.71, 5.0]
>>> floats
[1.1, 3.14, 2.71, 5.0]
>>> strings = ['hello', 'world', 'bye']
>>> strings
['hello', 'world', 'bye']
```


The cool thing about lists is that the elements don't need to be of the same type. You can mix different data types in the same list:

```
>>> mixed_list = [1, 'hello', 3.14, 2+3j]
>>> mixed_list
[1, 'hello', 3.14, (2+3j)]
```

Lists can be constructed using previously defined variables:

```
>>> my_name = 'John'
>>> names = ['Jane', my_name, 'Carol', 'Lucas']
>>> names
['Jane', 'John', 'Carol', 'Lucas']
```

You can also work with empty lists. For that, just write a set of empty square brackets:

```
>>> empty_list = []
>>> empty_list
[]
```

What is the length of a list?

The length of a list is the number of elements it contains. It can be obtained with the `len()` function, similar to how the length of a string is computed. For example, the length of the list `['John', 'Jane', 'Kevin']` should be 3, since it contains three elements (three strings in this case):

```
>>> names = ['John', 'Jane', 'Kevin']
>>> len(names)
3
```

In the particular case that the list has no elements, that is the list is empty, the length is 0:

```
>>> empty_list = []
>>> len(empty_list)
0
```

List index notation

You can access different elements of a Python list by using the **index notation**. In order to access a specific element in a list, write the name of the variable containing the list followed by square brackets `[]`, and inside the square brackets goes the index of the element you want to access. Remember that in Python index values start at 0, as you learned in the [strings chapter](#).

For example, the list `['John', 'Jane', 'Kevin']` has three elements. The first one is at index position 0, the second at index 1, and the third at index 2.



Let's access each element of the list of names using index notation:

```
>>> names = ['John', 'Jane', 'Kevin']
>>> names[0]
'John'
>>> names[1]
'Jane'
>>> names[2]
'Kevin'
```

List index can be negative. Negative indexes start at -1 for the last element in the list, -2 for the previous to last, and so on.



If you try to access an index that is not valid, Python will throw an error:

```
>>> names[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> names = ['John', 'Jane', 'Kevin']
>>> names[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> names[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Python list slice notation

Imagine you have a large list of items and you need to retrieve a specific subset of them. For example, each item may correspond to the mean price of BTC of each day on a given week. In the following example the list `prices` has seven elements, each one being a number corresponding to the price of each day of the week:

```
>>> prices = [16500, 16550, 16600, 16650, 16700, 16750, 16800]
```

The first item at index position 0 corresponds to the price on Sunday, second item at index position 1 is the price on Monday, and so on. Say you want to extract the prices ranging from Monday through Friday. Then you need to go from index 1 up to index 5, inclusive. For this you can use the square bracket index slice notation with two integer numbers separated by a colon:

```
>>> prices[start:stop]
```

The first integer number `start` is the starting index, which is *included*, and the second number `stop` is the stopping index which is *not included*. The result will be a new list ranging from `start` through `stop-1`. The number of items in the resulting list is the difference `stop-start`.

For the current example you have to use 1 and 6 for the starting and stopping positions, both separated by a colon:

```
>>> prices = [16500, 16550, 16600, 16650, 16700, 16750, 16800]
>>> prices[1:6]
[16550, 16600, 16650, 16700, 16750]
```

The start and stop indices can be left blank. The default value for the start index is 0, while the default for the stop is the length of the list minus one (which corresponds to the last element).

```
>>> prices = [16500, 16550, 16600, 16650, 16700, 16750, 16800]
>>> prices[:5]
[16500, 16550, 16600, 16650, 16700]
>>> prices[3:]
[16650, 16700, 16750, 16800]
```

A third parameter can be used in slice notation, which is the **step**. The step parameter is also an integer number, and indicates how many elements should the resulting list skip from the starting position. For example, you may want to start from the second element (index 1) all the way to the previous to last (index 5) and skipping one element on each step. This way the step parameter should be 2, while the start and stop parameters are 1 and 6 (remember the stop parameter is not included, that's why it should be 6, and not 5).

```
>>> prices = [16500, 16550, 16600, 16650, 16700, 16750, 16800]
>>> prices[1:6:2]
[16550, 16650, 16750]
```

Lists can contain other lists

Remember that the list data type can contain elements with mixed data types. The same list can contain integer numbers, floats, and even strings. As you may remember from a previous article of this course, there is a great number of data types Python has to offer. Lists can contain any of those data types as its elements.

One very useful technique that is often used among developers is to place *lists inside other lists*. This can be handy in many situations, for example if you need to store information in table format. Let's work on an example where only one of the elements of the list is another list.

```
>>> student = ['Jonh', 23, [73, 49, 94]]
>>> student
['Jonh', 23, [73, 49, 94]]
```

In this example you have a list containing a first element indicating the name of a student (a string), a second element being the age of the student (an integer) and a third element which can be a list of scores on a given course. Let's check the type of the **student** variable:

```
>>> type(student)
<class 'list'>
```

As expected, the variable `student` is of type `list`. Now let's check the length of this variable. What do you expect to be the length? Is it 3? Or should you also count the number of elements of the inner list? In the latter case the total length should be 5, since you have the name, the age, and the three notes.

```
>>> len(student)
3
```

As you can see, the length returns only 3. The inner list is counted as only one element for the outer list. You can check the length of the inner list by accessing it with index notation and calling `len()`:

```
>>> len(student[2])
3
```

This way you are only returning the length of the inner list. What happens if you call `len()` on the other elements?

```
>>> len(student[0])
4
>>> len(student[1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

The length of the first element is 4, since the string `'John'` has four characters. But the length of the second element throws an error, since the length of an integer is not defined.

Finally, let's see how you can access the elements of the inner list. The inner list is at index position 2, so you have to start writing the name of the variable `student` followed by the index 2 inside square brackets. Next, each element of the inner list also have indexes, in this case 0, 1 and 2. Add the corresponding index inside square brackets after the previous statement. You will end up with something like `student[inner_list_index][element_index]`:

```
>>> student = ['Jonh', 23, [73, 49, 94]]
>>> student[2][0]
73
>>> student[2][1]
49
>>> student[2][2]
94
```

In cases where more lists are inside other lists, you can use multiple square bracket notation to access inner elements like in this example.

List of lists. In this section you learned that lists can contain other lists. In the case that *all* the elements of a list are also other lists, it is said that you have a *list of lists*. For example, a list of lists can look like `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`. This is one of the ways to organize data that comes in table like format.

Python lists are mutable

Each of the elements of a list can be changed for another value. Let's go back to the `student` example and modify the name of the student.

The name of the student (string) is at index position 0. To change its value, write the name of the variable `student` followed by the index between square brackets. Let's assign the value `'Oliver'` to the first element of that list:

```
>>> student = ['Jonh', 23, [73, 49, 94]]
>>> student[0] = 'Oliver'
>>> student
['Oliver', 23, [73, 49, 94]]
```

As you can see, the list itself has changed (mutated) after the re-assignment. You can also try to edit the second score, which is the second item inside the inner list. That is, first index is 2 (the position of the inner list) and the second index is 1 (the second position inside the inner list):

```
>>> student[2][1] = 88
>>> student
['Oliver', 23, [73, 88, 94]]
```

You can also add and remove elements from a list. For this you have to make use of methods and functions. You will learn about those in later tutorials.

Python Lists vs Strings

Python lists are very similar to strings. They are both sequences of data. While a string can only contain a sequence of characters, lists can contain any type of data as its elements, even another list.

You can convert a string into a list by using the `list()` function:

```
>>> text = 'hello world'
>>> type(text)
<class 'str'>
>>> list_type = list(text)
>>> list_type
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> type(list_type)
<class 'list'>
```

You could try to convert a list into a string, but is not so easy. We will cover such techniques in later chapters of the course.

A big difference between lists and strings is that the elements of a list can be changed (the list is mutable), while a string is immutable.

Conclusion

In this article you learned some of the basics of a very important data type: lists. You learned how to create a list, which elements can be stored in a list, how to extract information from a list using index and slice notation and how to compute the length of a given list. You have also learned how to mutate a list, that is, how to change the values of certain elements by assigning new values using index notation.

There is a lot more you can do with Python lists, such as remove items, add new items, construct a list in a programmatic way using loops, sort its values, and more. You will learn these techniques and more in later tutorials.

Exercises

After you read the article on lists, you can practice what you have learned with the following exercises. Let's get to work!

1. In your own words, what is a list? What is the difference between a list and a string? What are their similarities?
2. Create a list of places you want to visit, for example countries or cities. Start with the ones you want to visit the most, so the list is ordered from best to worse in that sense. Save the

list in a variable. Check the type of your variable.

3. How many elements are on your list? Check your answer with the `len()` function, which computes the length of a list.
4. Retrieve the first element of the list, that is, the place you want to visit the most. Use the square bracket index notation to do this.
5. Repeat the exercise #4 for the last element of the list.
6. Make use of the square bracket slice index notation to retrieve the three most wanted places to visit.
7. Given the list `numbers = [1, 2, 3, [4, 5, 6, [7, 8, 9, [10]]], 11]`, how would you access each of the numbers it contains? For example, to obtain the number 1 you would do `numbers[0]`, for the number 2 `numbers[1]`, and so on.
8. Try to convert a list of characters to a string. For example, your list can look something like `['h', 'e', 'l', 'l', 'o']`. To convert the list to a string, use the `str()` function. What happened? Did it throw an error. If not, is the result what you expected?

Chapter 7: Working with dictionaries

The dictionary data type is very useful when dealing with data that comes in key-value pairs. Dictionaries are collections of data, in which the **keys** are unique identifiers related to each associated **value**. Dictionary keys are somewhat analogous to the index in a list, as you will see later in this tutorial'

Let's work with a simple example throughout this article, where a list of countries are associated with their respective capitals.

Country	Capital
India	New Delhi
Argentina	Buenos Aires
Spain	Madrid
France	Paris
Kenya	Nairobi

The countries are the **keys**, while their capitals are the **values**. Each value is associated with a *unique* key. While in this example both the countries and their capitals are unique, Python dictionaries can have repeated (non-unique) values, but the keys must be unique (you will see what happens if you try to insert a repeated key in a dictionary).

Python dictionaries: the basics

To define a Python dictionary use curly braces: `{}`. Inside the curly braces include key-value pairs separated by a colon `:`. For the example of this article, the key (country) and the value (capital) are both strings:

```
>>> countries = {'India': 'New Delhi'}
>>> type(countries)
<class 'dict'>
>>> countries
{'India': 'New Delhi'}
```

As you can see, checking the type of the newly created dictionary says that is of the **dict** type. Let's define the dictionary variable `countries` with all the information. To add new key-value pairs, separate them with a comma:

```
>>> countries = {'India': 'New Delhi', 'Argentina': 'Buenos Aires',  
'Spain': 'Madrid', 'France': 'Paris', 'Kenya': 'Nairobi'}  
>>> countries  
{'India': 'New Delhi', 'Argentina': 'Buenos Aires', 'Spain': 'Madrid',  
'France': 'Paris', 'Kenya': 'Nairobi'}
```

After printing the `countries` variable on the screen, you can see all the information from the example table, and each country is associated with its corresponding capital.

Another way to create a dictionary is to begin with an empty dictionary, and add new key-value pairs one by one. To create an empty dictionary use a set of empty curly braces, `{}`. To add new items to the dictionary, write the name of the variable where the empty dictionary is stored, followed by square brackets. Inside the square brackets write the `key`, in this case the country in string format. Finally assign the corresponding `value`, which is the capital.

```
>>> countries = {}  
>>> countries['India'] = 'New Delhi'  
>>> countries['Argentina'] = 'Buenos Aires'  
>>> countries['Spain'] = 'Madrid'  
>>> countries['France'] = 'Paris'  
>>> countries['Kenya'] = 'Nairobi'  
>>> countries  
{'India': 'New Delhi', 'Argentina': 'Buenos Aires', 'Spain': 'Madrid',  
'France': 'Paris', 'Kenya': 'Nairobi'}
```

By doing the assignment for each country and its capital this way, you will end up with the same dictionary you had before. This way of creating dictionaries can be helpful when dealing with cases where the dictionary must be created in a programatic way instead of manually.

What is the length of a dictionary?

The length of a dictionary is the number of the key-value pairs it contains.

For the example of this article, you should expect it to have a length of 5, which is the number of entries, or rows on the table. It is true that if you count both the countries (keys) and the capitals (values), you will end up with a total of 10, but the length of a dictionary is computed as the number of key-value *pairs*.

```
>>> countries = {'India': 'New Delhi', 'Argentina': 'Buenos Aires',  
'Spain': 'Madrid', 'France': 'Paris', 'Kenya': 'Nairobi'}
```

```
>>> len(countries)
5
```

In a particular case that the dictionary is empty, its length is 0:

```
>>> empty_dict = {}
>>> len(empty_dict)
0
```

Reading items from a dictionary

Now that you have learned how to define a dictionary with a bunch of key-value pairs, let's see how you can retrieve information from it. Imagine a scenario where you have a dictionary with all the countries and their capitals, and you want to check the capital of some specific country.

Having defined this simple example with 5 countries, let's check what is the capital of Spain. To do this, you need to find the *key* with the string value **Spain**, and obtain the associated *value*, which is the one after the colon. Write the name of the variable containing the dictionary, followed by square brackets. Inside the square brackets put the value of the key you are looking for:

```
>>> countries['Spain']
'Madrid'
```

Cool! Retrieving values from keys in a dictionary is analogous as getting a value from a list given its index. Let's see what would happen if you try to access an item with a non existent key:

```
>>> countries['China']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'China'
```

Python throws a **KeyError** claiming that the key China doesn't exist.

Editing key-value pairs

Dictionaries key-value pairs can be edited. Let's see how you can do this.

Adding new key-value pairs

You can add new key-value pairs to an already created dictionary. To do this, assign a new value (capital) to `countries[key]`, where the key is the new country. This is the same as adding new items to an empty dictionary.

```
>>> countries['Canada'] = 'Ottawa'
>>> countries
{'India': 'New Delhi', 'Argentina': 'Buenos Aires', 'Spain': 'Madrid',
 'France': 'Paris', 'Kenya': 'Nairobi', 'Canada': 'Ottawa'}
```

In this example you added the country Canada as a key, and assigned the value Ottawa, both strings. The updated `countries` variable is shown with the old values along with the new one at the end.

Updating key-value pairs

Remember that keys have to be unique. What would happen if you try to assign a new value to an already existing key? For example, let's assign the number 39 to Spain.

```
>>> countries
{'India': 'New Delhi', 'Argentina': 'Buenos Aires', 'Spain': 'Madrid',
 'France': 'Paris', 'Kenya': 'Nairobi', 'Canada': 'Ottawa'}
>>> countries['Spain'] = 39
>>> countries
{'India': 'New Delhi', 'Argentina': 'Buenos Aires', 'Spain': 39,
 'France': 'Paris', 'Kenya': 'Nairobi', 'Canada': 'Ottawa'}
```

What you are doing in this case is simply re-assigning a new value to an already existing key. This is analogous to updating a list item given its index.

Removing elements from a dictionary

You can also remove elements from your dictionary. For this you use the `del` keyword. For example, let's delete the Spain element:

```
>>> del countries['Spain']
>>> countries
{'India': 'New Delhi', 'Argentina': 'Buenos Aires', 'France': 'Paris',
 'Kenya': 'Nairobi', 'Canada': 'Ottawa'}
```

As you can see, the key Spain has been deleted, along with its corresponding value. When you delete elements from a dictionary by its key, what is deleted is the key-value pair. What happens if you try to delete an element that is not present in the dictionary? Let's try to delete the element with the Spain key again:

```
>>> del countries['Spain']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Spain'
```

As expected, a **KeyError** is raised, since there isn't an element with the key Spain anymore.

Dictionary key/value data types

In this article you have worked with a dictionary where the keys and values are all of the string data type. Python dictionaries can have other data types, such as integers, floats, lists, and even other dictionaries. Let's try some of those with a newly created dictionary.

```
>>> d = {}
>>> d[12] = 99.3
>>> d['pi'] = 3.14
>>> d['list'] = [1, 2, 3]
>>> d['dict'] = {'name': 'John', 'age': 19}
>>> d
{12: 99.3, 'pi': 3.14, 'list': [1, 2, 3], 'dict': {'name': 'John',
'age': 19}}
```

In this example, an empty dictionary **d** has been created. Next, a first item is added, with an integer key and a float as its value. Then a string as a key, and a float as the value. The third has a string as key and a list as its value. Finally, the last element has a string key and another dictionary as the value. By printing the dictionary **d** you can see everything works as expected.

So far so good. Let's try a list as a key:

```
>>> d[[4, 5, 6]] = 'key is list'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Ups! Not good. Python doesn't like lists as dictionary keys. What about another dictionary as a key?

```
>>> d[{1: 'one', 2: 'two'}] = 'key is dict'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

Oh man! What about a tuple?

```
>>> d[(1, 2, 3)] = 'key is tuple'
>>> d
{12: 99.3, 'pi': 3.14, 'list': [1, 2, 3], 'dict': {'name': 'John',
'age': 19}, (1, 2, 3): 'key is tuple'}
>>> d[(1, 2, 3)]
'key is tuple'
```

Tuple works as a key for a dictionary. But not lists nor dictionaries. The rule is that the *dictionary key must be of an immutable type*, such as integers, strings and even tuples, but not list nor dictionaries. Remember that lists and dictionaries are *mutable* data types, meaning their items can be changed.

Converting dictionaries to lists

Let's go back to the initial example, the one with the countries and their capitals. What would happen if you tried to convert that dictionary with key-value pairs (country-capital) to a list using the `list()` function? Let's see:

```
>>> countries = {'India': 'New Delhi', 'Argentina': 'Buenos Aires',
'Spain': 'Madrid', 'France': 'Paris', 'Kenya': 'Nairobi'}
>>> list(countries)
['India', 'Argentina', 'Spain', 'France', 'Kenya']
```

Interesting! The `list()` function applied to the dictionary returned a list with only the keys, sorted in the same order they were inserted in the first place. But the values seem to have been lost in the process.

You have seen the `list()` function here and in previous tutorials too. You may be wondering if there is a `dict()` function that converts lists to dictionaries. In fact there is. To use the `dict()`

function you need to define the key-value pairs as the elements of a list, where each one of them is a tuple containing one key-value pair separated by a comma:

```
>>> numbers_dict = dict([('one', 1), ('two', 2), ('three', 3)])
>>> numbers_dict
{'one': 1, 'two': 2, 'three': 3}
```

There are other ways to use the `dict()` function, but let's leave those for another time.

Conclusion

In this article you learned the basics of Python dictionaries. What they are, how to create a dictionary, how to edit its elements, delete, add, and more.

Things are getting more complicated as you advance through the course, so remember to practice what you have learned by doing the proposed exercises. Have fun!

Exercises

After you read the article on dictionaries, you should practice what you have learned with the following exercises. Good luck!

1. In your own words, what is a Python dictionary and how is it different from a Python list?
2. In this article I used a very simple example where the Countries are related to their Capitals. Another common example is to make a dictionary where personal information of a student is stored, such as its name, age, scores, etc. Think of an example where you could make use of a Python dictionary and construct it. Save it in a new variable.
3. Check the type of the newly created dictionary and verify that is of the class **dict**.
4. Use the square bracket notation to retrieve some of the values given the corresponding keys. Use this method to verify that every key-value pair is defined as is supposed to.
5. What is the length of your dictionary? Try to answer the question before using the built-in Python function that gives you the answer automatically.
6. Going back to exercise #2, you may or may not have defined one of the values as a list or a dictionary. If you haven't, add a new item to your dictionary where the value is either a list or another dictionary.
7. Use the square bracket notation to access the list or dictionary you have stored as one of the values. You should have done this either in exercise #2 or #6.
8. Use the square bracket notation to access one of the list or dictionary values you have defined inside the original dictionary. For this you will have to make use of two sets of square brackets, similar to how you access list elements inside another list.
9. Convert your dictionary to a list, and save the contents to a new variable. What is the length of this list? Where are the elements of this list coming from?

Chapter 8: Conditional statements: if, elif, else

Conditional statements are almost everywhere in computer programming. This part of the course will teach you about simple decisions you can work with on your Python programs.

You make simple decisions in your everyday life, based on a number of variables. For example, if you know that it is raining or is going to rain today, you may decide to take an umbrella when going out. Or if you get a call on your phone you will decide whether you attend it or not based on who it is.

Python programs can deal with simple decisions like those by combining *boolean expressions* and the `if` statement, which you will learn about shortly.

Revisiting the boolean data type

Think about the raining example of the introduction. How do you decide whether to take an umbrella or not? You have to check if it is raining. If you ask yourself, is it raining? What are the possible answers? **Yes** or **No**. Python has a special data type to deal with such binary values: the *boolean data type*.

Remember that boolean variables can take two values: `True` or `False`. For our example, `True` will be the **Yes**, while `False` is the **No**. You have learned the very basics of the boolean data type in a previous tutorial. Let's go further now that you need this data type to make simple decisions.

Boolean values and variables

Boolean values can be directly defined with the keywords `True` or `False`:

```
>>> raining = True
>>> raining
True
>>> raining = False
>>> raining
False
```

In this case the variable `raining` will contain `True` or `False`, depending on whether it is raining or not outside. Once you define your variable, you will make use of it for the logic of simple decisions.

Boolean operators: and, or, not

Boolean logic can be much more complex than just defining a single value as **True** or **False**. Imagine you want to check if two different variables are **True** at the same time. For this you can use the **and** boolean operator:

```
>>> raining = True
>>> cold = False
>>> raining and cold
False
```

The different combinations for the **and** boolean operator are on the table below:

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

Another boolean operator is the **or**, which is used to check if at least one of the variables is set to **True**.

```
>>> raining = True
>>> cold = False
>>> raining or cold
True
```

The combinations for the **or** operator are:

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

The **not** operator negates the value:

```
>>> not raining
False
```

```
>>> raining = True
>>> not raining
False
>>> raining = False
>>> not raining
True
```

Here is the table for the **not** operator:

a	not a
True	False
False	True

Simple decisions: the if statement

Let's see how a simple decision can be automated with a Python script. First of all, define a variable called **raining**, and assign it the value **True**, meaning it is actually raining today. Then comes the **if** statement, followed by a boolean expression. In this case the boolean expression is simply the variable **raining** containing the boolean value **True**. In the same line at the end, finish with a colon **:**.

Below the **if** statement comes the **body**. This is the part of the script you want to be executed only in case the boolean expression has a value of **True**.

```
raining = True

if raining:
    print("It's raining. Let me get my umbrella!")

print("Have a good day!")

# It's raining. Let me get my umbrella!
# Have a good day!
```

As you can see, the body of the **if** statement, that is the first **print**, executes because the boolean expression (the variable **raining**) has a value of **True**. What happens if the **raining** variable is set to **False**?

```
raining = False

if raining:
    print("It's raining. Let me get my umbrella!")

print("Have a good day!")

# Have a good day!
```

The script does not print the first message because the `if` statement encounters an expression that has a value of `False`. Then, the body of the `if` statement is skipped (not executed). Only the last message is printed to the screen, because that one is not part of the body of the `if` statement.

The body of an `if` statement can have more than one line. For example:

```
raining = True

if raining:
    print("It's raining. Let me get my umbrella!")
    day = 'Sunday'
    print('Today is ' + day + '. Have fun!')

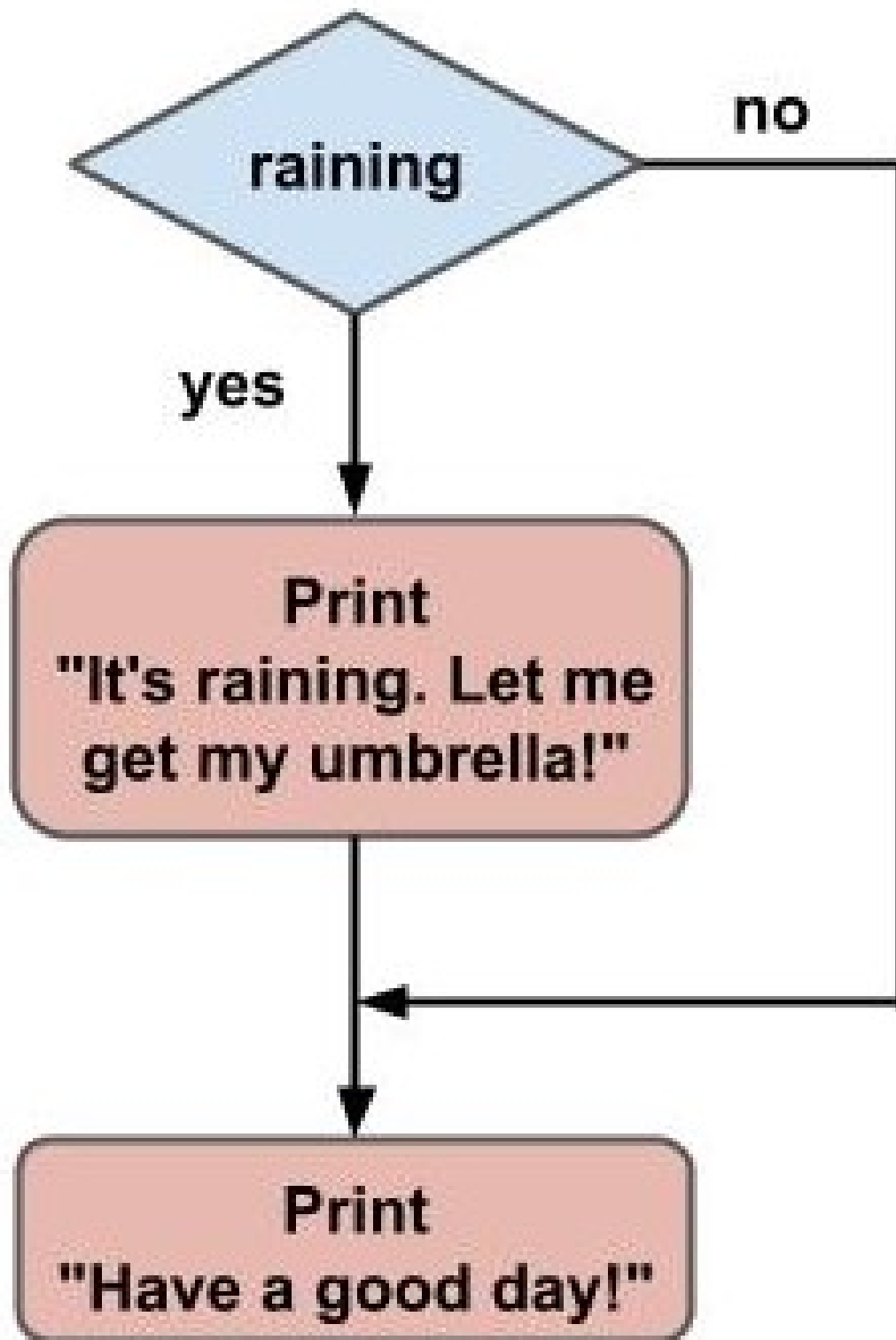
print("Have a good day!")

# It's raining. Let me get my umbrella!
# Today is Sunday. Have fun!
# Have a good day!
```

In this example the body of the `if` statement has three lines. When the boolean expression of the `if` statement has a value of `True`, the whole contents of the body will execute before moving on. In case it is `False`, the complete body will be skipped.

Flowchart diagrams

When dealing with simple decisions like the ones you are learning in this article, it is a good idea to organize your program and think about its logic before moving on to writing any code. This is a great idea specially when a program is getting too complex and the code is harder to read. A *flowchart* diagram for the last example would look like this:



The **diamond shaped box** contains the *conditional decision*, which will be part of the `if` statement in your program. In this case that is the `raining` boolean variable.

The **rectangular boxes** shows what the code will do, in this case those are simple `print` statements, but it can be much more than that, such as assigning values to variables, or more `if` statements.

The **arrow lines** indicate the *flow* of the program. Each diamond shaped box in this case can follow two possible paths: one for the case that the boolean expression has a value of **True**, and another when it is **False**. You can use **True/False** instead of **yes/no** in the diagram. For this example the **True** case makes it go to the first print statement, while the **False** case avoids the first print and jumps directly to the second one.

Python indentation

In all the examples where there is an **if** statement you may have noticed that the body is not in the same level as the rest of the lines of the script. If you are using a text editor like Sublime or VS Code, hitting Enter after the line containing the **if** statement (right after the colon) will make an automatic *indentation* for you.

In case you are not using a text editor that makes the indentation automatically, you will have to make it yourself. The indentation can be made using a tab, or using 2 or 4 spaces. What happens if you don't indent your code? Try the following, without indentation:

```
raining = True

if raining:
print("It's raining. Let me get my umbrella!")

'''
File "pythoncourse.py", line 4
    print("It's raining. Let me get my umbrella!")
    ^
IndentationError: expected an indented block
'''
```

Python will throw an **IndentationError**, which is very helpful for debugging purposes. It actually points where the indentation was expected in your code.

In other programming languages this is a good practice that improves readability of your code. But in Python the indentation is something that is used specifically to group blocks of code belonging to different parts of the script, for example the block code of an **if** statement. Indentation in Python is not optional, is mandatory.

The else statement

What if you want to print a different message if the condition is not met? In the previous example the message "Have a good day!" was shown whether it is raining or not. Let's leave that message as it is, but let's see how you can include an extra message for the case where it is not raining outside.

The `else` statement comes into play for this example. After the `if` statement and its corresponding code block, include the keyword `else` at the **same level of indentation** as the `if` statement, finishing with a colon `:`. Then comes the code block belonging to the `else` statement with the new message.

```
raining = True

if raining:
    print("It's raining. Let me get my umbrella!")
else:
    print("It is not raining. Leave the umbrella alone!")

print("Have a good day!")

# It's raining. Let me get my umbrella!
# Have a good day!
```

After running the example where `raining` is set to `True`, only the first message is printed to the screen, just as the previous example. What happens when you set the variable `raining` to `False`?

```
raining = False

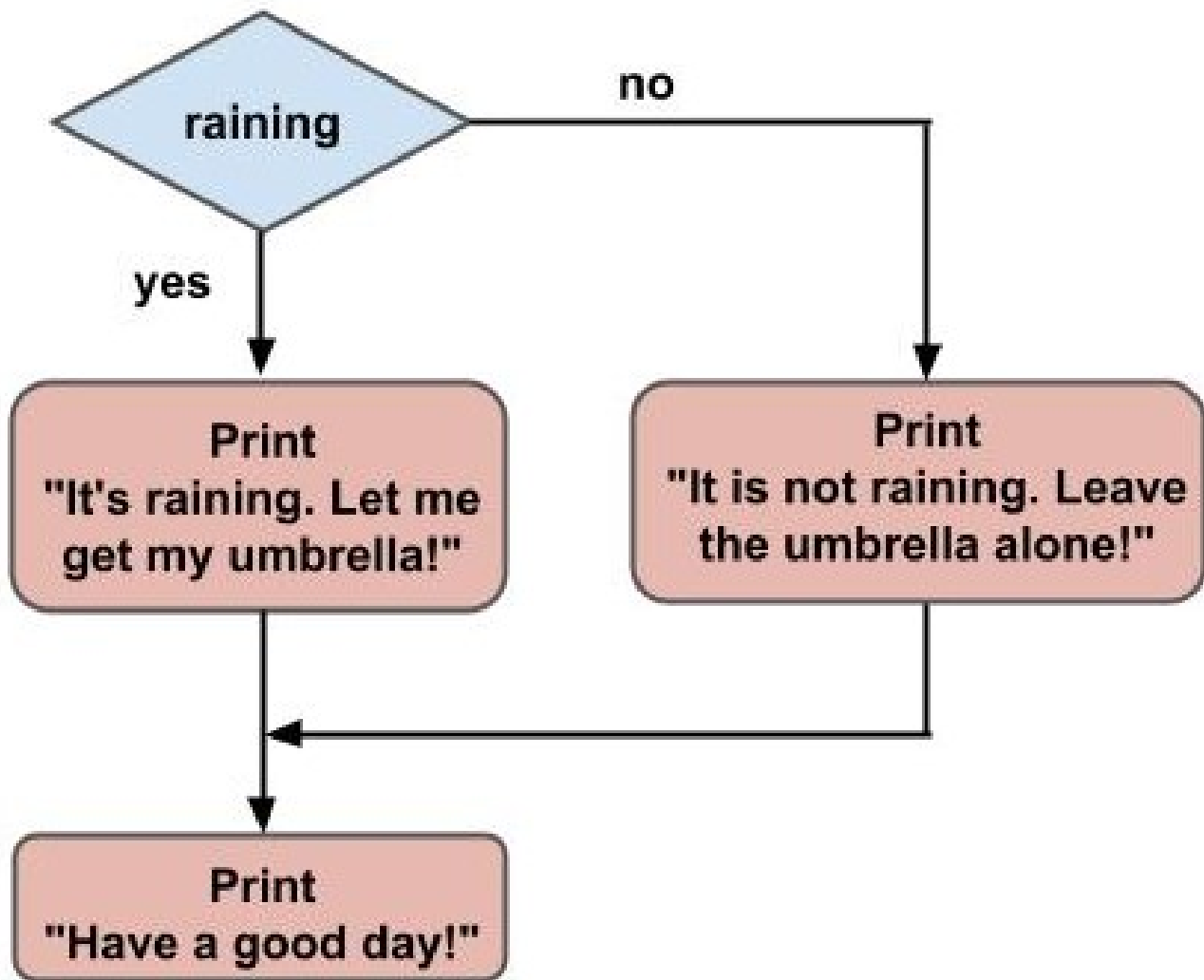
if raining:
    print("It's raining. Let me get my umbrella!")
else:
    print("It is not raining. Leave the umbrella alone!")

print("Have a good day!")

# It is not raining. Leave the umbrella alone!
# Have a good day!
```

In this case, only the second message is printed to the screen. The code block belonging to the `else` statement is executed only when the boolean value resulting from the `if` statement evaluates to `False`.

A flowchart diagram for this example would look like this:

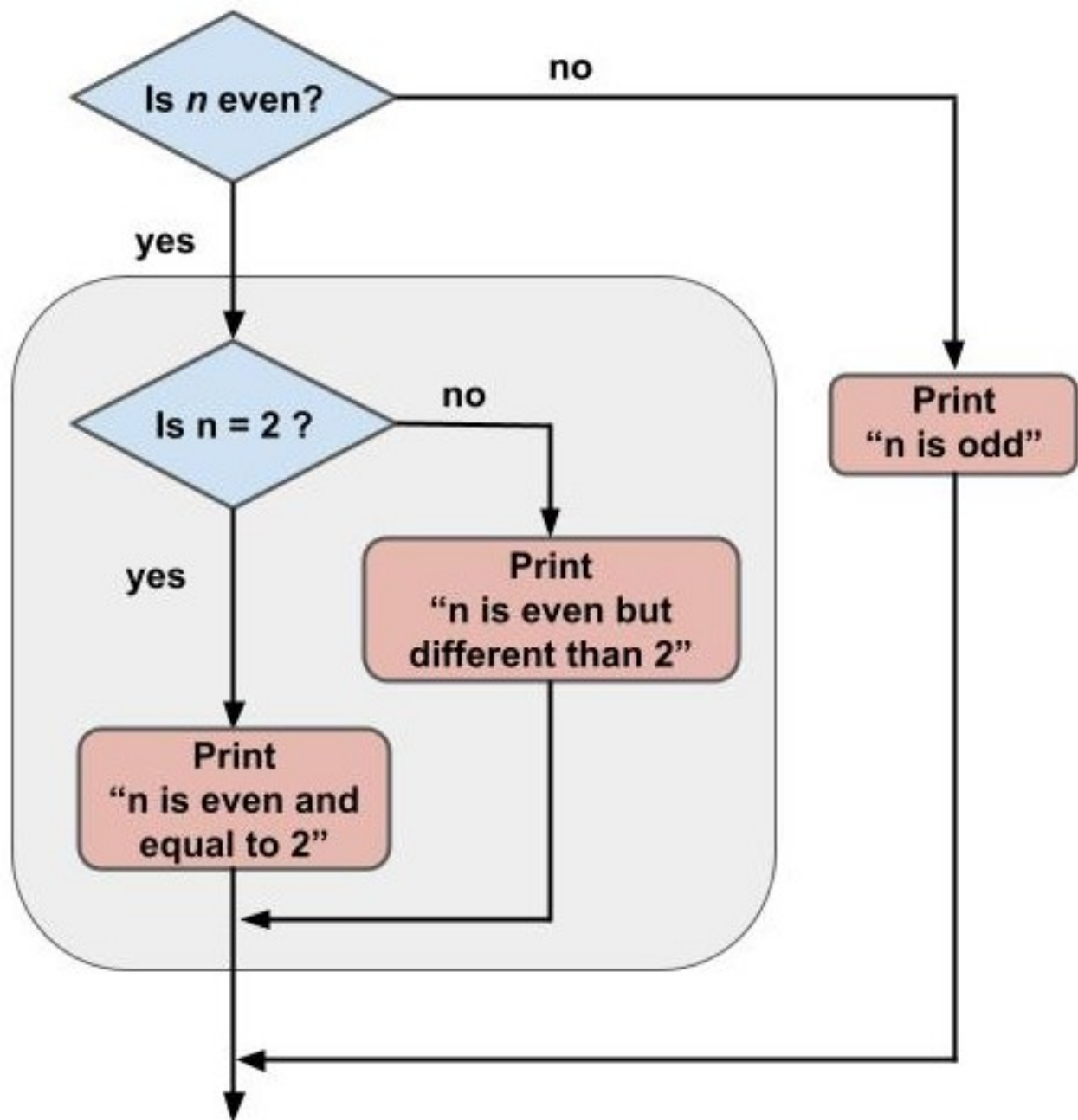


The **False** case now has a path with a **print** statement in between, making it show a custom message in between instead of jumping to the end of the script.

Nested if-else statements

So far you have seen simple **if-else** statements, where a block of code is executed only if a condition is met, and another if the condition is not met. But **if-else** statements can be made more complex by chaining more than one of them together.

For example, say you are working on a math problem where you have an integer number, and you want to check if that number is even or odd. Only in case the number is even (a multiple of 2), check if the number is equal to 2. Let's make a diagram to organize the logic before writing any code.



The first thing the program will do is to check whether the number `n` is even or odd. The diamond shaped box indicates that this is part of a main `if` statement. If the number is not even, it means that it is odd, and the program should terminate with a message saying that the number is odd.

If the number is even, you have to make another decision to check if the number is equal to 2 or not. Make another `if` statement to check for the equality, and if the number is equal to 2, show a message indicating that information, and leave the program. If the number is different than 2, print a different message and leave the program. For this later case you should make use of an `else` statement.

To check if a number `n` is even or odd you can make use of the *modulo (or modulus, or remainder) operator*, symbolized by `%`. This operator returns the remainder of the division

between two numbers. When you divide a number by 2, you can get two possible remainders: 0 or 1. If you get a remainder of 0, it means the number is a multiple of 2 (**even**), while if you get a remainder of 1, the number is not a multiple of 2 (**odd**). Open a Python interpreter and try out the modulo operator with different integer numbers:

```
>>> 1%2
1
>>> 2%2
0
>>> 3%2
1
>>> 4%2
0
>>> 132%2
0
>>> 767%2
1
>>>
```

To check the equality between the remainder of a number with 2 and 0, you can use the `==` operator, which returns True if both sides have the same value, otherwise it returns False. The condition `n%2 == 0` returns **True** for even numbers and **False** for odd numbers.

```
n = 5

if n%2 == 0:
    # Here comes the main if block
```

Inside the main **if** block comes another **if** statement to check if the number is equal to 2. If it is equal, print a message indicating this fact, and if it is different print another message (using an **else** statement).

Below the last print make another **else** statement, this time at the *same level of indentation as the first if statement*, since this last **else** belongs to the first **if**. This corresponds to the case where the number is odd.

```
n = 5

if n%2 == 0:
```

```
if n == 2:
    print("n is even and equal to 2")
else:
    print("n is even but different than 2")
else:
    print("n is odd")
```

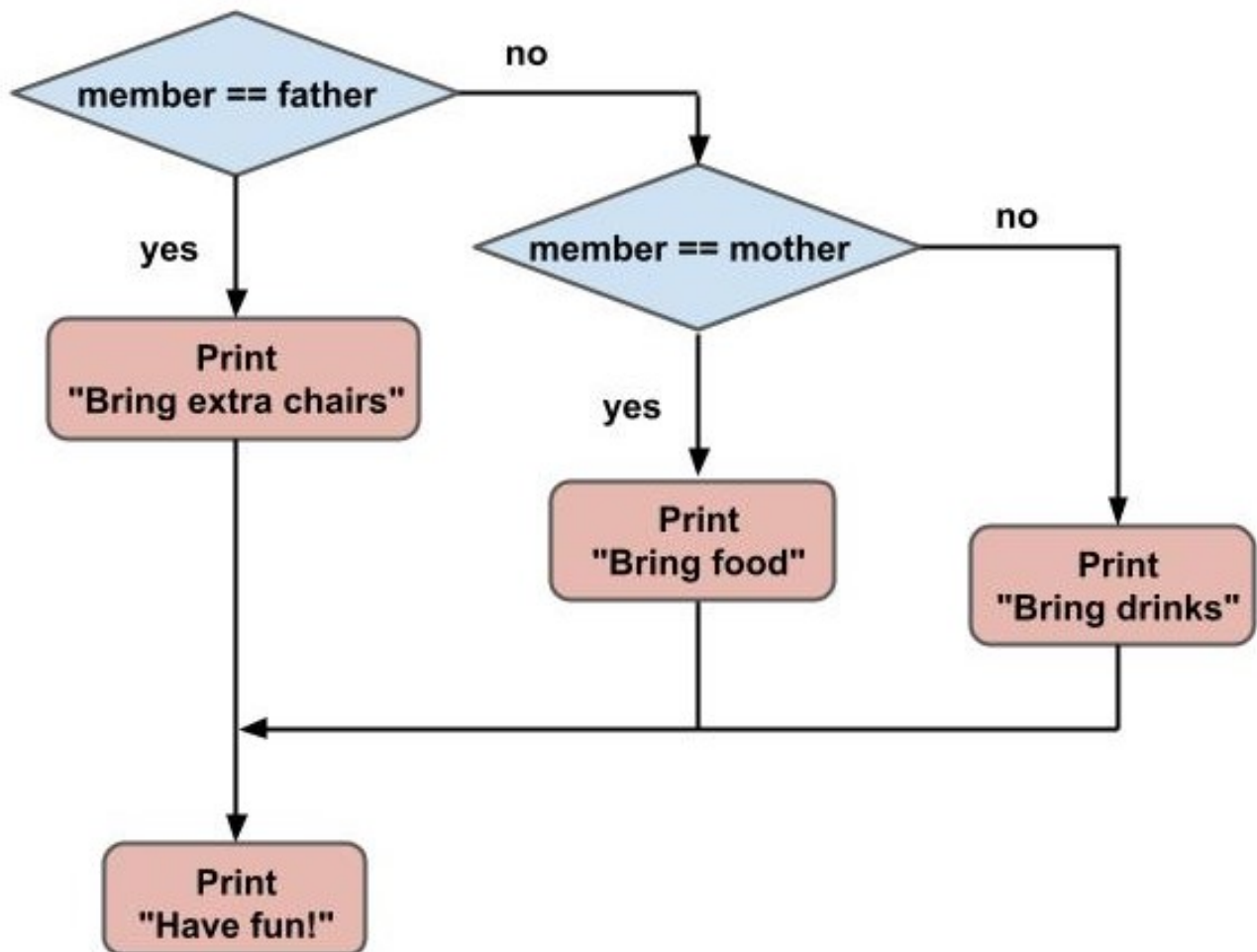
Things are getting complicated, right? This is just the beginning. Once you get comfortable understanding and coding such logics, you can move on to more advanced topics. At the end of this article you can practice more on this with the exercises.

The elif statement

Let's work on another example to see how you can make a still more complex logic. Imagine you are organizing a family dinner, and each of the family members have to bring something different to the meeting. The members can be your father, your mother and others.

Your father has to bring extra chairs (because you only have two chairs at your house), your mother brings the food, and everyone else the drinks. All of you still needs to decide on who is gonna cook, and who is gonna do the dishes. Let's leave this for a future tutorial where you can make use of random techniques with Python.

Let's make a simple Python program that takes a string variable `member` containing the name of the family member and prints a message indicating what he/she needs to bring to the reunion. Before moving on to the code, let's do a flowchart.



Start by checking if the member is the father. In case it is, print a message indicating that he needs to bring extra chairs. In case the member is not the father, you still need to check if it is the mother or not. If it is the mother, print a message saying that she needs to bring food, and exit the program. If the member is not the mother, there is only one last case in which the member (whoever it is) has to bring drinks. After that, exit the program and print a "Have fun!" message.

As you can see this example is similar to the last one, where you learned how to make nested if statements. But this is different because the nesting is on the negative (False) part of the decision. You could make this work by combining an `else` followed by an `if` statement, but Python has a built-in keyword to make this easier which is the `elif` statement.

Start with the `if` statement, checking if the member is the father. In case it is, print the corresponding message (after that, the program will automatically exit the `if` statement). Then instead of an `else` statement, place the `elif` statement at the same level of indentation as the `if`, and write the second condition, where the member is equal to the mother. If this is the case, print a message to bring the food. Finally, finish with the `else` statement in case the member is other than the father or the mother, in which case the message is to bring drinks.

```
member = 'father'

if member == 'father':
    print('Bring extra chairs')
elif member == 'mother':
    print('Bring food')
else:
    print('Bring drinks')

print('Have fun!')
```

You can make use of more `elif` statements in case you want to check for different members of the family.

Conclusion

In this article you learned about conditional statements and simple decision making for Python programs. Conditional statements are made with the keywords `if`, `elif` and `else`.

Flowchart diagrams are a great way to organize the logic of your programs, specially when they get long and complicated. You have learned how to read and construct simple flowchart diagrams for simple decision making programs.

After you read the contents of the article, keep practicing to fully learn the conditional statements topics.

Exercises

1. To recap the basics of boolean logic, what are the possible values a boolean variable can take?
2. Write a Python program to check if a given integer number (different than zero) is either positive or negative. Draw a flowchart diagram before writing any code.
3. Write a Python program similar to the one of exercise #2, and if the given number is positive, check if that number is equal to 10.
4. Write a program similar to that of exercise #2, but letting the number be zero also. Use the `elif` statement somehow.
5. How can you check if a given positive integer number is a multiple of 5? Write the boolean condition and test it against different numbers.
6. Write a program that given a string checks if its length is less than 10. In case it is less than 10, print a message with that information. If the length is not less than 10 (meaning the length is greater than or equal to 10), print a different message. Finally, the remaining case would be that the length is exactly equal to 10, in which case print a message with this information.

7. Write a program that, given a variable of any type (string, integer, float, dictionary, etc), prints a message saying the type of the variable. For example, if the variable is `my_variable = [1, 2, 3]` the program should print something like 'Your variable is of the list type'. You should make use of multiple `elif` statements for this exercise. The boolean expression to check if a variable is, for example, an integer, is `type(my_variable) == int`.

Chapter 9: Loops, for and while, break and continue

One of the most interesting and useful techniques in every programming language is the ability to automatically repeat a task a given number of times. Computers are perfect for such repetitive tasks, because they do it fast and they don't get tired of it as humans do.

The examples you are going to work with in this article are so simplistic that when you run them it will seem like the computations are practically instantaneous. This is not at all true! Computers takes some determined time to run a program, and this depends on how fast your computer itself is and the complexity of your program.

In this article you will learn the basics of loops, how they work, when to use them and its different use cases.

For loops

The first type of loop you will learn is the **for** loop. Let's take an example where you have a list of names, each of which is represented as a Python string. You can print each individual name separately by using a **for** loop.

A **for** loop is constructed beginning with the **for** keyword, followed by a *temporal variable* name (let's call it **name** for this example), then the keyword **in**, and finally a *generator* (the list of names in this case), and ending with a colon **:**.

Below the line containing the **for** statement comes the body, which is an indented block code, similar to the **if** statements. For this example the body of the **for** loop will be just a **print** with the **name** variable.

Iterating over a tuple

```
for name in ('John', 'Jane', 'Karen'):
    print(name)

# John
# Jane
# Karen
```

As you can see the program prints each of the names in a new line. The list of names was given as a tuple, which plays the role of the *generator*. The program you just run is equivalent to the following:

```
print('Jonh')
print('Jane')
print('Karen')
```

```
# John
# Jane
# Karen
```

I hope you already feel the power of **for** loops. Just imagine having a list with thousands of names. The second example where you have to make a **print** for each name written manually is not good at all. With the **for** loop you would do it much more easily. The only problem is that you need to have the list of names ready to give them to the loop. That list can also be read from a file or a database.

Iterating a list

Let's see another example with numbers this time. Given a list of integer numbers, how would you go about printing the square of each of them? The variable can be called **number**, and the generator is just a list of numbers. Let's work with the numbers from 1 to 6. To print the square of each number, make a **print** with **number**2**.

```
for number in [1, 2, 3, 4, 5, 6]:
    print(number**2)
```

```
# 1
# 4
# 9
# 16
# 25
# 36
```

Iterating a string

for loops can also be used to iterate over a string. Let's see what happens when you try to make a **for** loop with a string as the generator:

```
for c in 'hello world!':
    print(c)
```

```
# h
```

```
# e
# l
# l
# o
#
# w
# o
# r
# l
# d
# !
```

Each of the characters belonging to the string `'hello world!'` is printed to the screen when looping over it. This can be useful to find specific characters or substrings on a large text or document.

Iterating a dictionary

Iterating over a dictionary is also possible as you can see in this example:

```
d = {'name': 'John', 'age': 23, 'score': 74.5}

for item in d:
    print(item)

# name
# age
# score
```

But the only thing that prints when iterating over the dictionary are the keys, but not the values. How can you print both the keys and the values of the dictionary? One way is to use the square bracket notation using the variable name `d` and the `item` temporal variable:

```
d = {'name': 'John', 'age': 23, 'score': 74.5}

for item in d:
    print(item, d[item])

# name John
```



```
# age 23
# score 74.5
```

Combining for and if statements

You can combine **if** statements with your **for** loops. Let's make an example with a list of names, and print only those that start with 'J'.

```
for name in ('John', 'Jane', 'Karen', 'Jack', 'Ruth'):
    if name[0] == 'J':
        print(name)

# John
# Jane
# Jack
```

Pay attention to the indentation levels. In this example you see two different indentation levels: one for the **for** loop, and a second one for the **if** statement.

The range function

A highly common way of working with **for** loops is to use the **range** function. You will learn about Python functions later in this course, but the **range** function will be introduced now.

The **range** function is used to make a generator commonly used with **for** loops, instead of making a list or a tuple. For example, say you want to print the numbers from 1 to 100. Making a list of those numbers with what you know so far can be laborious, but with the **range** function is as easy as writing **range(1, 101)**. The first parameter is included, while the second is not included. This is similar to the index slice notation you learned in a previous tutorial.

```
for i in range(1, 101):
    print(i)

# 1
# 2
# 3
# 4
...
# 99
# 100
```

If you don't include the first parameter, the default will be 0:

```
for i in range(5):  
    print(i)  
  
# 0  
# 1  
# 2  
# 3  
# 4
```

You can also give a step parameter, which comes in the third place after the first two:

```
for i in range(3, 15, 2):  
    print(i)  
  
# 3  
# 5  
# 7  
# 9  
# 11  
# 13
```

Whenever you have to make use of a series of integer numbers in your **for** loops, always try to make use of the **range** function.

Nested for loops

for loops can be combined with other **for** loops. This means that you will have a main **for** loop iterating over certain items, and within it another **for** loop, iterating over (possibly) the values of the first loop.

Let's test the idea with an example. Make a first **for** loop with integer numbers from 1 to 3 using the **range()** function and a temporal variable called **i**. In the body of this **for** loop make a second **for** loop and use the **range()** function with the variable **i** of the first for loop, and use a second temporal variable called **j**.

This way you can print the contents of both **i** and **j** variables to see the effect of chaining two **for** loops together. Note the indentation levels. I have included an extra empty **print()** to separate the outputs of both loops.

```
for i in range(1, 4):
    for j in range(i):
        print(i, j)
    print()
```

```
# 1 0
#
# 2 0
# 2 1
#
# 3 0
# 3 1
# 3 2
```

The first loop will make the variable `i` take values from 1 to 3. For each of those values, the second loop will loop through integer values from 0 to `i` (not inclusive).

This is the most basic example, but the possibilities are endless. You can even combine `for` and `while` loops together, use `if-else` conditionals, and more.

While loops

The second type of loop is the `while` loop. This type of loop works with a boolean condition, similar to the `if` statement. The difference with the `if` statement is that the body of the `while` loop will repeatedly execute as long as the condition evaluates to `True`.

Let's work with an example to understand the dynamics of the `while` loop. Start by defining an integer variable `i` with a value of 0. This variable will be incremented in each iteration of the loop. The `while` statement is made with the keyword `while` followed by the boolean condition, and ending with a colon `:`.

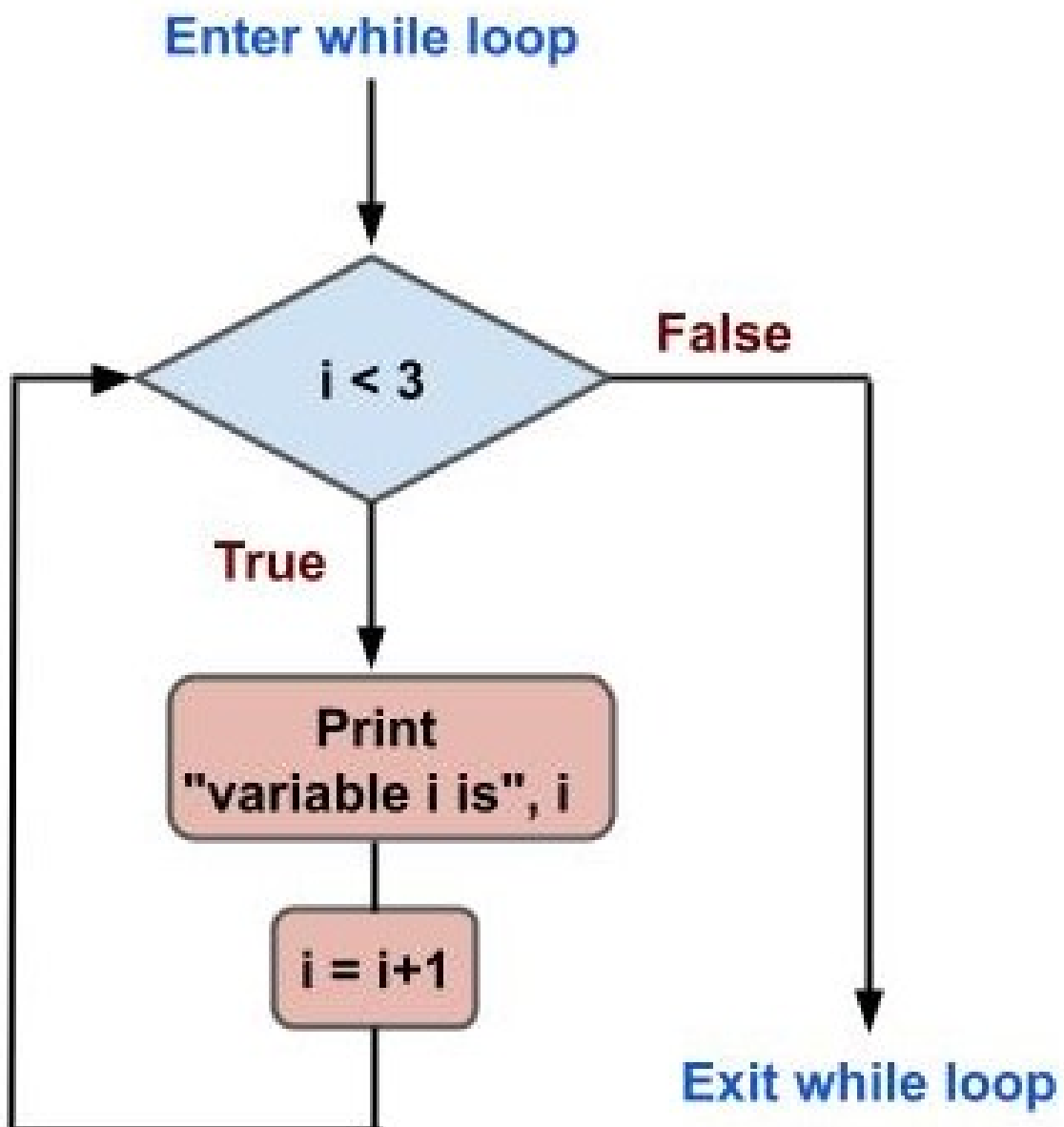
In the body of the `while` loop make a print statement to show the current value of the variable `i`. Also in the same body (same level of indentation) increase the value of the variable `i`. For this you can assign `i+1` to the same variable `i`, that is `i=i+1` (you can also increment a variable with `i+=1`).

```
i = 0
while i < 3:
    print('Variable i is ', i)
    i = i+1
```

```
# Variable i is 0
# Variable i is 1
# Variable i is 2
```

The program starts with `i=0`. When entering the `while` loop, it checks if the variable `i` is less than 3. Since 0 is less than 3, the condition evaluates to `True`, and the body of the `while` loop executes. The value of the variable `i` is printed and then the variable `i` is incremented by one. This process is repeated until the condition fails, that is when the value of `i` is equal to 3.

You can draw a flowchart diagram similar to those you learned in the conditional statements chapter.



Infinite loops

There is a very common "mistake" programmers do when working with loops, specially **while** loops. Let's go back to the previous example, but this time without the **i=i+1** line. Let's see what happens if you execute the program as it is:

```
i = 0
while i < 3:
    print('Variable i is ', i)

# Variable i is 0
# Variable i is 0
# Variable i is 0
# Variable i is 0
# Variable i is 0
...
```

Well, the script seems to be stuck and prints always the same message: **Variable i is 0**. This makes sense, because you defined the variable **i=0** at the beginning and the condition **i<3** is always True, because the variable **i** is never updated.

The loop will never terminate because of this, so what to do? If you are executing the program with a terminal like the Windows Powershell, you can do **Ctrl+C** to force the program to stop. You will get this:

```
...
Variable i is 0
Variable i is 0
Traceback (most recent call last):
  File ".\pythoncourse.py", line 3, in <module>
    print('Variable i is ', i)
KeyboardInterrupt
```

Python throws a **KeyboardInterrupt** error indicating that you interrupted your program with Ctrl+C in this case.

Always pay attention to the conditions and variable updates specially in **while** loops to avoid such headaches.

Control statements

Python loops can be further controlled for specific cases. So far you have seen that the loops iterate over a whole list of items before finishing using a **for** loop, or they iterate until a boolean

condition is met, evaluating to `False` with `while` loops.

There are situations where you need to prematurely stop your loop, or skip specific items. The `break` and `continue` statements are designed for such purposes.

The break statement

Loops can be terminated prematurely when a special condition is met. To do this there is a special keyword called `break`. When the body of a loop finds a `break` statement, the program exits the loop and continues with the rest of the program.

```
for name in ('Jonh', 'Jane', 'Karen', 'Susan'):
    print(name)
    if name == 'Karen':
        break

# John
# Jane
# Karen
```

In this example you have a `for` loop which iterates over a tuple containing four strings representing names. Each name is printed to the screen, and then there is an `if` statement which checks if the name is equal to 'Karen'. If the condition is met, a `break` statement is executed and the `for` loop is terminated. That's why the name 'Susan' is not printed.

The continue statement

There are situations where certain iterations needs to be skipped, but without the whole loop being terminated as is the case with the `break` statement. For example, say you have a list of names and you want to print the ones that don't start with a K. That is, skip the ones that start with a K.

```
for name in ('Jonh', 'Jane', 'Karen', 'Susan', 'Karol', 'Mike'):
    if name[0] == 'K':
        continue
    print(name)

# John
# Jane
# Susan
# Mike
```

As you can see, whenever the first character of each name is equal to K, a `continue` statement is executed and the rest of the loop body is skipped. Instead of exiting the main loop, the next iteration is executed.

The else statement in loops

`For` and `while` loops have the possibility of executing a special block of code in certain cases. In `for` loops, you can include an `else` statement at the same level of indentation. The `else` statement block code will execute only in the case that the loop has exhausted all the items of the generator. For example, if the name is equal to 'Mike', let's `break` from the loop. But if there is no 'Mike' item, print a custom message.

```
for name in ('Jonh', 'Jane', 'Karen', 'Susan'):
    if name == 'Mike':
        break
    print(name)
else:
    print('Mike is not here')

# John
# Jane
# Karen
# Susan
# Mike is not here
```

In `while` loops the `else` statement will execute when the boolean condition becomes false. As the case for the `for` loops, the `else` statement needs to be at the same level of indentation as the `while` statement. For example, let's print the integer numbers starting from 3 all the way through 1, using as condition that the number is greater than 0. The `else` statement will execute when the number is no longer greater than 0.

```
i = 3
while i > 0:
    print(i)
    i = i-1
else:
    print('i is no longer greater than 0')

# 3
# 2
```

```
# 1
# i is no longer greater than 0
```

In cases where you have a `break` statement and the loop terminates prematurely, the `else` would not execute:

```
i = 3
while i > 0:
    print(i)
    if i == 2:
        break
    i = i-1
else:
    print('i is no longer greater than 0')

# 3
# 2
```

Conclusion

In this chapter you learned about loops in Python. Loops are a very important subject in every programming language, since they let programmers automate many aspects of a problem. You learned about the `for` and `while` loops and how they work.

You also learned how to use the `break` and `continue` statements, which are useful to either terminate a loop completely or skip certain items.

Finally, the `else` statement can also be used in `for` and `while` loops.

Exercises

1. Make a list of 5 countries belonging to your continent (include the country you live in) and print them on the screen. For this first exercise do this only using the `print()` function, without using loops.
2. Repeat the exercise #1, this time using a `for` loop.
3. Print a list of the first 50 integer numbers, starting at 1. Use the `for` loop and the `range()` function.
4. Repeat exercise #3, this time using a `while` loop. Be careful with making the loop infinite!
5. Going back to exercise #2, include a `break` statement to exit the loop when you reach the country you live in.
6. Now go back to exercise #4. Instead of printing all of the 50 integer numbers, skip the ones that are multiple of 3. Which control statement should you use for this task?

7. Flowchart diagrams can be used with loops. Make a flowchart diagram for the exercise #6.

Chapter 10: Functions

A function is a very useful technique all programmers use in their everyday life. Not only programmers use functions to improve their code, making it work better and even faster, but scientists like mathematicians, physicists, etc, also use functions all the time.

In this article you will learn what a function is, how you can create and use functions in your Python programs, and what are some of the most commonly used Python built-in functions.

What is a function? The mathematical definition

Mathematically speaking, a function is a relation, mapping or rule between two or more sets (numbers) with some particular properties. Let's think of a function relating two sets for simplicity, represented by the variable names x and y . The relation between both sets have to be such that for each value of x there is only one related value of y . The x set is called the *domain* of the function, while the y set is the *range*.

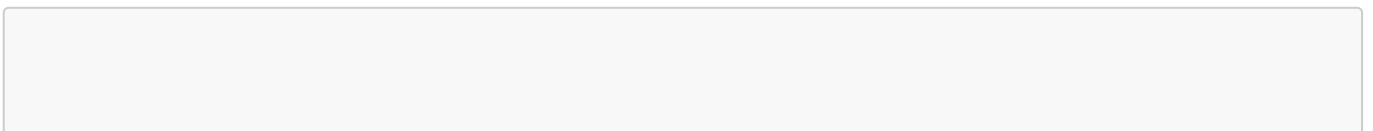
Functions are commonly represented as $y=f(x)$, which reads 'y is equal to f of x'. The variable x is said to be *independent*, while y is *dependent*. This means that to obtain a value of the variable y , the variable x has to be set first.

The letter f represents the rule between variables x and y . The rule can be of various forms; for example a linear function has the form $f(x)=ax+b$, where a and b are given constant numbers. x is the independent variable which can take any real value. After making the operations (a multiplied by x and added to b) a new value is obtained, which is the result assigned to the dependent variable y .

Functions in Python

Let's see how to make a linear function using Python. To create a new function you need to use the keyword `def`, followed by a function name (in this case let's just use f), and finally a set of parentheses containing the independent variable x . In Python, the independent variable is called the **parameter** of the function. A function can have more than one parameter, as you will see later. Finish the statement with a colon `:`.

Below the `def` statement comes the body of the function, which has to be indented, similar to the `if` and `for` statements. In this case we just need to compute a linear function and return its value. The linear function can be something like $2*x+1$ (feel free to try something different yourself). The `return` statement is used for the function to give a value back when is called in the main program.



```
def f(x):  
    return 2*x + 1
```

You have just defined the function `f`. How do you call this function to show a resulting value, given a value of `x`? To call the function, simply write its name (in this case `f`) followed by a value (a number) between parentheses. Make use of the `print` function to show the result on the screen.

```
def f(x):  
    return 2*x + 1  
  
print(f(2))  
  
# 5
```

When you give a concrete value for the parameter, that is called the **argument** of the function you just called. You could have defined the function in a slightly different way. First compute the value of `2x+1`, save the result in a variable called `y`, and lastly return `y`.

```
def f(x):  
    y = 2*x + 1  
    return y
```

Calling the function defined this way will work exactly the same as the previous example.

The difference between a **parameter** and an **argument** can be subtle at first. A parameter is the name used in the function definition at the `def` statement level. An argument is the value you give the parameter when the function is called.

Functions with more than one parameter

You can define a function that takes more than only one parameter. Let's make a function with three numerical parameters, `x`, `y` and `z`. Define the function `f` similar to the previous example, and separate each parameter with a comma.

```
def f(x, y, z):  
    return x+y+z  
  
print(f(1, 2, 3))
```

```
# 6
```

In this example the `return` statement computes the sum over the three arguments. There is no need to include all parameters in the `return` statement. For example:

```
def f(x, y, z):  
    return 2*y*z  
  
print(f(1, 2, 3))  
  
# 12
```

Functions which returns more than one value

Functions can return more than one value. In the `return` statement separate each returned value with a comma. When calling the function (and printing the output) you will get a tuple containing each returned value.

```
def f(x, y, z):  
    return x+y, x+z  
  
print(f(1, 2, 3))  
  
# (3, 4)
```

Functions with arbitrary data types

So far you have seen different examples of functions, all of them working with numerical data types. But with Python you can define functions that takes any data type values as parameters, and also return any data type values.

Let's set an example where a function takes a string (could be a name) and outputs another string with a welcoming message.

```
def hello(name):  
    return 'Welcome ' + name + '. Happy coding!'  
  
print(hello('John'))
```

```
# Welcome John. Happy coding!
```

Default arguments

What happens if you don't give the necessary arguments to a function? You will learn more on this with the exercises at the end of the article, but you can guess already that you may encounter some sort of error. You can give a specific parameter a default value for cases where you don't specify it when calling a function.

To set a default value for a parameter, make an equal sign `=` after its name in the `def` statement and specify the value you want it to take:

```
def hello(name='guest'):
    return 'Welcome ' + name + '. Happy coding!'

print(hello())

# Welcome guest. Happy coding!
```

As you can see, calling the function without any arguments makes the `name` parameter take the value of `'guest'`. But if you call the function giving an argument, like `'Karen'`, the `name` parameter will take that value. The function will ignore the default value:

```
def hello(name='guest'):
    return 'Welcome ' + name + '. Happy coding!'

print(hello('Karen'))

# Welcome Karen. Happy coding!
```

Built-in functions

Python has many functions that are already implemented for you to use directly. You have been using a couple of them throughout the course: the `print()` and the `len()` functions. Let's see how they work in more detail.

The print function

The print function, as you have seen throughout the course, is used to show certain information on the screen. You can use it to print a message using a string, or to show the value of certain

variables, and even combine both.

```
print('Hello World!')
# Hello World!

pi = 3.14
print(pi)
# 3.14

print('The value of pi is ' + str(pi))
# The value of pi is 3.14
```

But you can do more with the print function. Imagine you want to print a list of names. You can do it by using the print function for each of the names, which will result in the list of names printed one below the other:

```
print('John')
print('Jane')
print('Karen')

# John
# Jane
# Karen
```

The names are shown each in a different line by default. What if you wanted to show them in the same line, with each name separated by a comma? The print function can take an additional parameter `end`, which indicates what should be printed at the end of the line. The default value for the `end` parameter is the new line character `\n`, that's why the default behaviour is to print the names in different lines.

Let's repeat the last example, this time giving each print function a value for the `end` parameter. The first two will have an `end` value of `', '`, that is a comma and a space. The last print will have an end value of `'.'`, so that the last one ends with a period.

```
print('John', end=', ')
print('Jane', end=', ')
print('Karen', end='.')

# John, Jane, Karen.
```

Looks much better! Another way to get the same result is to use a single print statement with each name separated with a comma:

```
print('John', 'Jane', 'Karen')
```

```
# John Jane Karen
```

The names are separated with a space by default. You can specify a different separator with the `sep` parameter. Let's give the print function a `sep` parameter with a value of `', '` (comma and space):

```
print('John', 'Jane', 'Karen', sep=', ')
```

```
# John, Jane, Karen
```

So, the `sep` parameter has a default value of `' '`, which is a space character. What about the period at the end? You can include the `end` parameter with a value of `'.'`:

```
print('John', 'Jane', 'Karen', sep=', ', end='.')
```

```
# John, Jane, Karen.
```

And there you have it! You just made the same example in just one line of code.

The len function

The `len()` function is simpler than the `print()` function because it doesn't have any extra arguments. The `len()` function returns the number of items in an object, such as a list.

```
>>> len(['John', 38, 'California'])  
3
```

When the `len()` function is applied to a string, it returns the number of characters of it:

```
>>> len('Hello World!')  
12
```

There are specific cases like *sets*, where you have to pay attention because the number you see may not be equal to what the `len()` function returns:

```
>>> len({'a', 'b', 'c', 'a'})
3
```

In this example the output is 3, and not 4, because the set returns the unique elements of it, which in this case is `{'a', 'b', 'c'}`. The character `'a'` is repeated.

The input function

This is a very interesting and useful function that helps Python programs interact with the user. The `input()` function takes a single string parameter. When the function is called, a message is shown and the program waits for the user to input a string text.

```
>>> input('Enter your name: ')
Enter your name:
```

In this example we are using the terminal but you can try this with a script file and executing it. Enter your name and hit `Enter`:

```
>>> input('Enter your name: ')
Enter your name: DeusDev
'DeusDev'
```

The text you just wrote is shown (returned) on the screen. You can also save what the user inputs in a variable for later use:

```
>>> name = input('Enter your name: ')
Enter your name: DeusDev
>>> print('Your name is', name)
Your name is DeusDev
```

Keep in mind that the `input()` function returns string values. However you can use other Python functions to convert strings to integers, floats, etc, in case the input values represent numbers.

All built-in functions

There are more built-in functions, and they are all listed on the Python official documentation: <https://docs.python.org/3/library/functions.html>. Have some time to go over all of them, at least to have an idea of what Python is capable of.

Conclusion

In this article you learned about Python functions. Python has a number of built-in functions you can use directly. Each function has its own purpose, and you will learn to use them at given time.

Functions can also be built to fulfill your own needs. You learned how to define custom functions using the `def` keyword. The body of a function usually needs a `return` statement in order to give one or more values back.

In the next chapter you will learn about Python methods, which are functions that work for specific objects.

Exercises

The next list of exercises are intended for you to practice what you have learned on this article about functions.

1. In this article the functions defined in the examples all used the `return` statement to return a specific value. Sometimes a function can be defined without a `return` statement. Make a new function that takes one parameter `country`, and prints a message saying `'The country is India'` when the country parameter is equal to the string `'India'`. To do this, the body of the function will only contain a `print` statement. Test your function.
2. Make a new function that computes the square of a given number, $x^2 = x * x$, this is the number multiplied by itself. Include a `return` statement in your function definition, and test the function with different values, including floats and negative.
3. Make a function with two parameters `x` and `y`, and compute the product of both numbers. Test the function against different numbers.
4. Use the same function of the exercise #3, but this time return two values: the product and the sum of `x` and `y`.
5. Make a function that takes a list as its argument, and return the number of elements it contains. Test the function with lists of different lengths.
6. Use the function of exercise #5 but this time call it by using a string as its argument. Is it working as expected? Why or why not?
7. Use the function of exercise #3 and give the parameters default values. Test the new function for different cases.
8. This exercise is intended to show you a special value returned by some functions in special cases. Open a new terminal and enter the Python interpreter. Define a new function without any parameters (you can call it `f` or however you prefer). In the block of the function make a single `return` statement without anything else. If you call the function

like `f()` nothing will be shown. But what happens if you do `print(f())`? What is the function returning?

Chapter 11: Built-in Methods

In the previous part of the Python basic course we have gone through Python functions. There are functions that are bound to certain objects like a string or a list. Those functions let you perform certain actions and in some cases the object changes. These functions are called **methods**.

For example you may want to check if the letters of a **string** are all upper case. With **lists** you can add new items at specified positions. For **dictionaries** you can retrieve all **key:value** pairs or update values. And many more.

In this article you are going to study different **methods** you can use with different Python data types. The methods are said to be **built-in** because they come with Python itself.

Python methods

Python is an object oriented programming language, which means that Python works with **objects**. Objects have **values** and **methods**. You already know what a value is (a specific number, a string, a list, etc).

The topic of object oriented programming is outside the scope of this course, but I mention it so you have a very basic idea of what is behind Python methods. An object, besides having a value, has special functions assigned to it.

String methods

upper

Say you are working with the string **'Hello World!'** and you want to convert it to uppercase like **'HELLO WORLD!'**. While you can do it manually, you can automate this process with a single function like this:

```
>>> s = 'Hello World!'
>>> s.upper()
'HELLO WORLD!'
```

First you define your variable **s**, which in this case it is of string type. To apply the method you write the name of the variable, followed by a single period **.**, and finally the name of the method, in this case **upper()**.

All methods end with parentheses **()**, since they are actually functions (you learned about Python functions in the [previous chapter](#) of this course). Methods can also take parameters as

you will see later in this article.

You could ask yourself if the string which is assigned to the variable `s` of the example has changed. What happens if you print the value of `s` after applying the `upper()` method?

```
>>> s = 'Hello World!'
>>> s.upper()
'HELLO WORLD!'
>>> s
'Hello World!'
```

As you can see the value remains unchanged. If you need to store the value for later use, you need to use a variable:

```
>>> s_changed = s.upper()
>>> s_changed
'HELLO WORLD!'
```

You can also use methods on string values without the need to store them in variables in the first place:

```
>>> 'Hello World!'.upper()
'HELLO WORLD!'
```

All methods applied to strings **return** new values without affecting the original string, being on a variable or not. This is not always the case for other data types like lists. You will deal with methods for other data types shortly.

What happens if you try to use a method that doesn't exist? Python will throw an `AttributeError`:

```
>>> s.money()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'money'
```

lower

There is also a method to convert a string to lowercase letters. The method is named `lower()`, and the way to use it is identical to any other method like the `upper()` you just learned.

```
>>> s = 'Hello World!'
>>> s.lower()
'hello world!'
```

count

You can count how many times a specific character or substring appears in a string with the `count()` method:

```
>>> 'Hello World!'.count('o')
2
>>> 'Lord Lord Lost'.count('Lo')
3
```

find

With the `find()` method you can search for a specific substring. The method will return the index position where the substring has been found, and -1 in case it is not found.

```
>>> 'Hello World!'.find('W')
6
>>> 'Hello World!'.find('ello')
1
>>> 'Hello World!'.find('John')
-1
```

index

There is a very similar method to find a substring which is the `index()` method. This method works exactly the same as the `find()` method, but it raises a `ValueError` in case the substring is not found, instead of returning -1.

```
>>> 'Hello World!'.index('W')
6
>>> 'Hello World!'.index('ello')
1
```

```
>>> 'Hello World!'.index('John')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

replace

Substrings can be replaced using the `replace()` method. This method takes two required parameters: a substring present in the string, and a new string which will replace the former.

```
>>> 'Hello World!'.replace('World', 'John')
'Hello John!'
```

A third optional parameter can be set to determine how many occurrences of the string will be replaced, in case there are more than one. The default is to replace all occurrences.

```
>>> 'Hello World World World!'.replace('World', 'John', 2)
'Hello John John World!'
```

center

You can return a centered string with an optional specified character to be filled with the `center()` method. The default character is a space. The required parameter is an integer number, which is the length of the resulting string.

```
>>> 'Hello'.center(20)
'      Hello      '
>>> 'Hello'.center(20, '-')
'-----Hello-----'
```

format

The `format()` method is very useful when you need to print a message containing variables. There are various ways to make a format string. In this article you are going to learn some of the basic ones.

For example, use curly braces `{}` to place a variable name in your string. At the end of the string add a period and the `format` keyword to apply the method. The `format` method can take the variable names as its parameters in the format of key value pairs: `key=value`.

```
>>> s = "I'm {name} and I am {age} years old.".format(name='John',  
age=27)  
>>> s  
"I'm John and I am 27 years old."
```

Another way to format the same string is to only use the value as the parameter, and leaving the curly braces empty. Python will automatically fill each placeholder with each value in order.

```
>>> s = "I'm {} and I am {} years old.".format('John', 27)  
>>> s  
"I'm John and I am 27 years old."
```

You can use integer values to indicate the order of the parameters. The default order starts at 0:

```
>>> s = "I'm {0} and I am {1} years old.".format('John', 27)  
>>> s  
"I'm John and I am 27 years old."
```

If you need to use a different order, change the values inside the curly braces:

```
>>> s = "I'm {1} and I am {0} years old.".format(27, 'John')  
>>> s  
"I'm John and I am 27 years old."
```

List methods

Let's work with an example list and see what you can do to it. Suppose you are working with a list of daily tasks:

```
todo = ['make coffee', 'take out trash', 'call Sean', 'pay bills']
```

append

Let's add a new task to the **todo** list. The **append** method adds a new item to a list at the end of it.

```
>>> todo.append('buy milk')
>>> todo
['make coffee', 'take out trash', 'call Sean', 'pay bills', 'buy milk']
```

As you can see, the original list is changed (in this case with a new item at the end).

insert

To include a new item in the list but at a specified position, use the **insert** method. The first parameter is the index you want to insert the new element, and the second is the element itself:

```
>>> todo.insert(2, 'go to the gym')
>>> todo
['make coffee', 'take out trash', 'go to the gym', 'call Sean', 'pay bills', 'buy milk']
```

pop

To delete elements from the list, use the **pop** method. This method takes a single optional parameter indicating the index position of the element to be removed. If the index is not given, the default is -1, which is the last element of the list.

```
>>> todo.pop()
'buy milk'
>>> todo
['make coffee', 'take out trash', 'go to the gym', 'call Sean', 'pay bills']
>>> todo.pop(1)
'take out trash'
>>> todo
['make coffee', 'go to the gym', 'call Sean', 'pay bills']
```

Have you noticed that after using the **pop** method the removed element is printed to the screen? That is because this method *returns* the removed element. You could save it to a variable for later use:

```
>>> removed_element = todo.pop()
>>> todo
['make coffee', 'go to the gym', 'call Sean']
```



```
>>> removed_element  
'pay bills'
```

remove

What if you want to remove an element given its value? The `remove` method removes the first element found with a given value.

```
>>> todo.remove('go to the gym')  
>>> todo  
['make coffee', 'call Sean']
```

sort

A very common task in programming is sorting data in a specified way. Let's apply the `sort` method to the original list and see the output.

```
>>> todo = ['make coffee', 'take out trash', 'call Sean', 'pay bills']  
>>> todo.sort()  
>>> todo  
['call Sean', 'make coffee', 'pay bills', 'take out trash']
```

The `sort` method sorted the list of strings in ascending alphabetic order. You can also sort the list in descending order by giving the method the optional argument `reverse=True`.

```
>>> todo.sort(reverse=True)  
>>> todo  
['take out trash', 'pay bills', 'make coffee', 'call Sean']
```

Dictionary methods

Let's work with a dictionary and see what you can do with it. Say you have a dictionary where the keys are the countries and the values are the respective capitals:

```
>>> countries = {'India': 'New Delhi', 'Argentina': 'Buenos Aires',  
'Spain': 'Madrid'}
```

keys

With the **keys** method you can retrieve all the keys present in the dictionary.

```
>>> countries.keys()
dict_keys(['India', 'Argentina', 'Spain'])
```

values

You can also get the list of all the values in the dictionary with the values method.

```
>>> countries.values()
dict_values(['New Delhi', 'Buenos Aires', 'Madrid'])
```

items

The **items** method returns a list of all the key-value pairs of the dictionary. Each key-value pair is in a tuple.

```
>>> countries.items()
dict_items([('India', 'New Delhi'), ('Argentina', 'Buenos Aires'),
('Spain', 'Madrid')])
```

update

You can include new key-value pairs with the **update** method. The **update** method takes a parameter which is in the form of a new dictionary with a key-value pair to be added to the main dictionary. You can include more than one key-value pair at a time.

```
>>> countries.update({'France': 'Paris'})
>>> countries
{'India': 'New Delhi', 'Argentina': 'Buenos Aires', 'Spain': 'Madrid',
'France': 'Paris'}
```

pop

To remove a key-value pair from the dictionary, you can use the **pop** method with a key as its parameter.

```
>>> countries.pop('Spain')
'Madrid'
>>> countries
{'India': 'New Delhi', 'Argentina': 'Buenos Aires', 'France': 'Paris'}
```

Conclusion

In this article you learned some of the most common built-in methods Python has to offer. There are a lot more methods that weren't covered in this article. I invite you to investigate on them and learn what they do.

Exercises

With this exercises you will practice more on built-in methods. Some of them were not covered in the article, but you can learn to use them anyway.

1. Take the string `'Python is beautiful!'`. Return a new string with all the letters uppercase.
2. Make the string `'Python is beautiful!'` look like a title. That is, make the first character of each word uppercase. You can use the `title()` method for this exercise.
3. Check if the string `'Python is the best'` starts with `'Py'`. You can use the `startswith()` method for this.
4. Given the list `[1, 2, 2, 2, 3, 4, 5, 2, 2, 1, 5, 9]`, count how many times the number 2 appears. You can use the `count()` method.
5. Use the same list of exercise #4, and remove all elements from it. You can use the `clear()` method.
6. Take the dictionary `d = {'name': 'Karen', 'age': 22}` and get the value of each key. For this exercise, use the method `get()`, which takes the key as its parameter.
7. Make a list with all the integer numbers from 1 to 100: `[1, 2, 3, ..., 99, 100]`. Use a `for` loop and the `append` list method. You should initialize your list with an empty list first: `[]`.

Chapter 12: Modules

Python has a way to encapsulate a set of objects such as functions, variables, etc, in a single file with a given name. These objects can be used in your programs without the need to include all the code in your script.

In this article you are going to learn how to create a Python module, and how to use it in your programs. You will also learn about some of the most commonly used Python modules and its useful functions.

Python modules

Creating and using a custom module

To make a custom Python module, create a new file and give it a name like **deusdev.py** or whatever name you want it to have. It is important that the file has the **py** extension. This is the file that will contain all the functions to be used later.

In your module file, create a new function. For now let's make a simple function that prints a welcoming message like you did before in this course:

```
# deusdev.py

def greet(name):
    print('Hello {}. Welcome to Python!'.format(name))
```

How can this module with its function be used in another script? Let's suppose the file is saved in the Desktop. Open a new file, also in the Desktop, and give it some name like **test.py**. This will be the file containing the script that will call the function **greet()** from the **deusdev.py** module.

First you need to *import* the module. To do that, use the **import** keyword, followed by the name of the module, in this case **deusdev** (without the **.py** extension). Now that the module is included in your script, you can use the function defined in it.

The function **greet()** can be used in the same way as a method. Use the module name **deusdev** followed by a period, and then the name of the function **greet()**.

```
# test.py

import deusdev
```

```
deusdev.greet('John')

# Hello John. Welcome to Python!
```

Another way to use a module's function is to assign it to a variable. You can assign the function using `deusdev.greet` (without parentheses) to a variable called `greet_func` (or something different). Then you can call the function using `greet('John')`.

```
# test.py

import deusdev

greet_func = deusdev.greet
greet_func('John')

# Hello John. Welcome to Python!
```

There are other ways you can import functions from your module. To import the `greet` function from the `deusdev` module, use the keyword `from`, followed by the module's name `deusdev`, then `import` and finally the function you want to import, in this case `greet`. Then use the function without the module's name.

```
# test.py

from deusdev import greet

greet('John')

# Hello John. Welcome to Python!
```

If you have more than one function in your module, you can import each function separating them with a comma. Let's add a new function to the module that checks if a given number is even:

```
# deusdev.py

def greet(name):
    print('Hello {}. Welcome to Python!'.format(name))
```

```
def isEven(n):  
    return n%2 == 0
```

Then in your main file import and call both functions:

```
# test.py  
  
from deusdev import greet, isEven  
  
greet('John')  
# Hello John. Welcome to Python!  
  
print(isEven(41))  
# False
```

You can use the wildcard `*` to import all the functions at once, instead of writing each of them:

```
# test.py  
  
from deusdev import *  
  
greet('John')  
# Hello John. Welcome to Python!  
  
print(isEven(41))  
# False
```

You can also use the `as` keyword to give the module a different name to use it in the main program. This can be useful when module names are somewhat long and you want to reduce the typing in your program.

```
# test.py  
  
import deusdev as dd  
  
dd.greet('John')  
# Hello John. Welcome to Python!
```

The `as` keyword can be combined with the `from` keyword to import a module's function with a different name:

```
# test.py

from deusdev import greet as greetings

greetings('John')

# Hello John. Welcome to Python!
```

pycache files

You may have noticed that when you run your scripts importing a custom module, a new folder is created with the name **pycache**. Inside this folder there is a file with the name **deusdev.cpython-311.py** (you may see it with a slightly different name depending on your module's name and the Python version you are using). This is done to make the loading of modules faster, using something called *cache*. I won't get into much details in this course, but you can read more about this in the [official Python documentation](#).

Next you are going to learn about some of the most common Python's built-in modules.

The random module

The [random module](#) is used to generate pseudo-random numbers. Let's see some of the functions you can use with this popular module.

This time you can work with the Python interpreter, without the need to make a new file. First, import the `random` module.

```
>>> import random
```

Once the module is imported, you can begin to use its functions. Let's see some of them.

`random.random()`

The `random()` function returns a random floating point number between 0 and 1. 0 is included, meaning that sometimes you can get that exact value, but the 1 is not included in the range, so you will never get a 1 with this function.

```
>>> random.random()
0.5727401830032317
>>> random.random()
0.4977649539769565
>>> random.random()
0.11762935447218192
```

`random.uniform(a, b)`

The `uniform()` function is similar to the `random()` function, but with `uniform()` you can return a floating point number between two arbitrary numbers `a` and `b`.

```
>>> random.uniform(4, 10)
9.377423126967187
>>> random.uniform(4, 10)
6.98125740847348
>>> random.uniform(4, 10)
5.565696392526846
```

`random.randrange(start, stop, step)`

The `randrange()` function returns a random integer value between `start` (optional, default is 0) and `stop` (not included in the range), and you can give an optional parameter `step` (default is 1).

```
>>> random.randrange(5)
0
>>> random.randrange(5)
3
>>> random.randrange(5, 50)
17
>>> random.randrange(5, 50)
33
>>> random.randrange(5, 50, 10)
5
>>> random.randrange(5, 50, 10)
15
```

`random.randint(a, b)`

This is equivalent to the `randrange()` function with `a` and `b+1` as the parameters `start` and `stop`, because in this case `a` and `b` are both included.

```
>>> random.randint(5, 10)
6
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
10
```

`random.choice(seq)`

With the `choice()` function you can randomly select an element from a sequence, such as a list. For example, say you have a list of names and you want to select one of them randomly:

```
>>> random.choice(['John', 'Karen', 'Sean', 'Jane'])
'Karen'
>>> random.choice(['John', 'Karen', 'Sean', 'Jane'])
'Karen'
>>> random.choice(['John', 'Karen', 'Sean', 'Jane'])
'Jane'
>>> random.choice(['John', 'Karen', 'Sean', 'Jane'])
'Karen'
>>> random.choice(['John', 'Karen', 'Sean', 'Jane'])
'Sean'
```

`random.shuffle(seq)`

The `shuffle()` function takes a sequence, such as a list, and shuffles it (mixes its elements) in place (meaning that it doesn't return a new sequence).

```
>>> names = ['John', 'Karen', 'Sean', 'Jane']
>>> random.shuffle(names)
>>> names
['Sean', 'Jane', 'Karen', 'John']
```

The math module

The [math module](#) comes in handy when dealing with special mathematical operators. Let's import the math module to start.

```
>>> import math
```

Keep in mind that this module only works for real numbers. If you need to work with complex numbers, use the [cmath module](#).

Let's see some of the functions the math module has for you.

`math.ceil(x)`

The `ceil(x)` function returns the smallest integer value greater than or equal to the argument `x`.

```
>>> math.ceil(5.2)
6
>>> math.ceil(5)
5
>>> math.ceil(1.9)
2
```

`math.factorial(n)`

This function returns the factorial of the integer value `n`.

```
>>> math.factorial(5)
120
>>> math.factorial(3)
6
>>> math.factorial(0)
1
```

`math.gcd()`

Returns the greatest common divisor between two numbers. Since the Python version 3.9 this function can take more than two parameters.

```
>>> math.gcd(4, 18)
2
>>> math.gcd(6, 18)
6
>>> math.gcd(4, 18, 8)
2
```

`math.exp(x)`

Returns the [number e](#) to the power of `x`. The number `e` is approximately equal to 2.718.

```
>>> math.exp(2)
7.38905609893065
>>> math.exp(0)
1.0
```

`math.sin(x)`

Returns the sine of the number `x`.

```
>>> math.sin(7.32)
0.8607873878989017
>>> math.sin(0)
0.0
```

`math.pi`

Returns the constant [number pi](#). Although the number pi has an infinite number of decimals, this module gives this number with a fixed number of decimals.

```
>>> math.pi
3.141592653589793
```

The numpy module

The [numpy module](#) is widely used among developers who work with data analysis. Go ahead and import the [numpy](#) module.

```
>>> import numpy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'numpy'
```

Ups! It seems that **numpy** is not included with Python. Before using the **numpy** module, you have to install it. The [official numpy installation website](#) gives different options to install it. In this article I'm going to use **pip**. Execute **pip install numpy** in the terminal (outside the Python interpreter):

```
PS C:\> pip install numpy
Collecting numpy
  Downloading numpy-1.24.2-cp311-cp311-win_amd64.whl (14.8 MB)
    ----- 14.8/14.8 MB 17.7 MB/s
eta 0:00:00
Installing collected packages: numpy
Successfully installed numpy-1.24.2
```

If everything goes well, you should see a message similar to the one in the example, with information such as the module version that has been installed.

Let's go to the Python interpreter and try to import the numpy module, now that you have it installed.

```
>>> import numpy
>>>
```

No errors, that is good! Actually, let's import the module with a different name, which is the most commonly way to do it:

```
>>> import numpy as np
>>>
```

Now you are ready to use the numpy functions with the **np** name. I encourage you to check you the [numpy user guide for beginners](#). The website has a lot of examples and a very good and extensive documentation. In this article you are going to learn the very basics of numpy.

`numpy.array()`

You can construct arrays that are similar to Python lists with the numpy module. They are called *numpy arrays*. For example, you can construct an array from a list of values:

```
>>> np.array([23, 14, -2, 0, 9])
array([23, 14, -2,  0,  9])
```

You can also create an array filled with zeros:

```
>>> np.zeros(4)
array([0., 0., 0., 0.])
```

`numpy.arange()`

This function is used to construct a numpy array with a range of values. This works in a similar way as the `range()` function.

```
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(2, 10)
array([2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(4, 31, 3)
array([ 4,  7, 10, 13, 16, 19, 22, 25, 28])
```

`numpy.linspace()`

Another way to define a numpy array is with the `linspace()` function. This function takes a `start` and `end` parameters, and a third parameter which is the number of elements the array will contain. The values will be evenly spaced between each other.

```
>>> np.linspace(1, 15, num=5)
array([ 1. ,  4.5,  8. , 11.5, 15. ])
```

There is a lot you can do with the numpy module. Feel free to go over the documentation if you are interested, and have fun!

Conclusion

In this article you learned about the very basics of Python modules. You learned how to create a custom model, and how to import a model in your main Python scripts. Finally, you learned about three of the most popular Python modules and its basic functions.

This is the last chapter of the Python basic course. There is an extra chapter which is intended for you to practice what you have learned all the way by making a couple of practice projects. Finally, don't forget to take the exercises on this chapter too!

Exercises

This set of exercises are intended for you to practice the concepts learned on this chapter about Python modules.

1. Create a new file with the **.py** extension. Give it a name of your choosing, like **deusdev.py**. This is going to be used as a Python module. In this new Python module, create a function that prints a message saying something about the module. For example, the message can be an explanation of what your module can do, like **'This Python module has a set of functions that solves some mathematical equations'**.
2. Make a second function in your module. Make this function compute some mathematical formula with parameters. For example, you could make it compute the [discriminant for a quadratic equation](#). That is, given three float parameters **a**, **b** and **c**, the function should return the square root of $b^2 - 4ac$.
3. Make a new file with a name of **main.py** or something similar. This is going to be the main script where you call the functions from your module. What is the first thing you need to do in order to be able to use the functions in your new module?
4. Call the function you defined in exercise #1. You should be able to see the message printed on the screen.
5. Call the function you made in exercise #2. Use different parameters and test if everything is working as expected.
6. In this exercise you are going to learn how to use the [random.gauss\(\)](#) function from the [random](#) module. This function can take two optional parameters: the mean and the standard deviation from a normal distribution. Read the documentation to see more details, and try the function with different values to see the outputs.
7. Go to the [math module](#) page and look for a function to compute the logarithm of a number. Read the docs and use the function with different parameters to see the outputs.
8. In this exercise you will learn about the [datetime module](#). Go ahead and import the datetime module in a new Python interpreter or a new Python file. After importing the module, write `dir(datetime)` and hit **Enter**. The [dir function](#) will give you a list of names available from that module. For example, try `datetime.datetime.now()`, you should see the current date and time. Try to investigate what other things you can do with this module from the documentation.

Chapter 13: Practice Projects

This section of the basic Python course is intended for you to practice what you have learned so far. There is a lot you can do with what you already know, and in this short article you are getting a couple of ideas for practice projects. Let's get started and have fun!

Rock, paper, scissors game

This is a two player game where each of them randomly chooses between rock, paper or scissors. In this case you can make a user play against the computer. You can read more about this popular game in the [wikipedia article](#). The basic rules are:

- Rock beats scissors
- Paper beats rock
- Scissors beats paper

The Python program can work like the following after executing the script:

```
Game number 1
Rock (R), paper (P) or scissors (S)? (Q to quit): R
You have chosen Rock, while the computer chose Paper
You lost

Game number 2
Rock (R), paper (P) or scissors (S)? (Q to quit): p
You have chosen Paper, while the computer chose Rock
You won!

Game number 3
Rock (R), paper (P) or scissors (S)? (Q to quit): S
You have chosen Scissors, while the computer chose Rock
You lost

Game number 4
Rock (R), paper (P) or scissors (S)? (Q to quit): T
Incorrect input. Please try again.

Game number 4
Rock (R), paper (P) or scissors (S)? (Q to quit): r
You have chosen Rock, while the computer chose Scissors
You won!
```

```
Game number 5
Rock (R), paper (P) or scissors (S)? (Q to quit): r
You have chosen Rock, while the computer chose Scissors
You won!

Game number 6
Rock (R), paper (P) or scissors (S)? (Q to quit): P
You have chosen Paper, while the computer chose Scissors
You lost

Game number 7
Rock (R), paper (P) or scissors (S)? (Q to quit): q
Thank you for playing!
You won 3 out of 6 games (50.00%)
```

The user plays against the computer one game after another as long as the user doesn't choose the option to quit, which can be the letter Q. Here is a list of things to consider for the Python script:

1. In order to play games indefinitely until a condition is met (choosing Q to quit) you can make use of a `while` loop. There are a couple of ways to do it, but one popular way to make an infinite loop is by using the `while True`. Keep in mind that if you don't use a proper ending condition in the loop, you may end up with an infinite loop.
2. The program needs to prompt the player (you) for either rock, paper or scissors. Remember you can use the `input()` function for this.
3. The computer also needs to choose between rock, paper or scissors. You could make use of the `random` module to randomly choose between the three options, particularly with the `choice()` function.
4. In each game you need to check who won, or if there was a tie. For this you could make a series of `if-else` and possibly `elif` statements. For example, if the user chooses Rock and the computer chooses Scissors, then the user won the game.
5. You could keep a score of the game by using a variable `score`, initially being equal to 0. Each game you win adds up one point to the final score. Remember you can increase a variable with `score = score + 1` or `score += 1`.

Countdown timer and clock

In this simple project the idea is to prompt the user for a number of seconds and show a countdown timer with minutes and seconds. The total number of seconds will be an integer number.

After running the script you should be looking at something like this:


```
Enter the time in seconds: 62
You have entered 62 seconds
01:02
01:01
01:00
00:59
00:58
00:57
...
00:03
00:02
00:01
00:00
Time's up!
```

First, you can print a message showing the number of seconds the user just entered. This can sound silly for a small and simple program like this one, but it is a good idea for larger projects to show the user what the inputs are as a way to confirm.

Then the countdown begins. In the example above the user entered 62 seconds. That means that the time can be split in 1 minute and 2 seconds. Then, the first line should be something like **01:02** which is the standard format in which a digital clock shows the time.

As the time passes, the number of seconds decreases by one, and when the seconds reach zero the number of minutes decreases, at which point the seconds go all the way up to 59.

The program stops when the total number of seconds reaches zero. Then you can show a message indicating that the time is up.

A couple of things to consider when you work on this small project:

1. Use the `input()` function to prompt the user for the total number of seconds. Remember what is the data type that the `input()` function returns? The `input()` function has been covered in the [functions chapter](#) of this course. You will need to convert this to an integer number for later use.
2. If you need to compute the number of minutes and seconds from the total number of seconds, you could make use of the integer division and modulus operators. These topics were covered in the [terminal and python interpreter](#) article.
3. You learned about Python modules in a [previous article](#) from this course. There is a module called `time` which you can use in this project to wait for one second between each pass. Specifically, look for the `sleep()` method of this module.