AUGUST 22, 2023 / #PYTHON

The Python Code Example Handbook

Farhan Hasin Chowdhury

Very few programming languages are as universally loved as Python. The brainchild of Dutch programmer Guido van Rossum, Python is easy to learn, powerful, and an utter joy to work with.

Thanks to its popularity, video and written resources about Python are plentiful. This handbook, however, tries to be a bit different by not being a definitive guide to the language.

Instead you'll learn about all the topics that I consider to be the language fundamentals with a lot of code examples.

I haven't discussed object-oriented programming in this handbook because I believe it to be a very broad subject deserving of its own separate handbook.

Without any further ado, let's jump in!

Table of Contents

- Prerequisites
- How to Setup Python on Your Computer
- How to Install a Python IDE on Your Computer
- How to Create a New Project on PyCharm
- How to Write the Hello World Program in Python
- How to Initialize and Publish a Git Repository From PyCharm
- How to Work With Variables and Different Types of Data in Python
- How to Work With Simple Numbers in Python
- How to Take Inputs From Users in Python
- How to Work With Strings in Python
- What Are the Sequence Types in Python?
    - Lists in Python
    - Tuples in Python
    - Ranges in Python
    - How Indexing Works in Python
- What Are the Iterable Types and How to Use them for Loops in Python

- What Are Some Common Sequence Type Operations in Python?
  - How to Use the in Operator in Python
  - How to Use the + and * Operators with Sequence Types in Python
  - How to Use the len(), min(), and max() Functions in Python
- What Are Some String Type Operations in Python?
  - How to Capitalize Strings in Python
  - How to Convert Strings to Lower Case or Upper Case in Python
  - How to Count the Number of Occurrences of a Substring in a String in Python
  - How to Split and Join Strings in Python
- How to Write Conditional Statements in Python
- What are Relational and Logical Operators in Python?
- What Are Assignment Operators in Python?
- What Is the Set Type in Python?
- What Is the Mapping Type in Python?
  - What Are Dictionary View Objects in Python?
- How to Write Functions in Python

    - How to Work with local, nonlocal and global Variables in Python
    - How to Pass a Variable Number of Arguments to a Function Using *args and **kwargs in Python
- What Are Modules in Python?
- How to Use the Python Documentation Efficiently
- What's Next?
    - Object Oriented Programming
    - Algorithms and Data Structures
    - Django
    - Qt
    - PyGame
    - Data Science
- Conclusion

Prerequisites

You don't need to know any other programming language for this book, but knowing one may help you understand the basics of Python.

Other than that you'll need to be efficient enough with your choice of operating system to download and install new software, and gain administrative access if needed.

Computer

Installing Python on your computer is a very straightforward process. In fact, if you're on a Linux system, Python should already be installed.

Open up your terminal window and execute the following command:

```
python --version
```

If Python is installed on your system, you'll get an output like `Python 3.10.4` or some other minor version.

Although most of the modern Linux distribution use Python 3 as default, some of the older distributions may still use Python 2 by default.

So if the aforementioned command refers to Python 2, try out the following command:

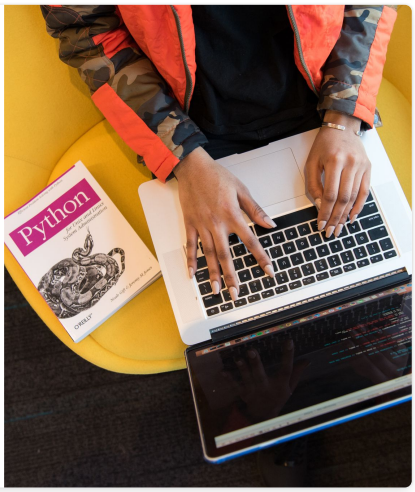
```
python3 --version
```

I'd also suggest that you check for updates on your Linux distribution and install any new updates for Python.

How to Install Python 3 on Mac and Update the Version with Pyenv – MacOS Homebrew Command Guide

When using Python, you may install different version variations for differentprojects. But...

Dillion Megida • freeCodeCamp.org

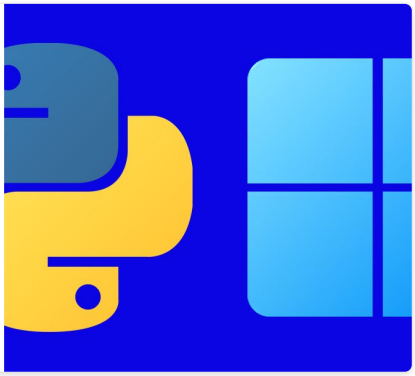


Finally, for Windows, I'd suggest you follow an article by Md. Fahim Bin Amin and properly install the latest version of Python.

How to Install Python on Windows

Before trying to run any Python program in your Windows operating system, you'llneed t...

Md. Fahim Bin Amin • freeCodeCamp.org



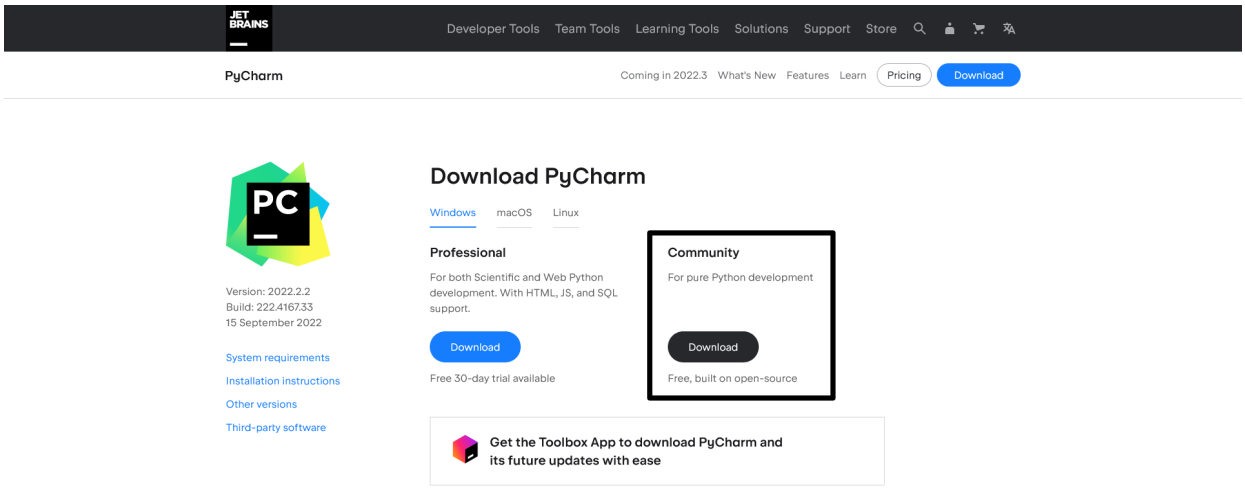
As long as you have a Python 3 version installed, you're good to go.

Much of your experience as a developer will depend on what program you've chosen to write your code in. A good integrated development environment (IDE) or Code Editor can really boost your productivity.

These days Visual Studio Code has become the go to code editor for all languages and platforms. But for the sake of simplicity, we'll use PyCharm in this book.

If you'd like to use VS Code, have written a full-length article on how to configure Visual Studio Code for Python development. Feel free to check that out if you do not mind configuring your editor manually.

The professional edition of the IDE can cost you $89.00 per year but there is also a free and open-source community edition. Head over to the PyCharm download page.



Download PyCharm page

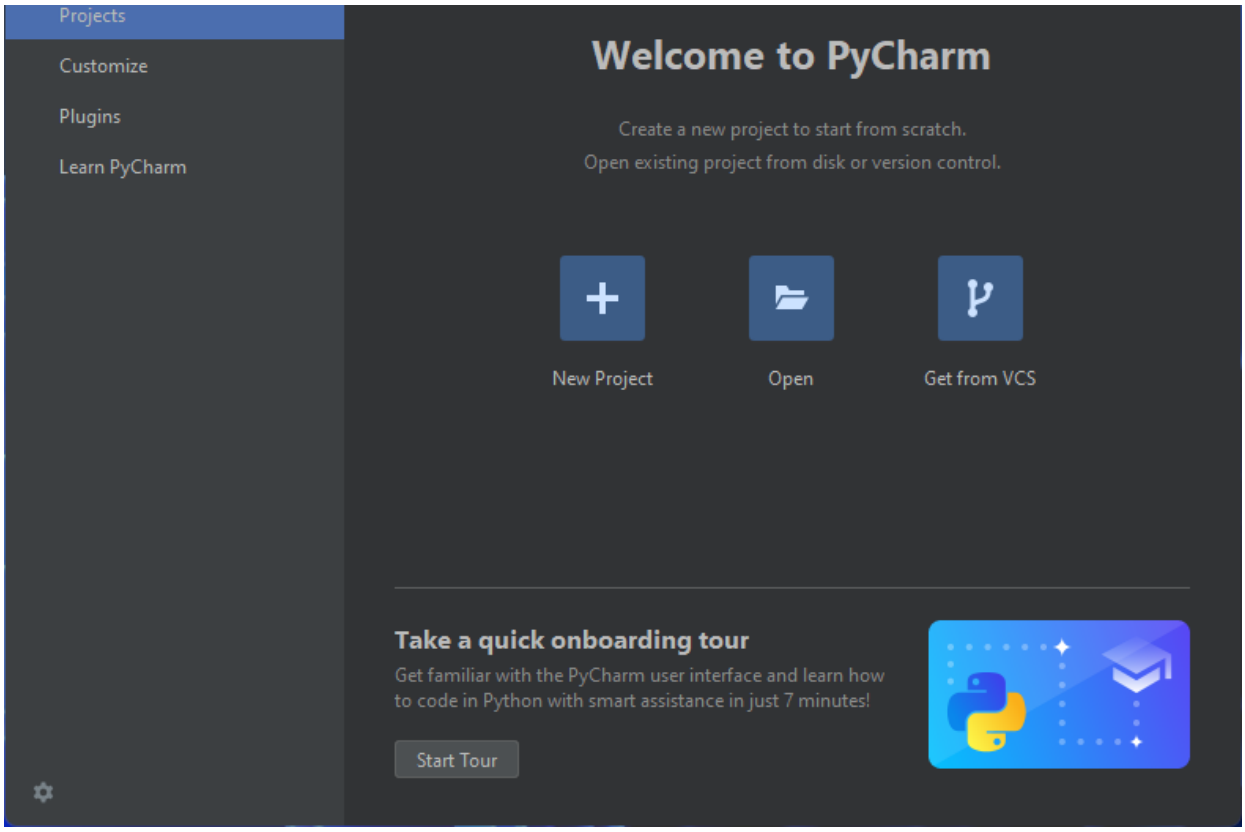
megabytes.

On Windows you'll get an executable installer, on macOS you'll get an Apple disc image, and on Linux you'll get a TAR archive.

I won't demonstrate the installation process in this book since it's similar to installing any other software on your machine.

Once installed, you can start the IDE from your start menu/app launcher. On your first launch, you'll be given the chance to configure a few things. I'd suggest you keep the defaults.

Once the configuration wizard ends, you should see the following welcome window:

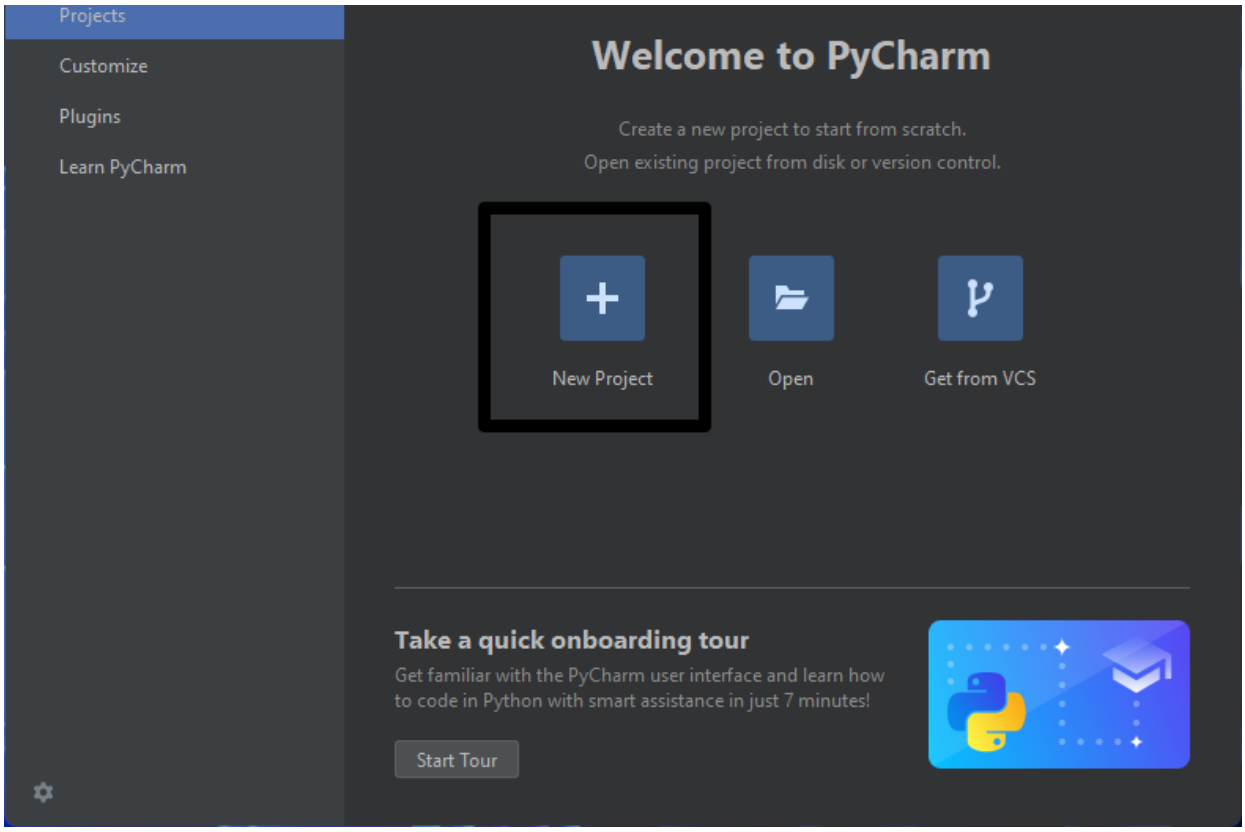


Welcome to PyCharm screen - with options to start a new project, open a project, or get one from your VCS

Picking one IDE or code editor instead of the other one will not affect your experience with following this handbook, so feel free to use whatever you feel comfortable with.

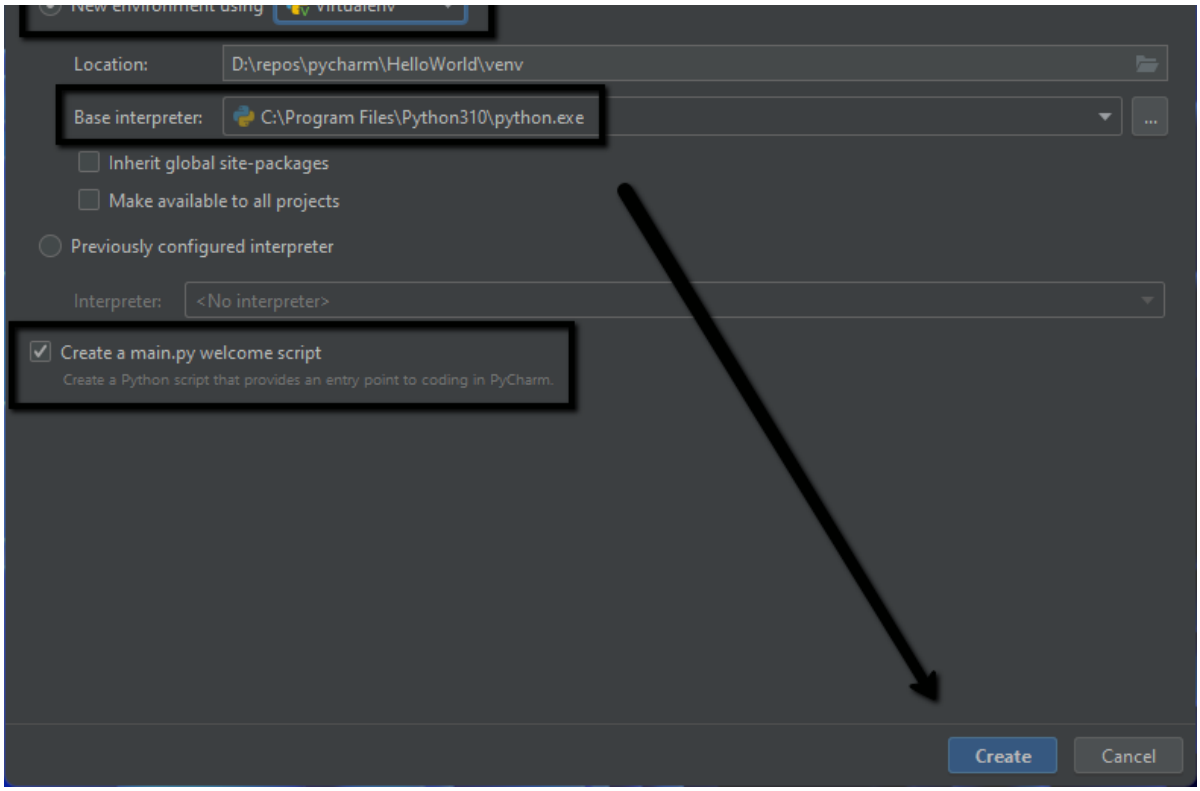
How to Create a New Project on PyCharm

If you have the welcome window open from the previous section, click on the "New Project" button.



Start a new project in PyCharm

In the next step, pick a location to store your project:



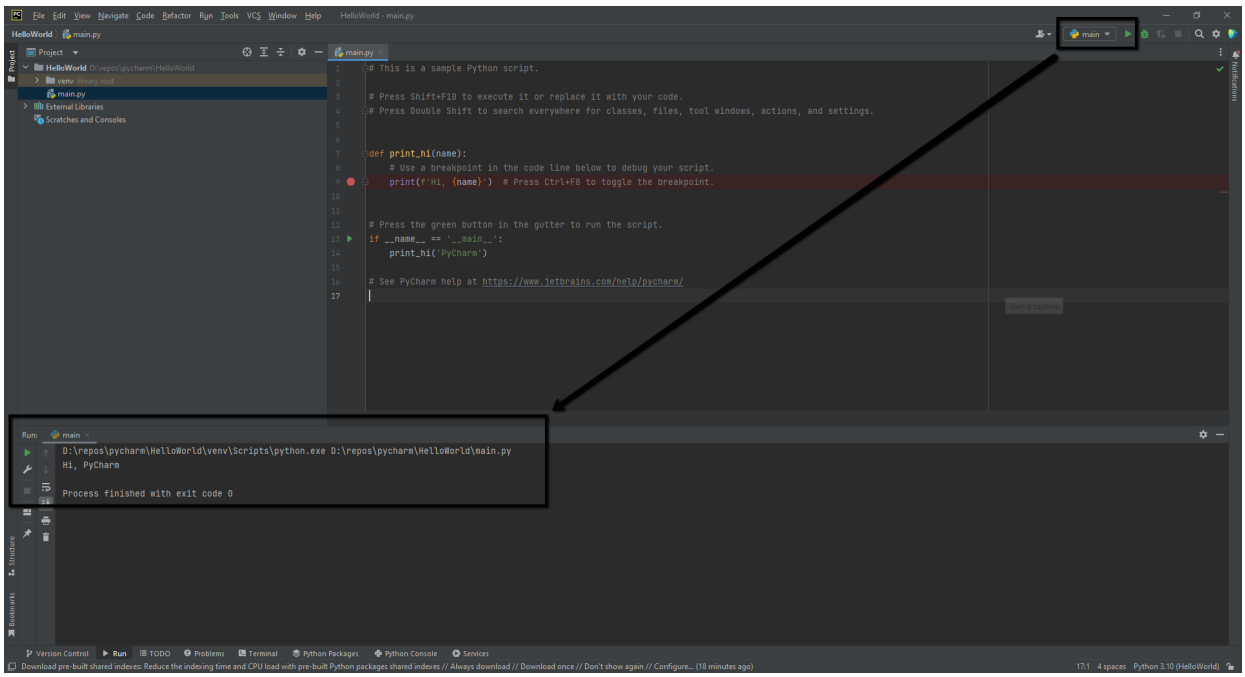
In the location input box, the `HelloWorld` part is the name of the project. Then make sure you have "New environment using Virtualenv" selected. Then, make sure that the correct version of Python is selected from the "Base interpreter" dropdown.

Virtualenv is a program that can create isolated Python environments from a given base interpreter. This is very helpful because later on when you'll work on multiple Python projects, their dependencies may conflict with each other.

Creating isolated environments for each project will solve that issue and it'll also keep your global Python installation free from any unnecessary package installation.

Since this may be your first Python project, I'd suggest you leave the "Create a main.py welcome script" option checked. Once

done, the IDE should open the project automatically for you.



You can use the play button at the top right corner to run the code. The button is configured to run the "main.py" file by default.

That's why you can see "main" written by its side. You can write your custom configuration as well, but that's a topic for a later section.

Hello

features as you go forward.

How to Write the Hello World Program in Python

Continuing on from the last section, open up the "main.py" file and replace all the preexisting code with the following line of code:

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

# Hello, World!
```

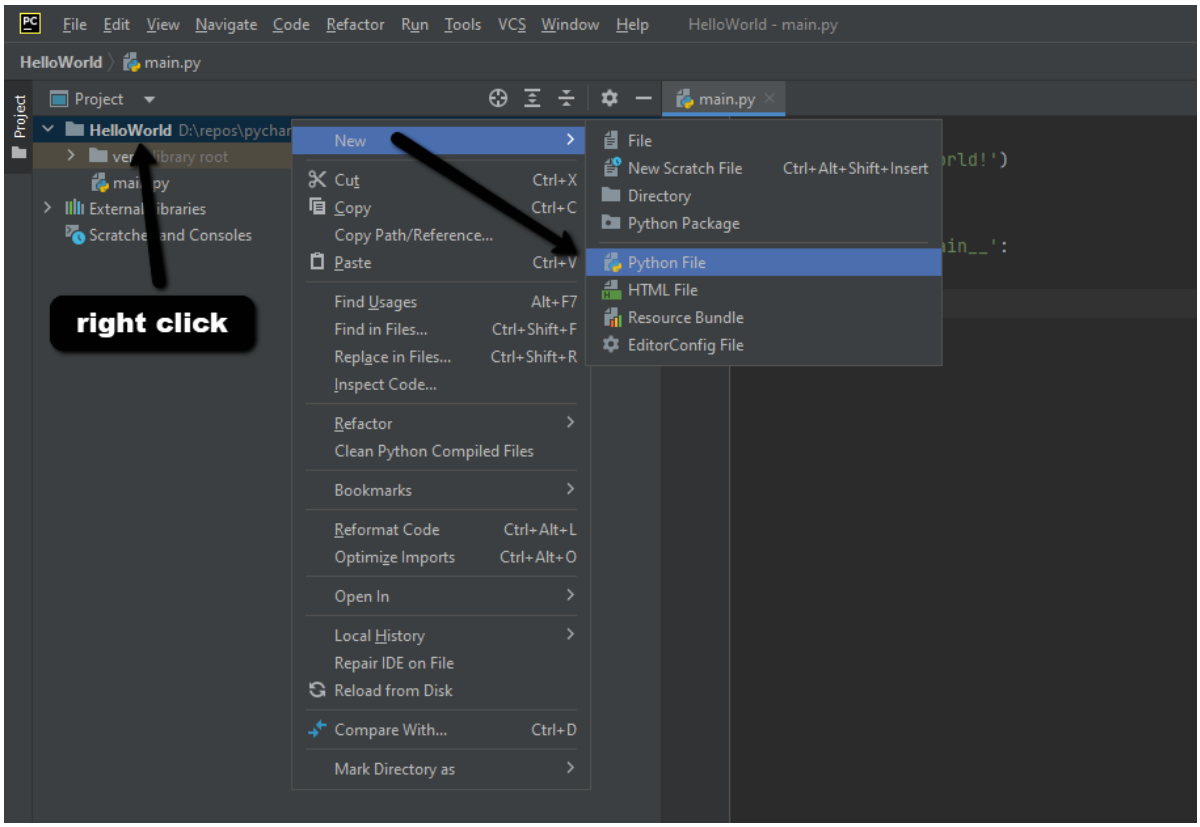
The `print()` function prints out anything that you pass into the set of parenthesis. You don't have to name your Python file "main.py" specifically. It's just a way to let you know that this is the entry point to this program.

That's all you need to write the simplest executable program in Python. But there is even a better way to do it. Update the code as follows:

```
def main():
    print('Hello, World!')
```

As you keep working on bigger projects, you'll eventually have more than one Python file in your project and this way of writing a script can be useful.

To simulate a bigger project, create another Python file by right clicking on the "HelloWorld" project name and selecting "Python File" under the "New" sub menu.



Name your file something like "library" and press enter while "Python file" is highlighted in the list of file types.

A new file with the name "library.py" will show up on your project folder. Put the following line of code inside the file:

```
def greet():
    print('Hello, World!')
```

This is a very simple function that prints out "Hello, World!" on the console. You can `import` and use this function in your "main.py" file.

To do so, update the code for "main.py" file as follows:

```
from library import greet

def main():
    greet()

if __name__ == '__main__':
    main()

# Hello, World!
```

Right now in your project, you have two types of Python file. You have the "main.py" file which is a script. In other words, you can run this file.

Then you have the "library.py" file which is a library. In other words, it houses a number of useful functions and variables that you can import inside other Python files.

Now imagine you have hundreds of files in your project and they more or less look the same. How would someone else find the entry point to the program?

The easiest way would be to perform a search for the line `if __name__ == '__main__'` on the entire project. This makes your code a lot more readable.

Now that I have you convinced that this is the way to go, let me explain what is actually going on here.

The `__name__` is a special Python variable. In case of a script, the value of this variable will be `__main__` and in case of a library, its value will be the name of that file.

So in the aforementioned program, the value of `__name__` inside the "main.py" file will be `__main__` and `library` inside the "library.py" file.

If you change the name of then "main.py" file to something else, the value will still be `__main__` because it's a script.

In languages like C/C++/Go/Java, you'll have a specified `main` function. That function will be the entry point to the program.

Since Python doesn't have anything like that, the usage of the `if __name__ == '__main__'` expression enforces a sense of a specified entry point to your program.

It tells the programmer and the IDE that this script is for execution (not for importing inside other Python files).

How to Initialize and Publish a Git Repository From PyCharm

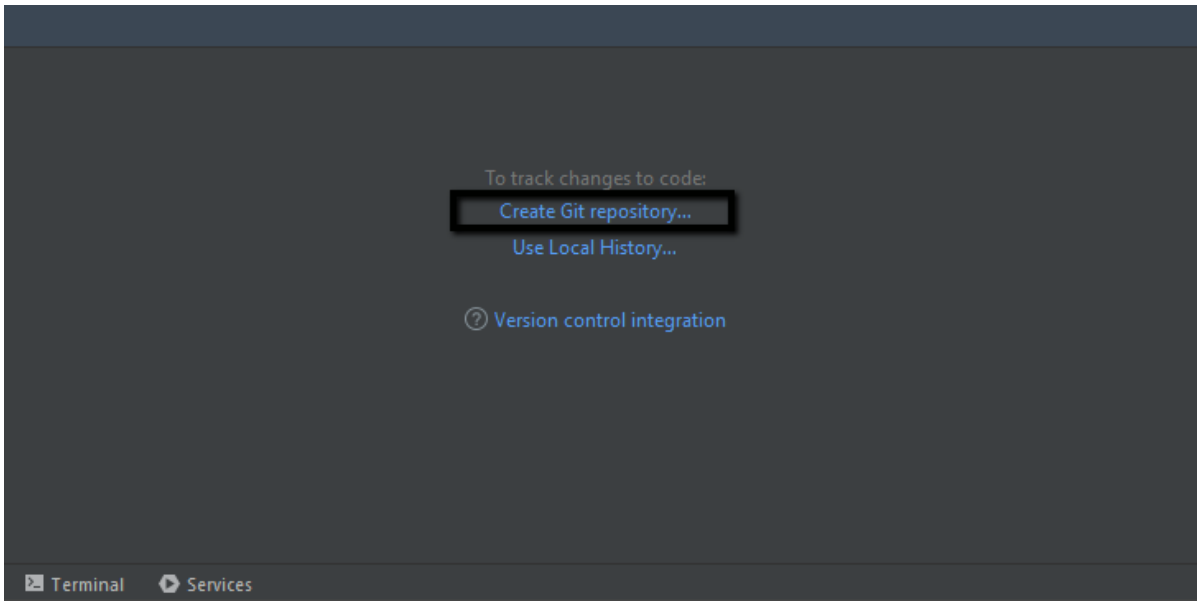
You may already be familiar with Git and know how to initialize a new repository. If you prefer using some other Git client, that's totally fine.

However I think knowing how to make commits right from your IDE can boost your productivity.

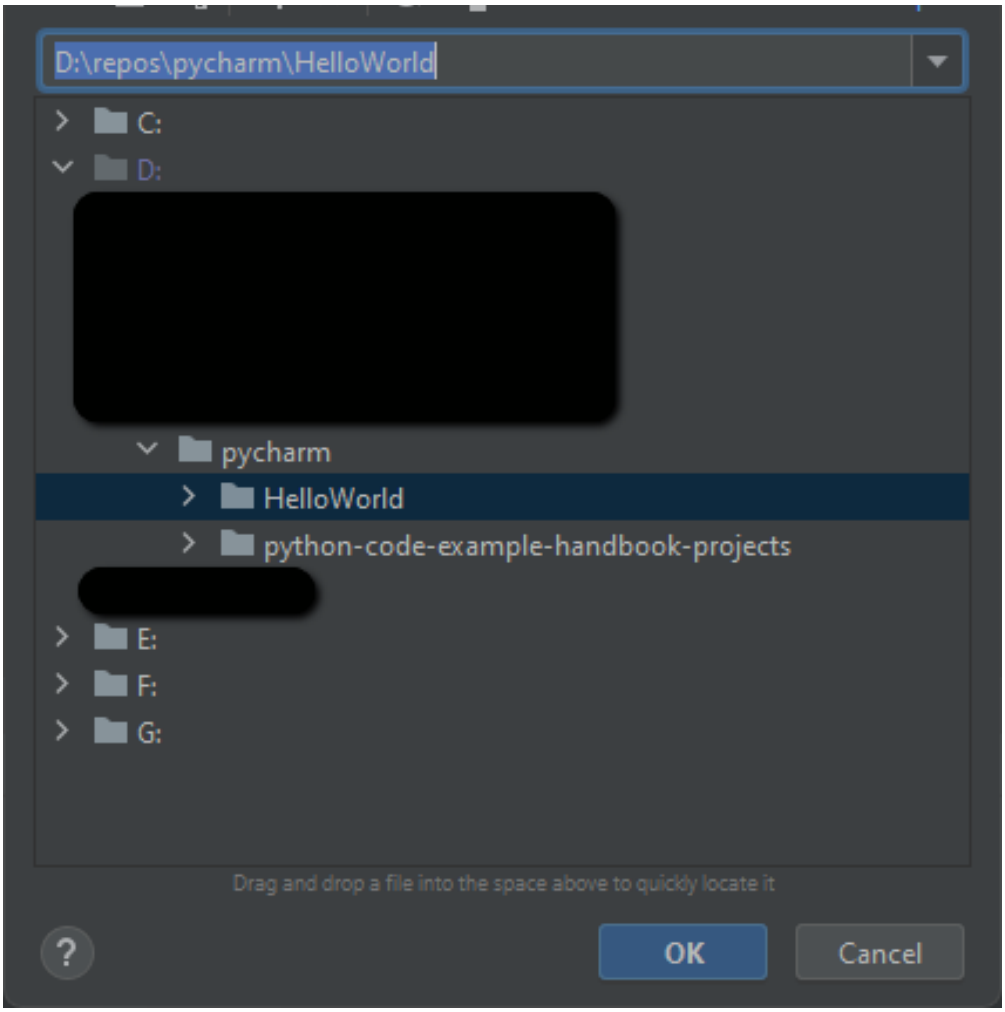
Keep in mind you'll need to have Git installed and configured on your system. If you don't have that, this article by Bolaji Ayodeji may come in handy.

Now, continuing on from the last section, if you look at the bottom of your IDE, you should see a "Version Control" tab.

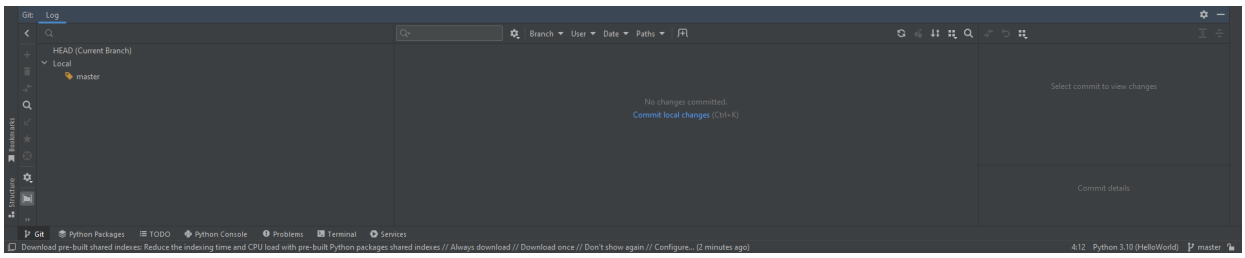
Click on it and you should switch to the version control tab. Now click on the "Create Git repository..." link.



PyCharm will ask you where you want to initialize the new repository. Make sure you're picking the right folder.

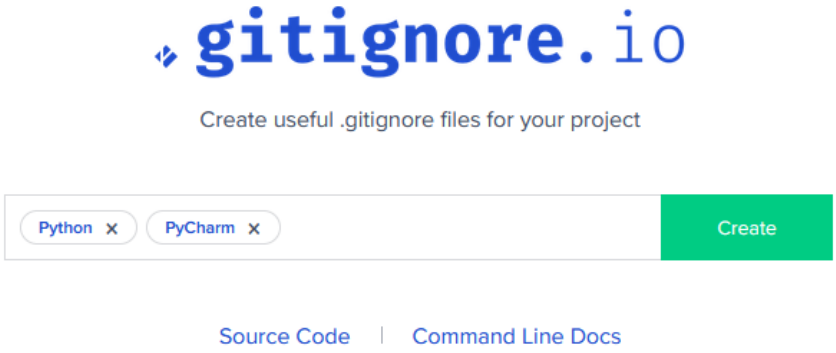


As soon as you press the "OK" button, the "Version Control" tab will change to a "Git" tab.



At it's current state, there are no commits. Before you make your first commit, I'd suggest you add a ".gitignore" file so that no unwanted file gets to the repository.

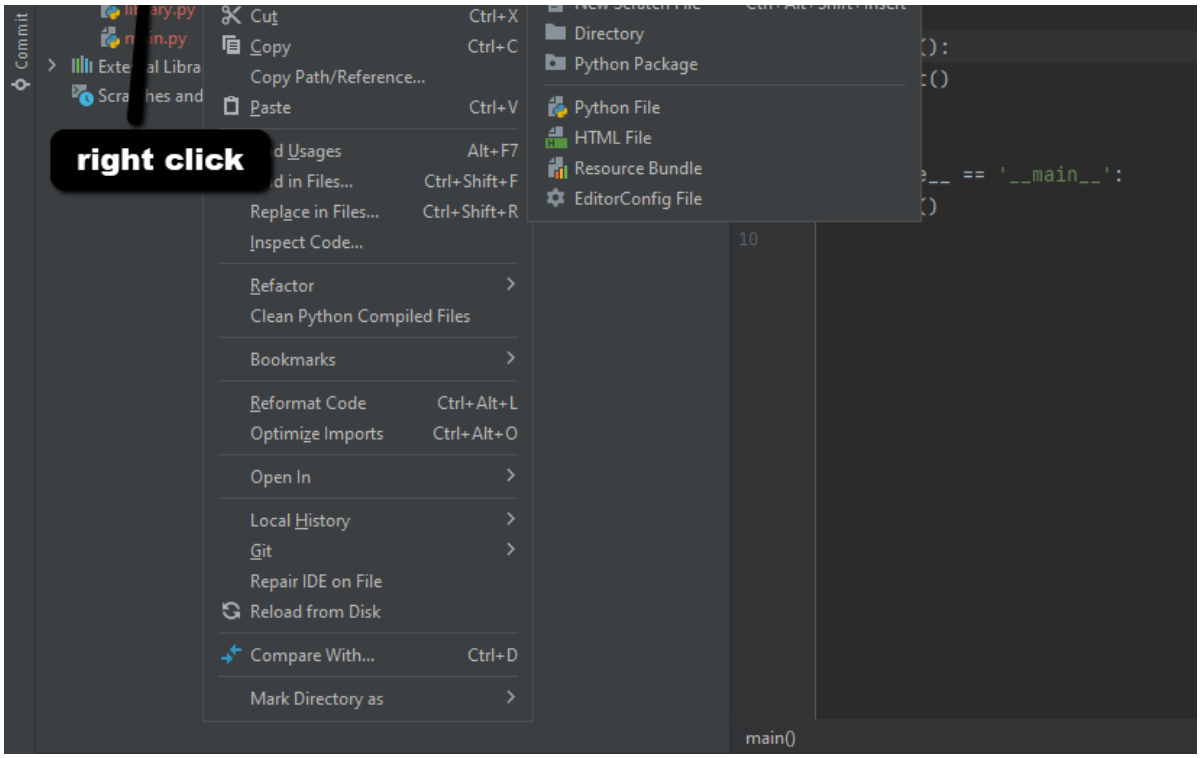
technologies from this website.



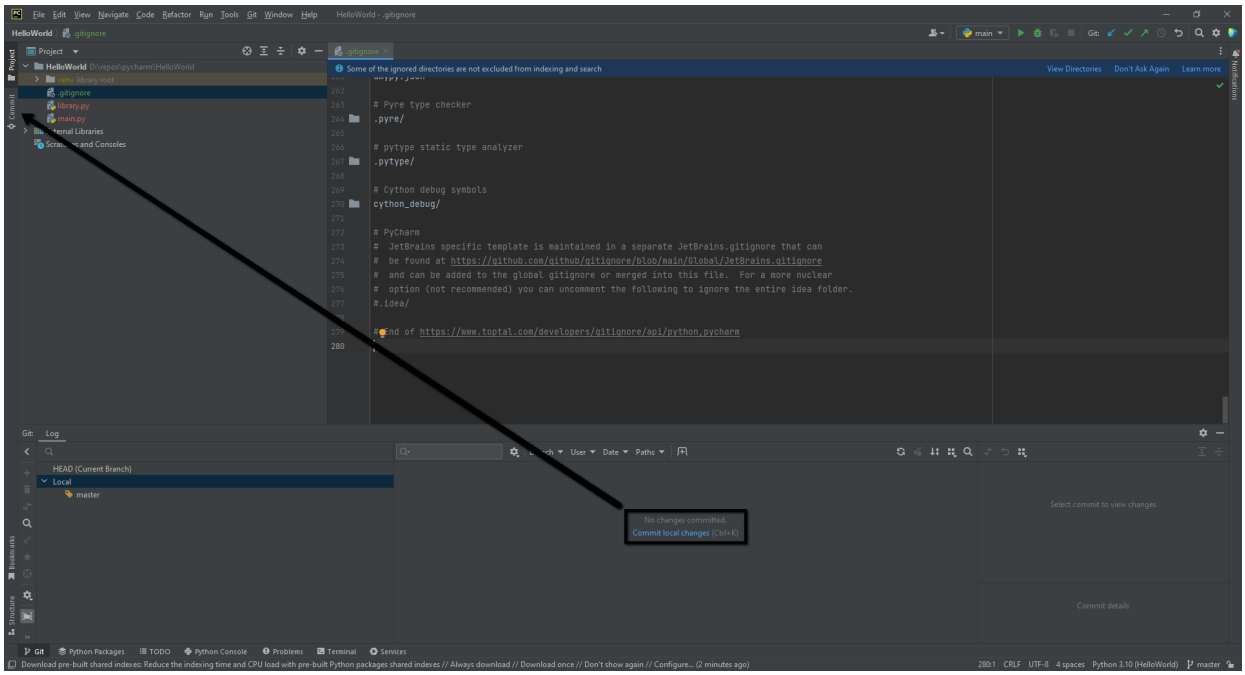
You write the name of the technologies that you want to generate the file for. I usually go with "Python", "PyCharm" and hit the "Create" button.

The website will display the content of your desired ".gitignore" file. Select and copy everything from there and go back to PyCharm.

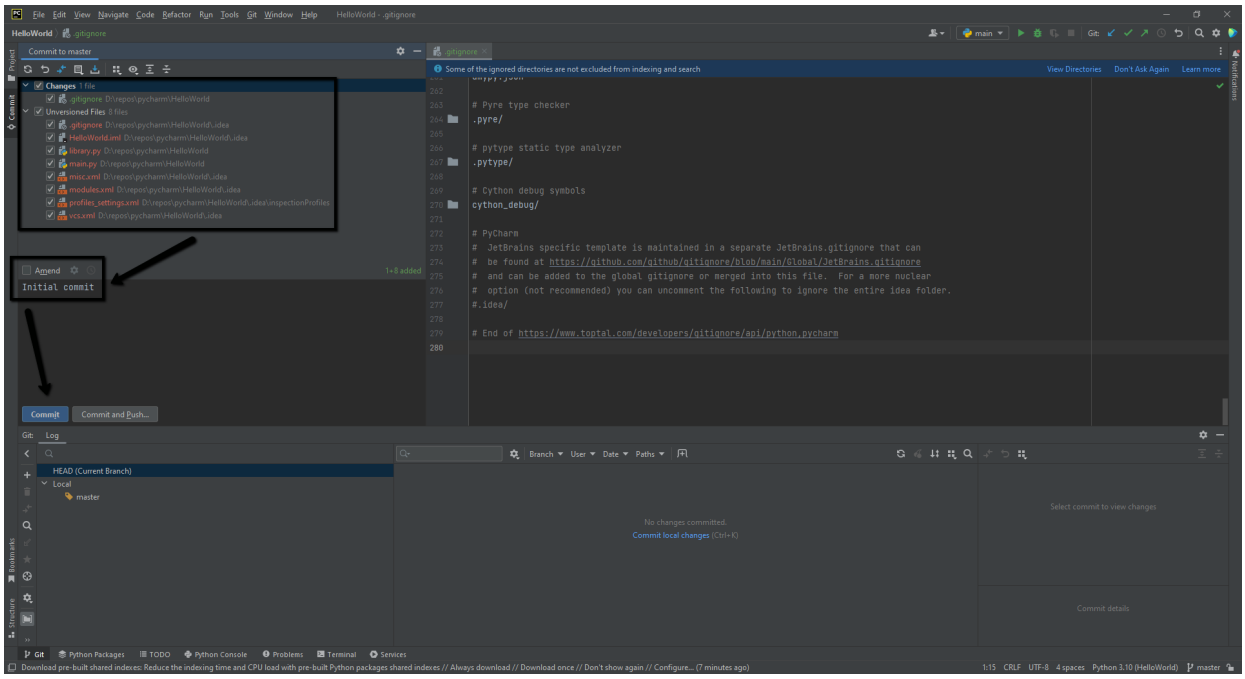
To simulate that, create a new file in your project by right clicking on the "HelloWorld" project name and selecting "File" under the "New" sub menu.



Name your file ".gitignore" and press enter. PyCharm will ask whether you want to add this file to Git or not. Click on Add and then paste the copied content.



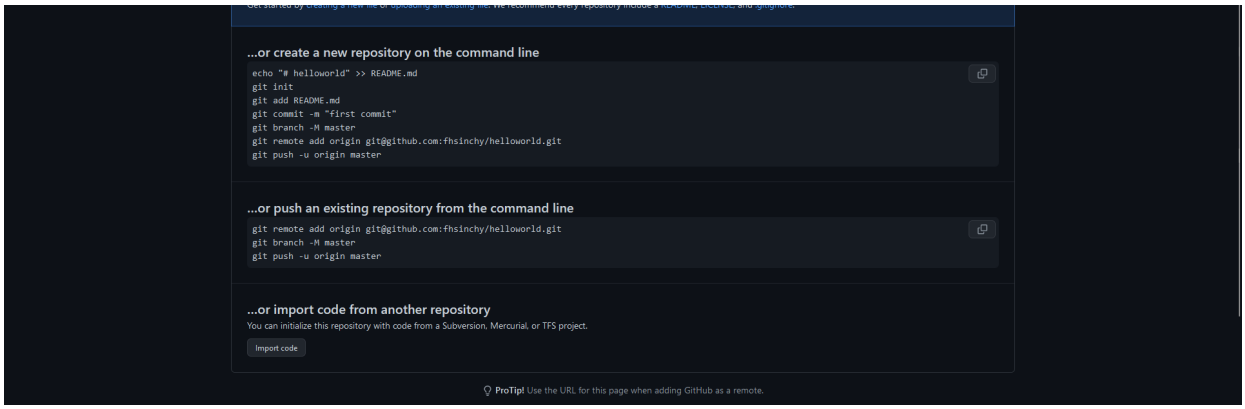
Since this is your first commit, check all "Changes" and "Unversioned Files" from the commit tab.



Since this is your first commit, put a descriptive commit message such as "Initial commit" and press the "Commit" button to finalize.

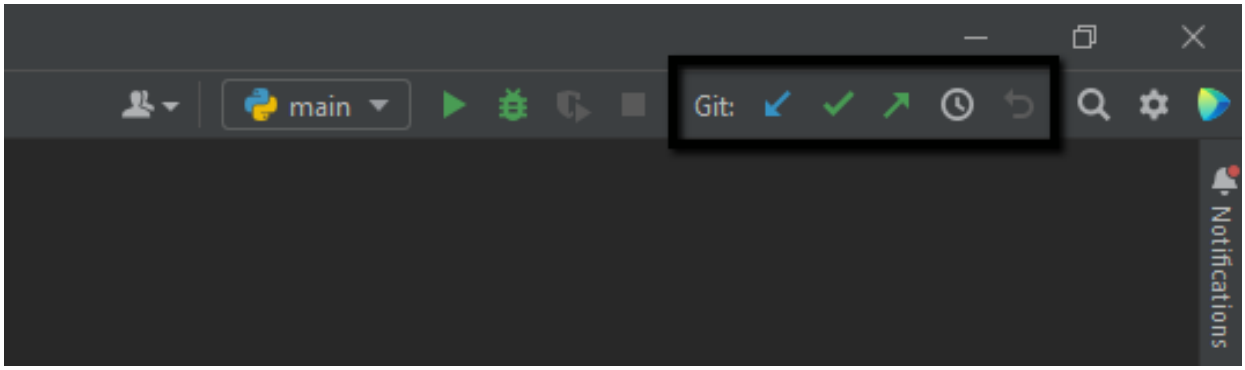
You've successfully made a commit to your local repository. You can now see all the commits under the master branch in detail.

Now it's time to publish this repository on GitHub. To do so, create a new repository under your GitHub account.



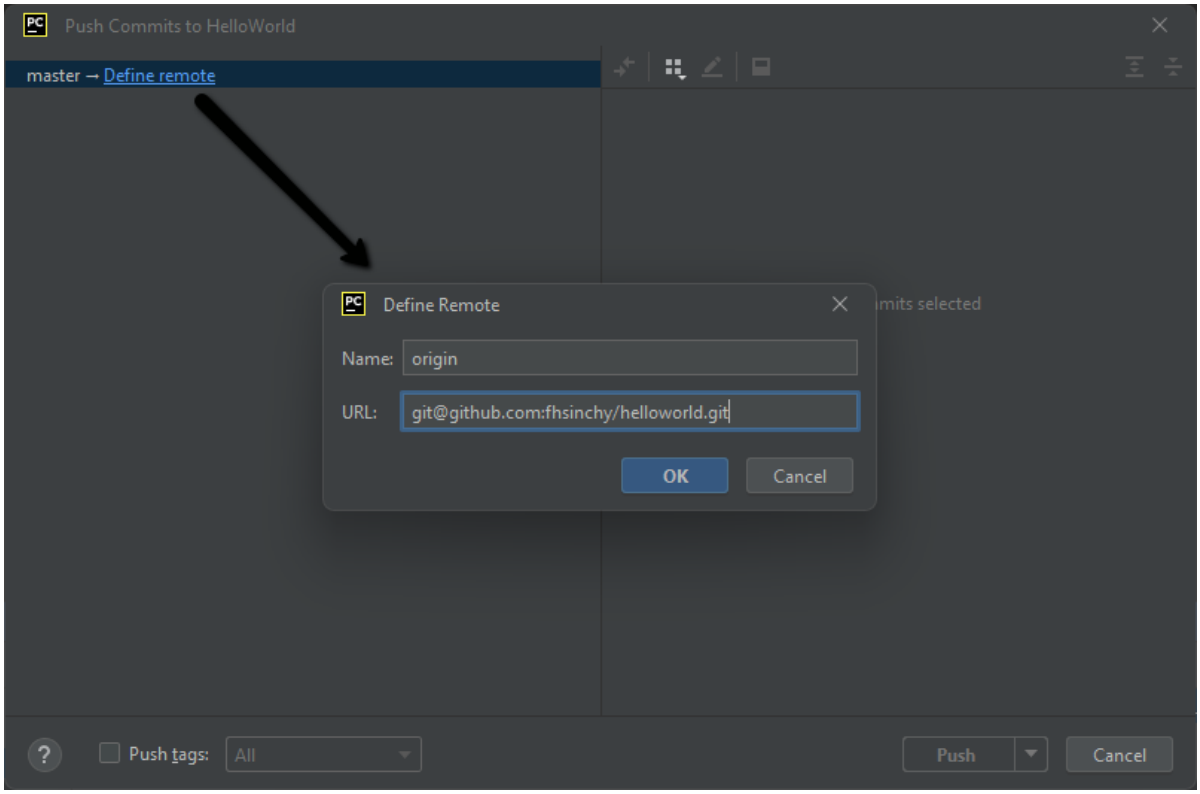
Then copy the SSH link to this repository. If you do not have SSH configured for your project, you may use the HTTPS link but I'd highly recommend SSH.

Now go back to PyCharm and look at the top right corner. Besides where it says Git, you'll find a few signs.

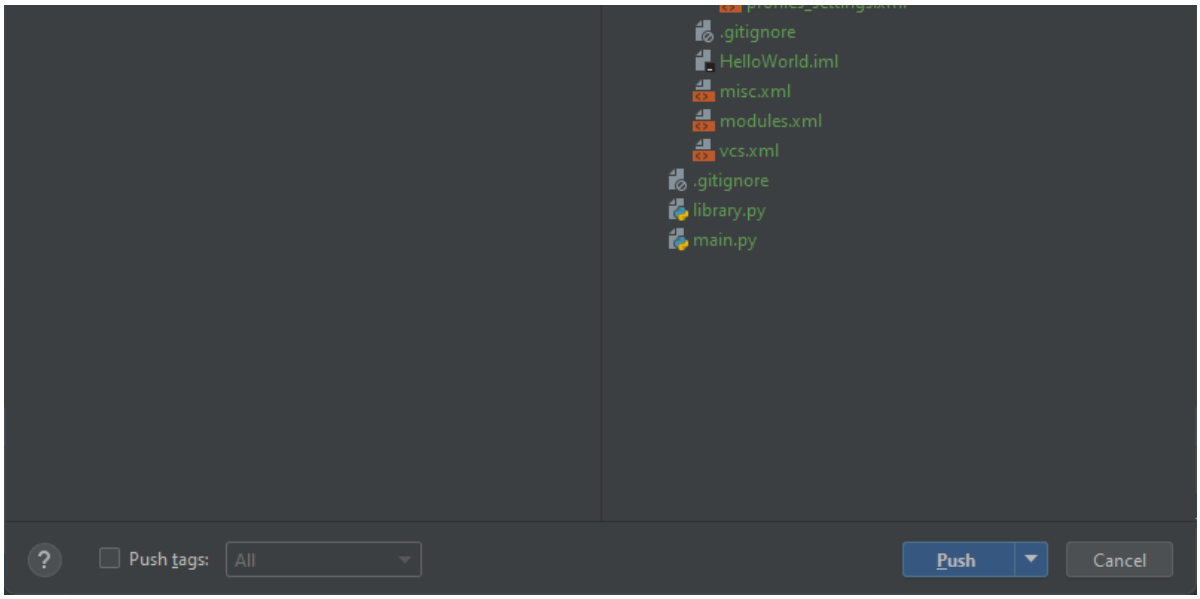


The downwards blue arrow will pull code from your remote repository, the tick sign will create a new commit, the upwards green arrow will push code.

The clock icon will show your commit history and finally the looped back arrow will revert your changes. Click on the push



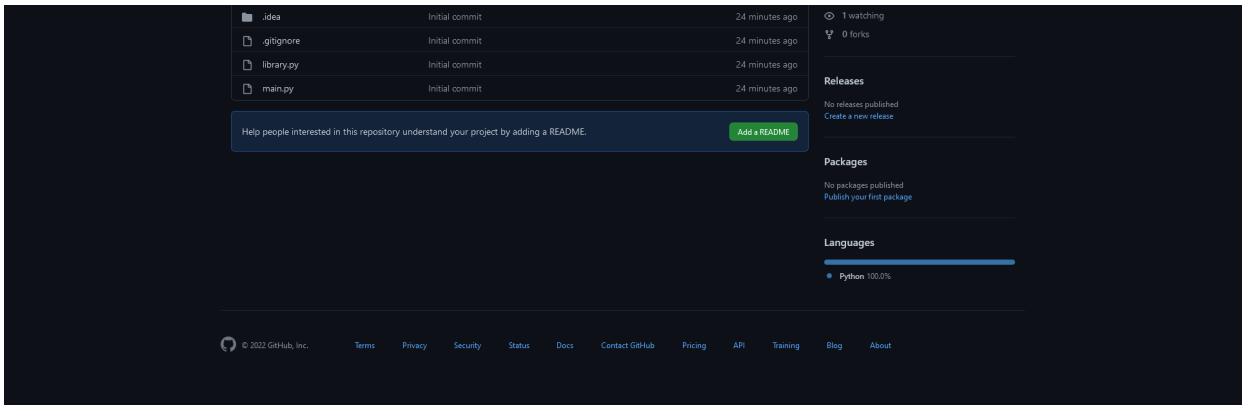
Click on the "Define remote" link and inside the URL input box, paste the link you've copied from GitHub. Press the OK button and wait until the process ends.



If everything goes fine, PyCharm will give you a "Push" button. It shouldn't take more than a few seconds to push the code to your remote repository.

If you're using HTTPS instead of SSH, you may have to provide your GitHub email and password on every push.

Once done, visit your remote repository and refresh the page to see if the changes have been pushed correctly or not.

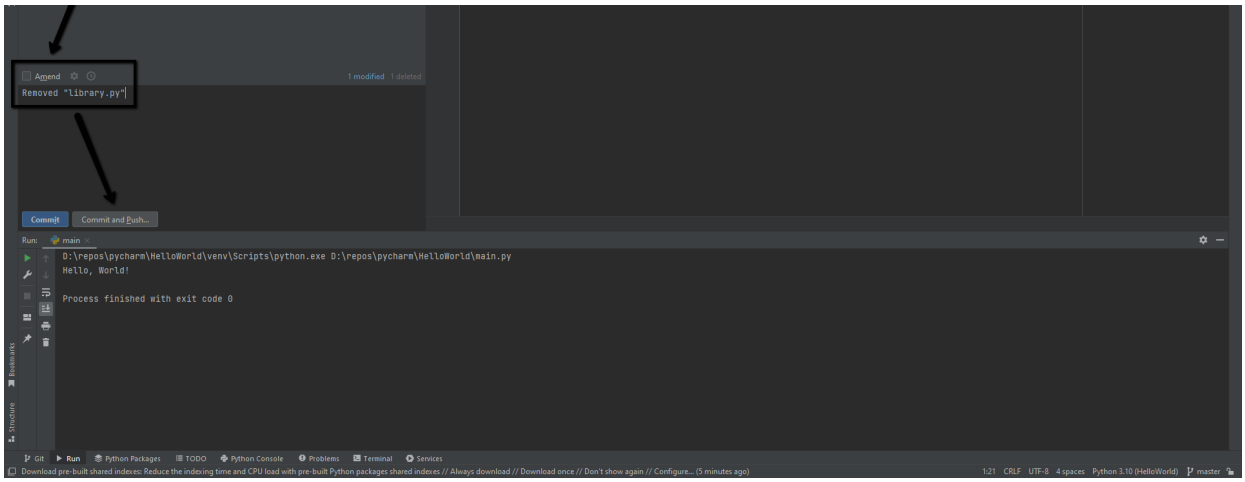


Now you can commit and push your code to GitHub right from your IDE every time you make any significant change.

For example, delete the "library.py" file and update the code inside the "main.py" file to print out "Hello, World!" on the console.

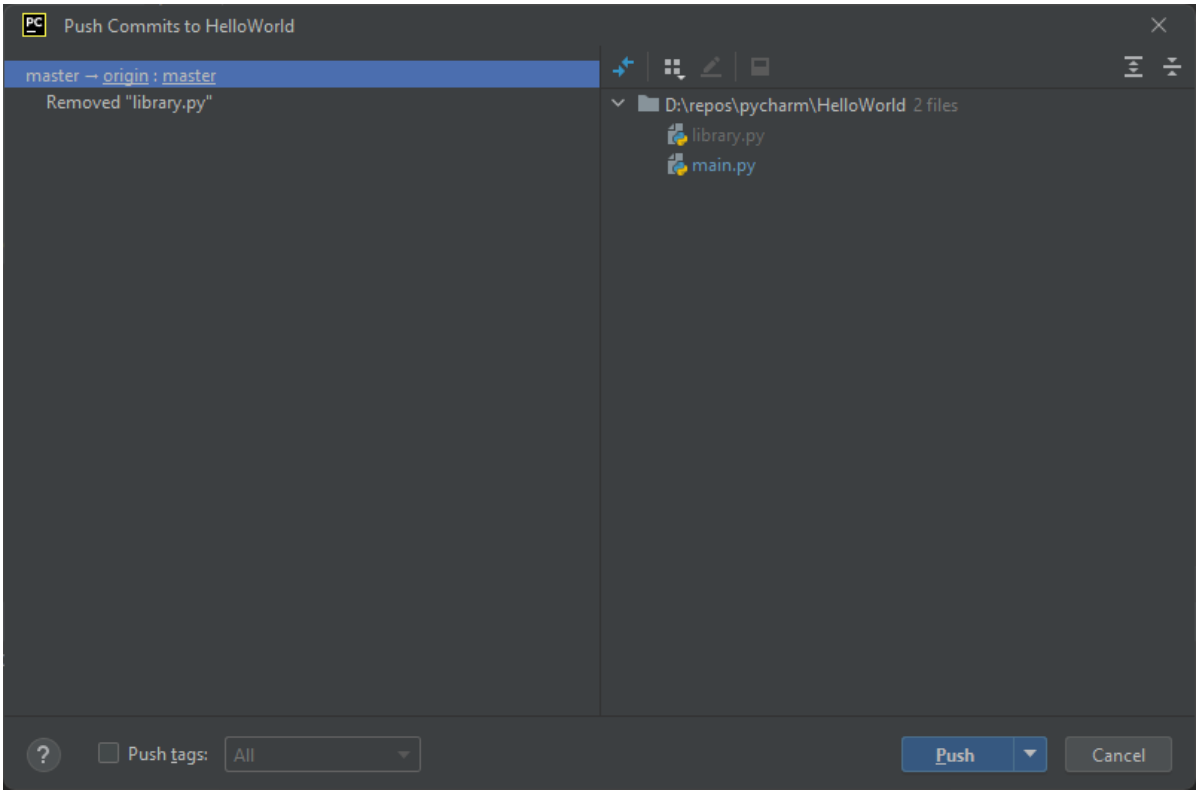
```
def main():
    print("Hello, World!")


if __name__ == '__main__':
    main()

# Hello, World!
```

Once you've made the changes, switch to the commit tab and you'll see all the uncommitted changes.



Make sure you've checked all the changes you want to commit. Write a descriptive commit message.

Then instead of just committing, try the "Commit and Push..." button this time. A new window will show up.



Remember, if you're using HTTPS you may have to reenter your email and password on every push.

You can check your remote repository on GitHub to make sure that the push has been done correctly.

You can do a lot more in terms of version controlling within PyCharm such as handling pull requests, but I'll leave those out for a later time.

How to Work With Variables and Different Types of Data in Python

A variable is an entity that can take on different values of different types. It's a named location in your computer's memory.

To create a new variable in Python, you just need to type out the name of the variable, followed by an equal sign and the value.

```
def main():
    book = 'Dracula'
    author = 'Bram Stoker'
    release_year = 1897
    goodreads_rating = 4.01

    print(book)
    print(author)
```

```
if __name__ == '__main__':
    main()

# Dracula
# Bram Stoker
# 1897
# 4.01
```

When it comes to naming your variable, the PEP 8 - Style Guide for Python says:

> Function names should be lowercase, with words separated by underscores as necessary to improve readability.

And

> Variable names follow the same convention as function names.

The guide also says:

> Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names. In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

As long as you're keeping these guidelines in mind, declaring variables in Python is very straightforward.

```
def main():
    book, author, release_year, goodreads_rating = 'Dracula', 'Bra

    print(book)
    print(author)
    print(release_year)
    print(goodreads_rating)


if __name__ == '__main__':
    main()

# Dracula
# Bram Stoker
# 1897
# 4.01
```

All you have to do is write the individual variable names in a single line using commas as separators.

Then after the equal sign you have to write the corresponding values in the same order as their names again using commas as separators.

In fact, you can also print them all out in one go. The `print()` method can take multiple parameters separated by commas.

```
def main():
    book, author, release_year, goodreads_rating = 'Dracula',
```

```
if __name__ == '__main__':
    main()

# Dracula Bram Stoker 1897 4.01
```

These parameters are then printed on the terminal in a single line using spaces for separating each of them.

Speaking of the `print()` method, you can use the `+` sign to add variables with strings inside a print method:

```
def main():
    book, author, release_year, goodreads_rating = 'Dracula', 'Bra

    print(book + ' is a novel by ' + author + ', published in ' +


if __name__ == '__main__':
    main()


# TypeError: can only concatenate str (not "int") to str
```

If you try to run this code you'll get a `TypeError` that says Python can concatenate or add together strings not integers.

In the code snippet above, `book`, `author`, `release_year`, and `goodreads_rating` are all variables of different types.

floating point number.

Whenever Python encounters a `+` sign in front of a numeric type, it assumes that the programmer may be performing an arithmetic operation.

The easiest way to solve this problem is to convert the numeric types to strings. You can do that by calling the `str()` method on the numeric variables.

```
def main():
    book, author, release_year, goodreads_rating = 'Dracula', 'Bra

    print(book + ' is a novel by ' + author + ', published in ' +


if __name__ == '__main__':
    main()

# Dracula is a novel by Bram Stoker, published in 1897. It has a r
```

That's better – but you can make that line of code even more readable by using a f string.

```
def main():
    book, author, release_year, goodreads_rating = 'Dracula', 'Bra

    print(f'{book} is a novel by {author}, published in {release_y
```

You can turn a regular string to a f string by putting a `f` in front of it and suddenly you can write variable names inside curly braces right within the string itself.

There is one last thing that's bugging me, and that's the length of the line of code itself. Fortunately you can split long strings into multiple shorter ones as follows:

```
def main():
    book, author, release_year, goodreads_rating = 'Dracula', 'Bra

    print(f'{book} is a novel by {author}, published in {release_y
          f' It has a rating of {goodreads_rating} on goodreads.')


if __name__ == '__main__':
    main()

# Dracula is a novel by Bram Stoker, published in 1897. It has a r
```

Now that's how a good piece of Python code should look. I'd suggest you try to make your code readable from the very beginning – you'll thank me later for that.

Other than `int` and `float`, there is another numeric type called `complex` in Python. It was specifically designed for

`False` and nothing else. You can actually ask Python questions and it'll answer in boolean.

Throughout this book you'll not see complex numbers in action and booleans will come into play much later. So for now, lets focus on simple numbers and strings.

How to Work With Simple Numbers in Python

Simple numbers in Python are of two types. Whole numbers are integers and numbers with floating points in them are floats.

In Python, you can represent integers using four different bases. These are decimal, hexadecimal, octal, and binary.

BASE	REPRESENTATION
Decimal	404
Hexadecimal	0x194
Octal	0o624
Binary	0b000110010100

So you can represent the value of 404 in hexadecimal, octal, or binary by prefixing the corresponding value with `0x` , `0o` , or `0b` respectively.

place may be inaccurate.

There are six different arithmetic operations that you can perform on any of the simple numeric types. The simplest of the bunch are addition and subtraction.

```
def main():
    num_1 = 15
    num_2 = 12

    print(f'sum of num_1 and num_2 is: {num_1 + num_2}')
    print(f'difference of num_1 and num_2 is: {num_1 - num_2}')

if __name__ == '__main__':
    main()

# sum of num_1 and num_2 is: 27
# difference of num_1 and num_2 is: 3
```

In case of a subtraction operation, the result will be negative if the second operand is larger than the first one.

```
def main():
    num_1 = 15
    num_2 = 12

    print(f'difference of num_2 and num_1 is: {num_2 - num_1}')

if __name__ == '__main__':
    main()
```

Similarly you can perform multiplication and division operations using their corresponding operators.

```
def main():
    num_1 = 15
    num_2 = 12

    print(f'product of num_1 and num_2 is: {num_1 * num_2}')
    print(f'quotient of num_1 and num_2 is: {num_1 / num_2}')
    print(f'floored quotient of num_1 and num_2 is: {num_1 // num_


if __name__ == '__main__':
    main()

# product of num_1 and num_2 is: 180
# quotient of num_1 and num_2 is: 1.25
# floored quotient of num_1 and num_2 is: 1
```

Keep in mind that you can not divide a number by zero in Python. If you attempt that, you'll get a `ZeroDivisionError` error (more on that later).

Output from a division operation will always be a float value, unless you perform a floored division by using two division operators.

```
def main():
    num_1 = 15
```

```
if __name__ == '__main__':
    main()

# floored quotient of num_1 and num_2 is: 1
```

In this case the result will be rounded off to the nearest integer low – so, for example, 0.25 will be lost. So only perform this operation when such loss of data is permissible.

The last operation to discuss is finding the remainder of a division operation.

```
def main():
    num_1 = 15
    num_2 = 12

    print(f'remainder of num_1 / num_2 is: {num_1 % num_2}')


if __name__ == '__main__':
    main()

# remainder of num_1 / num_2 is: 3
```

This operation is also called a modulo or modulus operation. So if someone mentions the modulo or modulus operator, they're referring to the percent sign.

```
def main():
    float_variable = 1.25
    integer_variable = 55

    print(f'{float_variable} converted to an integer is: {int(floa
    print(f'{integer_variable} converted to a float is: {float(int


if __name__ == '__main__':
    main()

# 1.25 converted to an integer is: 1
# 55 converted to a float is: 55.0
```

Loss of data in case of a float to integer conversion is inevitable, so be careful. You can use the `int()` and `float()` methods on strings as well (more on that later).

Any arithmetic operation involving a float operand will always produce a float result, unless converted to integer explicitly.

```
def main():
    float_variable = 5.0
    integer_variable = 55

    print(f'the sum of {float_variable} and {integer_variable} is:
    print(f'the sum of {float_variable} and {integer_variable} '
          f'converted to integer is: {int(float_variable + integer
```

```
# the sum of 5.0 and 55 converted to integer is: 60
```

If you ever want to get the absolute value of a signed value you can do so using the `abs()` method.

```
def main():
    num_1 = -5.8

    print(f'the absolute value of {num_1} is: {abs(num_1)}')


if __name__ == '__main__':
    main()

# the absolute value of -5.8 is: 5.8
```

There is a similar method `pow(x, y)` that you can use to apply `x` as the power of `y` like this.

```
def main():
    x = 2
    y = 3

    print(f'{2} to the power of {3} is: {pow(2, 3)}')
    print(f'{2} to the power of {3} is: {2 ** 3}')


if __name__ == '__main__':
    main()
```

You can perform the same operation using two multiplication operators but I always prefer the `pow()` method.

Finally there is the `divmod()` method that you can use to combine the division and modulo operation.

```
def main():
    num_1 = 8
    num_2 = 2

    print(f'division and modulus of {num_1} and {num_2} is: {divmo


if __name__ == '__main__':
    main()

# division and modulus of 8 and 2 is: (4, 0)
```

The method returns a tuple of numbers (more on that later). The first one is the result of the division and the second one is the result of the modulo operation.

These are the basic operations you can perform on simple numbers right from the get go. But you can do much more once you start to pull in the built-in modules.

Learning how to take input from a user is an important milestone because it lets you create programs that a human being can interact with.

Unlike many other programming languages, taking user inputs in Python is very straightforward.

```
def main():
    name = input('What is your name? ')

    print(f'Nice to meet you {name}')


if __name__ == '__main__':
    main()

# What is your name? Farhan
# Nice to meet you Farhan
```

The built-in `input()` method does exactly what it sounds like. The method accepts a single parameter `prompt` which is of string type.

Whatever you write as the value of this parameter will be shown in the console – like in this case, "What is your name?" is the prompt.

Once the user writes something on the console and presses enter, the input method will return that as a string.

input() will return that as a string.

```
def main():
    name = input('What is your name? ')
    age = input(f'How old are you {name}? ')
    current_year = input(f'What year is this again? ')

    print(f'If my calculations are right, you were born in {curren


if __name__ == '__main__':
    main()

# What is your name? Farhan
# How old are you Farhan? 27
# What year is this again? 2023
# TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Even though Python is taking all the user inputs correctly, it fails to calculate the user's birth year because arithmetic operations are not a good fit for strings.

To solve this problem, you just have to convert the user inputs to numeric types using the `int()` or `float()` functions as needed.

```
def main():
    name = input('What is your name? ')
    age = int(input(f'How old are you {name}? '))
    current_year = int(input(f'What year is this again? '))

    print(f'If my calculations are right, you were born in {curren
```

```
# What is your name? Farhan
# How old are you Farhan? 27
# What year is this again? 2023
# If my calculations are right, you were born in 1996
```

There you go, works like a charm. You can perform this conversion at any point in the code. It's not mandatory to convert them right at the beginning.

```
def main():
    temperature_in_celsius = input('What is the temperature in cel

    temperature_in_fahrenheit = (float(temperature_in_celsius) * 1

    print(f'{temperature_in_celsius} degree celsius is equivalent


if __name__ == '__main__':
    main()

# What is the temperature in celsius? 32
# 32 degree celsius is equivalent to 89.6 degree fahrenheit.
```

This program can convert temperature from Celsius to Fahrenheit. In this program, I didn't convert the input from string to a numeric type right away.

I performed the conversion during the calculation leaving the original input variable intact. Also notice the use of `float()`

How to Work With Strings in Python

You've already seen examples of strings in the previous sections – but there is a lot more that you need to learn about strings.

In Python, anything enclosed within a set of single, double, or triple quotes is a string. These are sequences of bytes representing Unicode characters.

```
def main():
    book = 'Dracula'
    author = "Bram Stoker"

    print('Title:', book)
    print('Author:', author)


if __name__ == '__main__':
    main()

# Title: Dracula
# Author: Bram Stoker
```

Declaring a string with single or double quotes makes no difference whatsoever. But based on the scenario, you may have to choose on over the other.

For example, if you have an apostrophe within your sentence, you may want to use double quotes.

```
    print(question)


if __name__ == '__main__':
    main()

# What's your name?
```

The opposite can also occur. For example, when you have a direct quote within your string:

```
def main():
    sentence = 'Harriet Jacobs writes, "She sat down, quivering in

    print(sentence)


if __name__ == '__main__':
    main()

# Harriet Jacobs writes, "She sat down, quivering in every limb"
```

You can also go for escape sequences if you want to, but the PEP 8 - Style Guide for Python Code recommends avoiding the usage of back slashes within strings.

Triple quotes are a different case altogether. You can put multi-line strings within triple quotes and Python will preserve the white spaces as well.

```
    synopsis = """Dracula comprises journal entries, letters, and
It begins with Jonathan Harker, a young English lawyer, as he trav
Harker plans to meet with Count Dracula, a client of his firm, in
When he arrives in Transylvania, the locals react with terror afte
Though this unsettles him slightly, he continues onward.
The ominous howling of wolves rings through the air as he arrives
When Harker meets Dracula, he acknowledges that the man is pale, g
Harker becomes further concerned when, after Harker cuts himself w
Soon after, Harker is seduced by three female vampires, from whom
He then learns Dracula's secret—that he is a vampire and survives
Harker correctly assumes that he is to be the count's next victim.
He attacks the count, but his efforts are unsuccessful.
Dracula leaves Harker trapped in the castle and then, along with 5

    print('Synopsis:', synopsis)


if __name__ == '__main__':
    main()

# Synopsis: Dracula comprises journal entries, letters, and telegr
# It begins with Jonathan Harker, a young English lawyer, as he tr
# Harker plans to meet with Count Dracula, a client of his firm, i
# When he arrives in Transylvania, the locals react with terror af
# Though this unsettles him slightly, he continues onward.
# The ominous howling of wolves rings through the air as he arrive
# When Harker meets Dracula, he acknowledges that the man is pale,
# Harker becomes further concerned when, after Harker cuts himself
# Soon after, Harker is seduced by three female vampires, from who
# He then learns Dracula's secret—that he is a vampire and survive
# Harker correctly assumes that he is to be the count's next victi
# He attacks the count, but his efforts are unsuccessful.
# Dracula leaves Harker trapped in the castle and then, along with
```

So if you ever want to print out a multi line string while preserving the white spaces, go for triple quotes.

usage of all the double quotes.

There is a lot more to learn about strings, but I'd like to introduce you to some other sequence types in Python.

What Are the Sequence Types in Python?

In Python, there are three sequence types. They are lists, tuples, and ranges. I'll start with the lists because it's probably the most utilized sequence type in Python.

Lists in Python

A list in Python is exactly what it sounds like: a collection of data stored sequentially on the computer's memory.

You can create a new list in Python by writing out its name followed by an equal sign, followed by the values to store enclosed in square brackets:

```
def main():
    horror_books = ['Dracula', 'Carmilla', 'The Imago Sequence']

    print(horror_books)


if __name__ == '__main__':
    main()
```

In this example, `horror_books` is a list of strings. But you can create lists of integers, floats, or even of mixed types.

```
def main():
    a_random_list = ['Dracula', 1, 5.7, 'Carmilla']

    print(a_random_list)


if __name__ == '__main__':
    main()

# ['Dracula', 1, 5.7, 'Carmilla']
```

Though this is perfectly valid, you may find yourself creating lists of the same types more often.

Lists in Python are mutable. This means you can modify a list after its creation. For example, you can use the `pop()` method to get rid of the last value in a list.

```
def main():
    horror_books = ['Dracula', 'Carmilla', 'The Imago Sequence']

    print(horror_books.pop())
    print(horror_books)


if __name__ == '__main__':
```

As you can see, the `pop()` method returns the last value from the list and gets rid of it. Like `pop()` there is the `append()` method for inserting new item to the list.

```
def main():
    horror_books = ['Dracula', 'Carmilla', 'The Imago Sequence']

    print(horror_books)

    horror_books.append('The Exorcist')

    print(horror_books)


if __name__ == '__main__':
    main()

# ['Dracula', 'Carmilla', 'The Imago Sequence']
# ['Dracula', 'Carmilla', 'The Imago Sequence', 'The Exorcist']
```

As you can see from the method name, it adds the new item at the end of the list. Given their mutable nature, lists can also be sorted.

Feel free to check out the following article written by my colleague Dionysia Lemonaki here on freeCodeCamp about how to sort lists in Python:

Tuples in Python

Lists are not the only sequence type in Python. The closest sibling of lists in Python are tuples.

You can create a new tuple in Python by writing out its name followed by an equal sign, then enclosing inside a pair of parenthesis the values you want to store.

```
def main():
    horror_books = ('Dracula', 'Carmilla', 'The Imago Sequence')

    print(horror_books)


if __name__ == '__main__':
    main()

# ('Dracula', 'Carmilla', 'The Imago Sequence')
```

Just like lists, you can also mix and match different types of data within a single tuple as you see fit.

```
    print(a_random_list)


if __name__ == '__main__':
    main()

# ('Dracula', 1, 5.7, 'Carmilla')
```

The most glaring dissimilarity between a list and a tuple is the fact that a tuple is immutable. So there's no popping and appending for us this time.

Ranges in Python

The final sequence type that you're going to learn about in this section is a range. A range in Python is just a range of numbers.

You can create a range by calling the `range()` method and it'll return a range of numbers. You can call the method in a few different ways.

The most common is by passing a single number as a parameter. In this case, the method will treat that number as the end of the range and 0 as the start.

```
def main():
    a_range = range(10)

    print(a_range)
```

```
if __name__ == '__main__':
    main()

# range(0, 10)
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Printing out a range as is won't give you much information. You'll have to convert the range to a list or a tuple by either calling the `list()` or `tuple()` method.

Once converted, you can then print out the entire range to the console. Notice how 10 or the number passed to the `range()` method is not included in the range.

The second way of calling the method is by supplying both the starting and ending numbers for the range.

```
def main():
    a_range = range(5, 15)

    print(a_range)

    list_a_range = list(a_range)

    print(list_a_range)

    tuple_a_range = tuple(a_range)

    print(tuple_a_range)
```

```
# [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
# (5, 6, 7, 8, 9, 10, 11, 12, 13, 14)
```

Once again, the number you pass as the ending for the range will not be included in the resultant range.

The third and final way to call the method is by also defining a step. For example, imagine you want a range comprising of all the odd numbers within 1 to 10.

```
def main():
    a_range = range(1, 10, 2)

    print(a_range)

    list_a_range = list(a_range)

    print(list_a_range)

    tuple_a_range = tuple(a_range)

    print(tuple_a_range)


if __name__ == '__main__':
    main()

# range(1, 10, 2)
# [1, 3, 5, 7, 9]
# (1, 3, 5, 7, 9)
```

It may take some time to wrap your head around this concept but practicing with different step values will help.

Or you can read the following article written by Bala Priya C:

How Indexing Works in Python

One of the most important concepts regarding sequence types that you need to understand is indexing.

You see, each element in a sequence has a number attached to it that expresses its position in the list called an index. These indices are 0 based.

horror_books = ['Dracula', 'Carmilla', 'The Imago Sequence']		
0	1	2
Dracula	Carmilla	The Imago Sequence

place.

The second one is at the 1st place and the third one is at the 2nd place. This zero-based indexing is may seem confusing at first but you'll get the hang of it.

The most basic usage of a index is to access its corresponding value from the sequence.

```
def main():
    horror_books = ['Dracula', 'Carmilla', 'The Imago Sequence']

    print(horror_books[0])
    print(horror_books[1])
    print(horror_books[2])


if __name__ == '__main__':
    main()

# Dracula
# Carmilla
# The Imago Sequence
```

You can also use negative numbers as indices but in that case the counting will start from the end.

```
def main():
    books = ['Dracula', 'Frankenstein', 'The Omen', 'The Exorcist'
             'And Then There Were None', 'The ABC Murders', 'The V
```

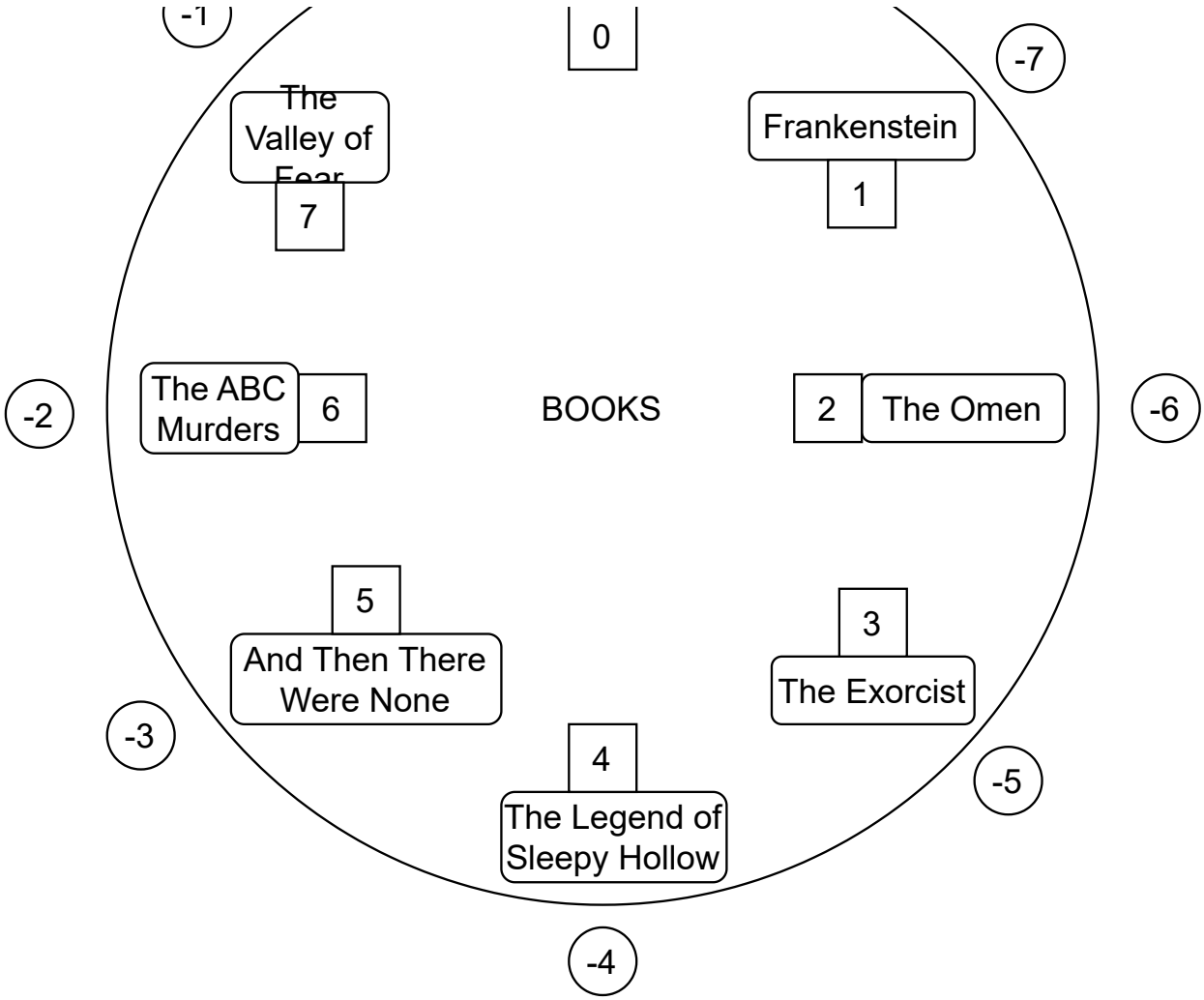
```
    print(books[-1])

    print(books[2])
    print(books[-2])


if __name__ == '__main__':
    main()

# Dracula

# Frankenstein
# The Valley of Fear

# The Omen
# The ABC Murders
```

The 0th element in a list will always be the first one. Now if you access the element on the 1st position you get "Frankenstein".

But if you try to access the element on the -1st position, you get "The Valley of Fear" because that's the second item in reverse.

The element on the 2nd position is "The Omen" but the element at the -2nd position is "The ABC Murders" because that's the third item in reverse.

If you're finding it hard to wrap your head around, imagine the list like a clock.



Zero-based indexing represented as a circular diagram like a clock

Here the outer number is the negative index and the inner number is the positive index. If you try to match the outputs against this imaginary clock, it should be easier to understand.

What Are the Iterable Types and How to Use them for Loops in Python

So far you've learned about creating collections of data and accessing them one by one. That's cool but there is something

bunch of numbers.

Now you want to multiply each number in that list by two, insert the multiplied numbers in a new list, and print out the list on the terminal.

This is an excellent use case for the `for` statement in Python. Let's begin by first iterating through each number in a given list.

```
def main():
    random_numbers = [6, 1, 3, 8, 0, 9, 12, 3, 4, 0, 54, 8, 100, 5

    for number in random_numbers:
        print(number)

if __name__ == '__main__':
    main()

# 6
# 1
# 3
# 8
# 0
# 9
# 12
# 3
# 4
# 0
# 54
# 8
# 100
# 55
# 60
```

You start by writing out the word `for` followed by a variable name. I've used `number` but you can use anything that makes sense to you.

Although you write it as `for number`, Python reads it as `for each number` and wonders where are these numbers staying?

That's when you say `in` followed by the name of the sequence, `random_numbers` in this case.

Now Python understands that you want to do something with each number in the `random_numbers` sequence, but what?

That's what you have to write out after the colon and be very careful about the indentation. Anything indented one level after the for loop declaration is considered the loop body.

Inside the for loop you can write whatever you want to do with the current value of the `number` variable.

Since there are 17 numbers in the sequence, the loop will run 17 times and each time it'll have a new value.

It'll start at index 0 which has the value of 6 and go through index 1, 2, 3, 4, 5, and so on.

Instead of printing out the original value, you can multiply it by 2 and print out the resultant value instead.

```
def main():
    random_numbers = [6, 1, 3, 8, 0, 9, 12, 3, 4, 0, 54, 8, 100, 5

    for number in random_numbers:
        print(number * 2)

if __name__ == '__main__':
    main()

# 12
# 2
# 6
# 16
# 0
# 18
# 24
# 6
# 8
# 0
# 108
# 16
# 200
# 110
# 120
# 140
# 170
```

Now you're getting the multiplied values. The final task is to insert these multiplied values in a new list and print out the new

```
def main():
    random_numbers = [6, 1, 3, 8, 0, 9, 12, 3, 4, 0, 54, 8, 100, 5
    multiplied_random_numbers = []

    for number in random_numbers:
        multiplied_random_numbers.append(number * 2)

    print(multiplied_random_numbers)

if __name__ == '__main__':
    main()

# [12, 2, 6, 16, 0, 18, 24, 6, 8, 0, 108, 16, 200, 110, 120, 140,
```

For that you'll need an empty list. Then, after multiplying the number, you can simply call the `append()` method on the new list and pass the multiplied value.

Finally, make sure that you're putting the print statement outside of the loop body otherwise you'll end up printing out the list 17 times.

The `for` loop works with all the sequence types and any iterable type in the Python language. What is an iterable type, I hear you ask.

Well, any object that has the `__iter__()` method is considered an iterable in Python.

```
def main():
    random_numbers = [6, 1, 3, 8, 0, 9, 12, 3, 4, 0, 54, 8, 100, 5

    print(dir(random_numbers))

if __name__ == '__main__':
    main()

# ['__add__', '__class__', '__class_getitem__', '__contains__', '_
```

You can see some familiar methods such as `append`, `count`, and `index` but most importantly it has the `__iter__` method.

As you keep working in Python you'll eventually remember the types supported by the `for` loop but you can always use the `dir()` method on a object to find out.

How to Use While Loops in Python

There is another type of loop in Python known as the `while` loop. Unlike `for`, a `while` loop can execute a statement as long as a given condition evaluates to `true`.

```
def main():
    number = 1
```

```
if __name__ == '__main__':
    main()

# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
# 10
```

Here you have a variable `number` with the value `11` and a `while` loop that prints out the value of number, then increases it by 1.

A `while` loop starts by writing out `while` followed by the condition. Then you write the loop body starting from the next line after the colon.

`for` loops are useful when you're trying to access every element inside an iterable. `while` loops are useful when you want to repeat the same set of instructions an arbitrary number of times.

The line `number += 1` is another way to write `number = number + 1` and it's very commonly used by programmers across different programming languages.

You can also nest one loop inside another. For example, look at the following code that prints out multiplication tables:

```
def main():
    for x in range(1, 6):
        print()
        for y in range(1, 11):
            print(f"{x} x {y} = {x * y}")


if __name__ == '__main__':
    main()

#
# 1 x 1 = 1
# 1 x 2 = 2
# 1 x 3 = 3
# 1 x 4 = 4
# 1 x 5 = 5
# 1 x 6 = 6
# 1 x 7 = 7
# 1 x 8 = 8
# 1 x 9 = 9
# 1 x 10 = 10
#
# 2 x 1 = 2
# 2 x 2 = 4
# 2 x 3 = 6
# 2 x 4 = 8
# 2 x 5 = 10
# 2 x 6 = 12
# 2 x 7 = 14
# 2 x 8 = 16
# 2 x 9 = 18
# 2 x 10 = 20
#
# 3 x 1 = 3
# 3 x 2 = 6
```

```
# 3 x 7 = 21
# 3 x 8 = 24
# 3 x 9 = 27
# 3 x 10 = 30
#
# 4 x 1 = 4
# 4 x 2 = 8
# 4 x 3 = 12
# 4 x 4 = 16
# 4 x 5 = 20
# 4 x 6 = 24
# 4 x 7 = 28
# 4 x 8 = 32
# 4 x 9 = 36
# 4 x 10 = 40
#
# 5 x 1 = 5
# 5 x 2 = 10
# 5 x 3 = 15
# 5 x 4 = 20
# 5 x 5 = 25
# 5 x 6 = 30
# 5 x 7 = 35
# 5 x 8 = 40
# 5 x 9 = 45
# 5 x 10 = 50
```

To be honest, this is a very simple bit of code that makes use of a lot of the things you've already learned in this handbook.

To create a multiplication table we need two operands: one remains constant for the entire table and the other increases by 1 until it reaches 10.

The first loop iterates through a range of 1 to 5 and the second loop iterates through a range of 1 to 10.

Since the ending number of a range is exclusive, you need to put a number that is 1 higher than the desired ending number.

First the Python interpreter encounters the outer loop and starts executing it. While inside that loop, the value of `x` is 1.

The interpreter then encounters the inner loop and starts executing that. While inside the inner loop, the value of `x` remains 1 but the value of `y` increases in each iteration.

The inner loop is the body of the outer loop in this case, so the first iteration of the outer loop lasts until the inner loop finishes.

After finishing 10 iterations of the inner loop, the interpreter comes back to the outer loop and starts executing it once again.

This time the value of `x` becomes 2 since that's what comes next in the range.

Just like that, the outer loop executes 5 times and the inner loop executes 10 times for each of those iterations.

Like a lot of other concepts, wrapping your head around nested loops can be difficult, but practice will make things easier.

You can also take the two numbers from the user and print the multiplication table within that range.

For example, if the user puts 5 and 10 as inputs, then you'll print out the multiplication tables of all the numbers from 5 to 10.

You can nest loops to even deeper levels, but going deeper than two loops may cause performance issues so be careful with that.

What Are Some Common Sequence Type Operations in Python?

Assuming you remember the text sequence type (strings), you're now familiar with the four most popular Python sequence types.

So I think it's time for you to learn some common operations that you can perform on them. Let's begin, shall we?

How to Use the `in` Operator in Python

The `in` operator is the most common way of checking for any object's existence. For example, assume that you have a string and you want to check if it contains the word "red" or not.

```
def main():
    a_string = 'Little Red Riding-Hood comes to me one Christmas E
```

```
if __name__ == '__main__':
    main()

# True
```

It's literally like asking Python, if the word `Red` is `in` the `a_string` variable. And Python will give you either `True` or `False` as an answer.

The `in` operator is not exclusive to strings. You can actually use it on any other collection type such as lists, tuples, and ranges.

```
def main():
    books = ['Dracula', 'Frankenstein', 'The Omen', 'The Exorcist'
    movies = ('A Christmas Carol', 'The Sea Beast', 'Enchanted', '
    numbers = range(10)

    print('A Christmas Carol' in books)
    print('Enchanted' in movies)
    print(5 in numbers)


if __name__ == '__main__':
    main()

# False
# True
# True
```

You may also want to find out about the absence of an object. For that, you can use the `not` operator in conjunction with the `in` operator.

```
def main():
    books = ['Dracula', 'Frankenstein', 'The Omen', 'The Exorcist'
    movies = ('A Christmas Carol', 'The Sea Beast', 'Enchanted', '
    numbers = range(10)

    print('A Christmas Carol' not in books)
    print('Enchanted' not in movies)
    print(15 not in numbers)


if __name__ == '__main__':
    main()

# True
# False
# True
```

A Christmas Carol doesn't exist in the `books` list, so the first statement evaluates to `true`. The second one evaluates to false because Enchanted is present in the `movies` list.

The last one is self explanatory at this point. The `in` and `not in` operators come in very handy when working with conditional statements.

You've already learned about `+` and `*` as arithmetic operators – but in the case of sequence types, they play a very different role.

The `+` operator lets you merge two sequences together.

```
def main():
    books = ['Dracula', 'Frankenstein', 'The Omen', 'The Exorcist'
    more_books = ['And Then There Were None', 'The ABC Murders', '

    print(books + more_books)

if __name__ == '__main__':
    main()

# ['Dracula', 'Frankenstein', 'The Omen', 'The Exorcist', 'The Leg
```

As you can see, the operator has appended the content of the `books` list to the content of the `more_books` list.

The `*` operator, on the other hand, makes multiple copies of a given sequence.

```
def main():
    books = ['Dracula', 'Frankenstein', 'The Omen', 'The Exorcist'

    print(books * 2)
```

```
# ['Dracula', 'Frankenstein', 'The Omen', 'The Exorcist', 'The Leg
```

So multiplying the `books` list by 2 gives us all the 5 books in the list twice. These operators work the same for tuples, strings, ranges or any other sequence types.

How to Use the `len()`, `min()`, and `max()` Functions in Python

The `len()` function can return the length of a given sequence. And the `min()` and `max()` functions can return the minimum and maximum value in a given sequence, respectively.

```
def main():
    random_numbers = [6, 1, 3, 8, 0]


    print(len(random_numbers))
    print(min(random_numbers))
    print(max(random_numbers))


if __name__ == '__main__':
    main()

# 5
# 0
# 8
```

The smallest value in the list is 0 and the largest value is 8 which are the outputs from the `min()` and `max()` function calls, respectively.

Depending on the type of programs you end up writing in the future, these three functions can prove to be some of the most useful ones.

What Are Some String Type Operations in Python?

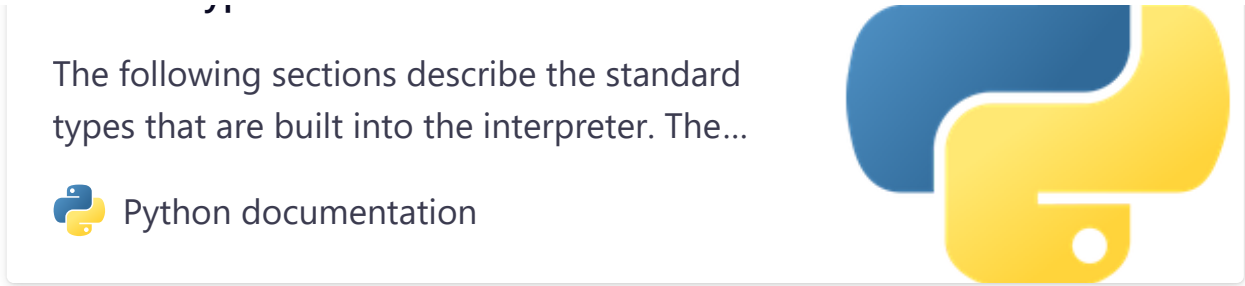
In the previous section, you've learned about some common operations that you can perform on any sequence type including strings.

However, the text sequence type aka strings have some special operations available to them.

In this chapter I'll introduce you to some of the most common string methods. Keep in mind that this is not a definitive list.

Although each of the methods I'm going to teach you performs a different task, they have one thing in common. None of them modifies a given string variable in place, but rather returns a new, modified copy.

If you want to learn about all the available string methods, feel free to consult the official Python documentation.



Also remember it's not a matter of just going through each method and memorizing their usage.

It's about knowing what works best in a given scenario and coming up with clever solutions. And that requires practice.

How to Capitalize Strings in Python

The first method you're going to learn is called `capitalize()` and it does what it sounds like.

```
def main():
    country_name = 'bangladesh'

    print(country_name.capitalize())


if __name__ == '__main__':
    main()

# Bangladesh
```

As you can see from the code snippet above, the `capitalize()` method turns the first letter of the word to a capital letter.

```
def main():
    book_name = 'the house of silk'

    print(book_name.capitalize())


if __name__ == '__main__':
    main()

# The house of silk
```

Although the method did its job, there is slight problem. Depending on what you're trying to achieve, you may expect the first letter of each word to be capitalized.

That's where the `title()` method comes in. This method returns a title cased version of a given string.

```
def main():
    book_name = 'the house of silk'

    print(book_name.title())


if __name__ == '__main__':
    main()

# The House Of Silk
```

```
def main():
    book_name = "alice's adventures in wonderland"

    print(book_name.title())


if __name__ == '__main__':
    main()

# Alice'S Adventures In Wonderland
```

As you can see, the `title()` method treats the `s` following the apostrophe as a separate word and capitalizes it.

Regarding this issue, the official documentation states:

> The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries.

The `capwords()` helper function can solve this issue. This function breaks the string into multiple words based on the spaces between them, capitalizes the words, joins them back into a string and returns that to the user.

```
def main():
    book_name = "alice's adventures in wonderland"

    print(capwords(book_name))


if __name__ == '__main__':
    main()

# Alice's Adventures In Wonderland
```

Pay attention to the `import` statement at the top. The `capwords()` function is not a method within the string type but a function that resides inside the `string` module.

You'll learn about modules and imports in more details later on. For now, just roll with it. Although the function uses spaces to split words, you can overwrite it.

```
from string import capwords


def main():
    address = 'house 42, road 02, wonderland'

    print(capwords(address, ', '))


if __name__ == '__main__':
    main()

# House 42, Road 02, Wonderland
```

The `capwords()` function can take a custom delimiter as its second parameter. You can pass any string as the delimiter.

Finally, there is the `istitle()` method that can check whether a given string is in title case or not.

```
def main():
    book_name = 'hearts in atlantis'

    print(f'Is "{book_name}" in title case? {book_name.istitle()}'
    print(f'Is "{book_name.title()}" in title case? {book_name.tit


if __name__ == '__main__':
    main()

# Is "hearts in atlantis" in title case? False
# Is "Hearts In Atlantis" in title case? True
```

However, keep in mind that the `istitle()` method doesn't work with the `capwords()` helper function.

How to Convert Strings to Lower Case or Upper Case in Python

Apart from capitalization, you may want to convert an entire string to upper case or lower case. You can do that by using the `upper()` and `lower()` methods in Python.

```
    print(book_name.upper())

    another_book_name = 'DRACULA'

    print(another_book_name.lower())


if __name__ == '__main__':
    main()

# MORIARTY
# dracula
```

There are also the `isupper()` and `islower()` methods to check whether a given string is already in either of the letter cases or not.

```
def main():
    book_name = 'moriarty'

    print(book_name)
    print(f'Is {book_name} in upper case? {book_name.isupper()}')
    print(f'Is {book_name} in lower case? {book_name.islower()}')

    another_book_name = 'DRACULA'

    print(another_book_name)
    print(f'Is {another_book_name} in upper case? {another_book_na
    print(f'Is {another_book_name} in lower case? {another_book_na


if __name__ == '__main__':
    main()

# moriarty
```

```
# Is DRACULA in lower case? False
```

There is one last method called `casefold()` which is kind of a more aggressive version of the `lower()` method.

According to the official documentation:

> Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter 'ß' is equivalent to "ss". Since it is already lowercase, lower() would do nothing to 'ß'; casefold() converts it to "ss".

The usage of this method is identical to the `lower()` method.

```
def main():
    book_name = 'DRACULA'

    print(book_name.casefold())


if __name__ == '__main__':
    main()

# dracula
```

The `swapcase()` method can do just that.

```
def main():
    book_name = 'HEARTS IN ATLANTIS'

    print(book_name.swapcase())


if __name__ == '__main__':
    main()

# hearts in atlantis
```

As you can see, the method has converted the book name into lower case from upper case.

How to Count the Number of Occurrences of a Substring in a String in Python

If you want to find out the number of occurrences of a substring within a string, you can use the `count()` method in Python.

```
def main():
    paragraph = '''At three in the morning the chief Sussex detect
    Birlstone, arrived from headquarters in a light dog-cart behin
    the morning he had sent his message to Scotland Yard, and he w
    welcome us. White Mason was a quiet, comfortable-looking perso
    ruddy face, a stoutish body, and powerful bandy legs adorned w
```

```
    print(f'The substring "{substring}" shows up {paragraph.count(

if __name__ == '__main__':
    main()

# The substring "morning" shows up 2 times in the paragraph.
```

If you call the `count()` method without passing anything to it, the method will return the number of empty spaces in the given string.

How to Split and Join Strings in Python

You can actually break a string into a list of words or join a list of words in a string in Python.

```
def main():
    string = 'Holmes was certainly not a difficult man to live wit

    word_list = string.split()

    print(word_list)

if __name__ == '__main__':
    main()

# ['Holmes', 'was', 'certainly', 'not', 'a', 'difficult', 'man', '
```

You can override that by passing a custom separator and also fix the number of splits you want.

```
def main():
    string = 'Holmes,was,certainly,not,a,difficult,man,to,live,wit

    word_list = string.split(',', 5)

    print(word_list)


if __name__ == '__main__':
    main()

# ['Holmes', 'was', 'certainly', 'not', 'a', 'difficult,man,to,liv
```

This time I've replaced the spaces in the source string with commas. I've also overridden the default separator with a comma and fixed the number of splits to five.

As you can see in the output, there are five splits and the rest of the string is kept unchanged as the sixth element in the list.

The `split()` method is good for using with data that has been intentionally delimited. Using it with natural text with punctuation may produce unexpected results.

The opposite of the `split()` method is `join()` and it works on any iterator type in Python.

```
    word_list = ['Holmes', 'was', 'certainly', 'not', 'a', 'diffic
    string = ''

    string = string.join(word_list)

    print(string)

    word_list = ['Holmes ', 'was ', 'certainly ', 'not ', 'a ', 'd
    string = ''

    string = string.join(word_list)

    print(string)


if __name__ == '__main__':
    main()

# Holmeswascertainlynotadifficultmantolivewith
# Holmes was certainly not a difficult man to live with
```

There you have it. Notice how the `join()` method didn't care about adding spaces as separator after each word in the first call.

So I appended a space with each word in the list and in the second call the line has become much more readable.

How to Write Conditional Statements in Python

I hope you remember the `boolean` data type from a previous section – the one that can only hold `True` or `False` values.

Well, you can use a boolean with an `if` statement (a conditional statement) in Python to perform an action conditionally.

```
def main():
    number = int(input('what number would you like to check?\n- '))

    if number % 2 == 0:
        print(f"{number} is even.")
    else:
        print(f"{number} is odd.")


if __name__ == '__main__':
    main()

# what number would you like to check?
# - 10
# 10 is even.
```

You start by writing out `if` followed by a condition and a colon. By condition, what I mean is a statement that evaluates to a boolean value (true or false).

You've been using the `==` operator since the beginning and already know that it checks whether the value on the left side of it is equal to the one in the right or not.

You can use the `if...else` statement to choose between two different options. But, if you have multiple options to choose from, you can use the `if...elif...else` statement.

```
def main():
    year = int(input('which year would you like to check?\n- '))

    if year % 400 == 0 and year % 100 == 0:
        print(f"{year} is leap year.")
    elif year % 4 == 0 and year % 100 != 0:
        print(f"{year} is leap year.")
    else:
        print(f"{year} is not leap year.")


if __name__ == '__main__':
    main()

# which year would you like to check?
# - 2004
# 2004 is leap year.
```

The `elif` statement usually goes after an `if` statement and before an `else` statement.

Think of it like "else if", so if the `if` statement fails, then the `elif` will take over. You write it exactly like a regular `if` statement.

If the expressions on both sides of the `and` statement evaluates to `true`, then the whole expression evaluates to `true`. Simple.

Don't worry if you do not understand the `and` operator in detail at the moment. You'll learn about it and its siblings in the very next section.

Another thing you need to understand is that these `if` statements are just regular statements so you can do pretty much anything inside them.

```
def main():
    number = int(input('what number would you like to check?\n- '))

    is_not_prime = False

    if number == 1:
        print(f"{number} is not a prime number.")
    elif number > 1:
        for n in range(2, number):
            if (number % n) == 0:
                is_not_prime = True
                break

        if is_not_prime:
            print(f"{number} is not a prime number.")
        else:
            print(f"{number} is a prime number.")


if __name__ == '__main__':
    main()
```

This example is a bit more complex than what you've seen so far. So let me break it down for you. The program checks whether a given number is a prime number or not.

First, you take a number from the user. For a number to be prime, it has to be divisible only by 1 and itself. Since 1 is only divisible by 1, it's not a prime number.

Now, if the given number is larger than 1, then you'd have to divide the number with all the numbers from 2 to that particular number.

If the number is divisible by any of these numbers, then you'll turn the `is_not_prime` variable to `True` , and `break` the loop.

The `break` statement simply breaks out of a loop immediately. There is also the `continue` statement that can skip the current iteration instead of breaking out.

Finally, if the `is_not_prime` variable is `True` then the number is not prime, otherwise it's a prime number.

So as you can see, not only you can put loops inside a conditional statement but also put conditional statements inside a loop.

The final example that I'd like to show you is the `for...else` statement. As you can see in the example above, you have a `for`

```
def main():
    number = int(input('what number would you like to check?\n- '))

    if number == 1:
        print(f"{number} is not a prime number.")
    elif number > 1:
        for n in range(2, number):
            if (number % n) == 0:
                print(f"{number} is not a prime number.")
                break
        else:
            print(f"{number} is a prime number.")


if __name__ == '__main__':
    main()

# what number would you like to check?
# - 5
# 5 is a prime number.
```

If you put an `else` statement on the same level as a `for` statement, then Python will execute whatever you put inside that `else` block as soon as the loop has finished.

What are Relational and Logical Operators in Python?

In the examples above, you've seen the usage of `==` as well as the `and` operators. In this section, you'll learn about them in detail.

operators.

OPERATOR	EXPLANATION	USAGE
`==`	Equal To	`5 == 5` gives you `True`, but `5 ==`
`!=`	Not Equal To	`5 != 10` gives you `True`, but `5 !=`
`>`	Greater Than	`10 > 5` gives you `True`, but `5 > 1`
`<`	Less Than	`5 < 10` gives you `True`, but `10 <`
`>=`	Greater Than or Equal	`10 >= 5` and `10 >= 10` gives you
`<=`	Less Than or Equal	`5 <= 10` and `5 <= 5` gives you `Tr`

You've been using the `equal to` operator since the very beginning. The other ones you'll learn about as you keep going.

Apart from these, there are three logical operators in Python. They are the `and`, `or`, and `not` operators.

Take an RPG game, for example, where the hero has to have a level 45 or up shield and a level 48 or up sword in order to go to the next level.

```
def main():
    shield = int(input('what is your shield level? '))
    sword = int(input('what is your sword level? '))

    if shield >= 45 and sword >= 48:
        print('you shall pass!')
```

```
if __name__ == '__main__':
    main()

# what is your shield level? 42
# what is your sword level? 52
# you shall not pass!
```

Unless you meet both conditions, the statement will evaluate to `False`. You can have more conditions in a statement like this:

```
def main():
    shield = int(input('what is your shield level? '))
    sword = int(input('what is your sword level? '))
    armor = int(input('what is your armor level? '))

    if shield >= 45 and sword >= 48 and armor >= 25:
        print('you shall pass!')
    else:
        print('you shall not pass!')


if __name__ == '__main__':
    main()

# what is your shield level? 45
# what is your sword level? 50
# what is your armor level? 10
# you shall not pass!
```

The `or` operator, on the other hand, is a bit more forgiving. If any of the given conditions evaluates true, than the entire statement will evaluate to true.

dead.

```
def main():
    age = 10_000
    is_legally_dead = True

    if is_legally_dead or age > 500_000:
        print('you shall pass!')
    else:
        print('you shall not pass!')


if __name__ == '__main__':
    main()

# you shall pass!
```

You can mix the `and` and `or` operators together. I won't list out all the possible combinations of these operators, but as you keep working with Python, you'll get to use a lot of them.

The last logical operator that I'd like to discuss is the `not` operator. This operator takes only one operand and returns the opposite value.

```
def main():
    print('not True =', not True)
    print('not False =', not False)


if __name__ == '__main__':
    main()
```

For example, if you change the rules of the horror game we talked about in the previous example and make is so that only people who are over 500,000 years old and not Van Helsing can enter the castle.

```
def main():
    age = 800_000
    is_van_helsing = True

    if age > 500_000 and not is_van_helsing:
        print('you shall pass!')
    else:
        print('you shall not pass!')


if __name__ == '__main__':
    main()

# you shall not pass!
```

Since we've been talking about conditional statements and some of the operators associated with them, I'd like to introduce you to another statement first introduced in Python 3.10, the `match...case` statement.

Python Switch Statement – Switch Case Example

Since my colleague Kolade Chris has written such a nice article on the topic, I'll not repeat that here. Feel free to check it out at your leisure.

What Are Assignment Operators in Python?

You've already encountered the simple assignment operator which is the `=` sign you used to assign a value to a variable.

Now there are a few variations to this operator that you can use to perform arithmetic and bitwise operations while also assigning a value.

Bitwise operations are a little out of the scope of this book, so I'll stick to the arithmetic operations.

There are seven different assignment operators in Python. Since you've already learned about the simple one, I'll discuss the other six in the following table.

OPERATOR	USAGE	EQUIVALENT TO
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`

/=	a /= b	a = a / b
%=	a %= b	a = a % b
**=	a **= b	a = a ** b

These operators are not exclusive to Python, and in most programming resources, you'll find these in a much earlier chapter.

But I wanted to wait until you've learned about taking input from the user, working with ranges, and looping through them before introducting them here.

Assume that you want to write a program that calculates the sum of all the numbers within a given range.

```
def main():
    start = int(input('which number do you want to start from?\n-
    end = int(input('which number do you want to stop at?\n- '))

    total = 0

    for number in range(start, end + 1):
        total += number

    print(f"the sum of the numbers between {start} and {end} is: {


if __name__ == '__main__':
    main()
```

```
# the sum of the numbers between 1 and 10 is: 55
```

I hope you remember that the ending number of a `range()` function call is exclusive. So I had to add a `+1` with the ending number.

Otherwise it's a very simple range based for loop that adds each number to the `total` variable and prints it out once the loop has finished.

What Is the Set Type in Python?

So far you've learned about a number of iterable types such as lists, tuples, and also strings. There is another one known as a set. Let's look at an example:

```
def main():
    numbers = {1, 2, 3, 4, 5}

    for number in numbers:
        print(number)


if __name__ == '__main__':
    main()

# 1
# 2
```

You can create a new set by putting the values between a set of curly braces. Keep in mind however, you can not create an empty set using the curly braces.

You'll have to use the `set()` function for that.

```
def main():
    numbers = {}

    print(type(numbers))

    numbers = set()

    print(type(numbers))


if __name__ == '__main__':
    main()

# <class 'dict'>
# <class 'set'>
```

As you can see, usage of empty curly braces creates a dictionary whereas the `set()` function creates an empty set.

Sets may seem similar to lists, but they are actually quite different. For starters, you can not put duplicate values in a set.

```
    print(numbers_list)

    numbers_set = set(numbers_list)

    print(numbers_set)


if __name__ == '__main__':
    main()

# [1, 2, 3, 4, 5, 3, 2, 4]
# {1, 2, 3, 4, 5}
```

The list of numbers can hold duplicate values without any problem. But as soon as you create a set from that list, all the duplicate values will be gone.

Sets are mutable, so you can add new values to a set using the `add()` method.

```
def main():
    numbers = {1, 2, 3, 4, 5}

    numbers.add(500)

    print(numbers)


if __name__ == '__main__':
    main()

# {1, 2, 3, 4, 5, 500}
```

altogether.

```
def main():
    numbers = {1, 2, 3, 4, 5}

    numbers.discard(3)

    print(numbers)

    numbers.clear()

    print(numbers)


if __name__ == '__main__':
    main()

# {1, 2, 4, 5}
# set()
```

Notice how an empty set shows up as `set()` instead of `{}` because the latter indicates an empty dictionary.

Apart from the fact that a set never contains duplicate values, there is another speciality of this type.

You can perform set operations such as union, intersection, complement, and difference using sets in Python.

My colleague Estefania Cassingena Navone has written an excellent guide on sets, frozen set and all the operations that you can perform on them.

Finally, if you'd like to get a definitive look at the set type, the official documentation will more than suffice.

What Is the Mapping Type in Python?

You've already learned about the sequence types and set types in Python. Those are really useful for containing a bunch of data.

But situations where you want to store data in key value pairs are not uncommon. Take, for example, an online bookshop where you have to store the prices of the books.

```
def main():
    programming_books = {
        'C Programming Language': 35,
        'Introduction to Algorithms': 100,
        'Clean Code: A Handbook of Agile Software Craftsmanship':
    }

    print(programming_books)


if __name__ == '__main__':
```

The variable `programming_books` here is a mapping type usually known as a dictionary. Declaring a dictionary is similar to declaring a list or tuple but you use a set of curly braces instead of square braces or parenthesis.

Enclosed within the braces are a bunch of key value pairs. The strings on the left side are the keys and the numbers are the values. You can access any of the keys using the `get()` method.

```
def main():
    programming_books = {
        'C Programming Language': 35,
        'Introduction to Algorithms': 100,
        'Clean Code: A Handbook of Agile Software Craftsmanship':
    }

    cpl = 'C Programming Language'
    algo = 'Introduction to Algorithms'

    print(f"The price of {cpl} is ${programming_books.get(cpl)}")
    print(f"The price of {algo} is ${programming_books[algo]}")


if __name__ == '__main__':
    main()

# The price of C Programming Language is $35
# The price of Introduction to Algorithms is $100
```

Dictionaries are mutable which means you can add new items to them, remove or change existing items.

```
def main():
    programming_books = {
        'C Programming Language': 35,
        'Introduction to Algorithms': 100,
        'Clean Code: A Handbook of Agile Software Craftsmanship':
    }

    key = 'C Programming Language'

    programming_books[key] = 45

    programming_books['The Pragmatic Programmer'] = 32

    print(programming_books)


if __name__ == '__main__':
    main()

# {'C Programming Language': 45, 'Introduction to Algorithms': 100
```

You can change an existing item by accessing the item using the square braces and assign a new value to it. The price of C Programming Language has gone up by $10.

If you put a nonexistent key in between the square braces, then that'll show up as a new item. The price of The Pragmatic

For removing an item from a dictionary, you can use the `popitem()` or the `pop()` method.

```
def main():
    programming_books = {
        'C Programming Language': 35,
        'Introduction to Algorithms': 100,
        'Clean Code: A Handbook of Agile Software Craftsmanship':
    }

    print(programming_books.popitem())

    key = 'C Programming Language'

    print(programming_books.pop(key))

    print(programming_books)


if __name__ == '__main__':
    main()

# ('Clean Code: A Handbook of Agile Software Craftsmanship', 50)
# 35
# {'Introduction to Algorithms': 100}
```

The `popitem()` method removes the last item in the dictionary and returns that as a tuple.

The `pop()` method, on the other hand, returns the value for a given key and removes the pair.

Finally, there is the `clear()` method that wipes out all the pairs in a given dictionary in one go.

```
def main():
    programming_books = {
        'C Programming Language': 35,
        'Introduction to Algorithms': 100,
        'Clean Code: A Handbook of Agile Software Craftsmanship':
    }

    programming_books.clear()

    print(programming_books)


if __name__ == '__main__':
    main()

# {}
```

What Are Dictionary View Objects in Python?

So far in this section, you've seen dictionaries printed out as long comma separated lines between pairs of curly braces – but that's not very readable.

This is where the view objects come in handy. You can call some specific methods on dictionaries and get view objects in return.

```
def main():
    programming_books = {
        'C Programming Language': 35,
        'Introduction to Algorithms': 100,
        'Clean Code: A Handbook of Agile Software Craftsmanship':
    }

    for key in programming_books.keys():
        print(key)


if __name__ == '__main__':
    main()

# C Programming Language
# Introduction to Algorithms
# Clean Code: A Handbook of Agile Software Craftsmanship
```

Just like the `keys()` method, there is the `values()` method that returns the values in a dictionary instead.

```
def main():
    programming_books = {
        'C Programming Language': 35,
        'Introduction to Algorithms': 100,
        'Clean Code: A Handbook of Agile Software Craftsmanship':
    }

    for value in programming_books.values():
        print(value)
```

```
# 35
# 100
# 50
```

Finally, if you want both the keys and the values as tuples, you can use the `items` method.

```
def main():
    programming_books = {
        'C Programming Language': 35,
        'Introduction to Algorithms': 100,
        'Clean Code: A Handbook of Agile Software Craftsmanship':
    }

    for item in programming_books.items():
        print(item)


if __name__ == '__main__':
    main()

# ('C Programming Language', 35)
# ('Introduction to Algorithms', 100)
# ('Clean Code: A Handbook of Agile Software Craftsmanship', 50)
```

How to Write Functions in Python

```
def print_hello():
    print('Hello, World!')


def main():
    print_hello()


if __name__ == '__main__':
    main()

# Hello, World!
```

You define a function by writing out `def` followed by the name of the funtion and a colon. You can then write the function body from the next indented line.

In this example, the `print_hello()` prints out `Hello, World!` on the terminal. It doesn't accept any argument.

```
def print_hello(message):
    print(message)


def main():
    print_hello('Hello, Universe!')


if __name__ == '__main__':
    main()
```

Now instead of printing out `Hello, World!` all the time, you can pass a custom message for the function to print out.

You can make a function accept multiple arguments and even set a default value for it.

```
def print_hello(message, is_lower=False):
    if is_lower:
        print(message.lower())
    else:
        print(message.upper())


def main():
    print_hello('Hello, Universe!')
    print_hello('Hello, Universe!', True)


if __name__ == '__main__':
    main()

# HELLO, UNIVERSE!
# hello, universe!
```

Setting a default value to a function parameter makes it optional. So if you do not pass a value during the function call, your program will use the default value.

Instead of printing out the message outright, you can make the function return the message.

```
        return message.lower()
    else:
        return message.upper()


def main():
    print(hello('Hello, Universe!'))
    print(hello('Hello, Universe!', True))


if __name__ == '__main__':
    main()

# HELLO, UNIVERSE!
# hello, universe!
```

Since the function doesn't print out the message anymore, changing its name from `print_hello()` to just `hello()` makes more sense.

When you call the function with or without a custom message, the function returns a string that you can then print out within the `main()` function.

You can also save the message in variables instead of passing them to the `print()` function directly.

```
def hello(message, is_lower=False):
    if is_lower:
        return message.lower()
    else:
        return message.upper()
```

```
    lowercase_message = hello('Hello, Universe!', True)
    print(lowercase_message)


if __name__ == '__main__':
    main()

# HELLO, UNIVERSE!
# hello, universe!
```

It's not that you can only pass singular values to a function. You can pass lists, tuples, dictionaries or any other object to a function.

```
def total(numbers):
    s = 0
    for number in numbers:
        s += number
    return s


def main():
    print(total([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))


if __name__ == '__main__':
    main()

# 55
```

In this function, you can pass a list of numbers to and get their sum. I had to name the function `total()` instead of `sum()`

discuss in this section, and that is recursion.

Recursion in Python or programming in general is the technique of making a function call itself to perform a task iteratively.

For example, imagine a function that accepts an integer and calculates the sum of all natural numbers up to that given integer. You can write this program using loops.

```
def natural_sum(last_number):
    if last_number < 1:
        return last_number

    total = 0
    for number in range(1, last_number + 1):
        total += number

    return total


def main():
    last_number = int(input('up to which number would you like to

    print(natural_sum(last_number))


if __name__ == '__main__':
    main()

# up to which number would you like to calculate the sum?
# - 10
# 55
```

loop.

```
def recursive_natural_sum(last_number):
    if last_number < 1:
        return last_number

    return last_number + recursive_natural_sum(last_number - 1)


def main():
    last_number = int(input('up to which number would you like to

    print(recursive_natural_sum(last_number))


if __name__ == '__main__':
    main()

# up to which number would you like to calculate the sum?
# - 10
# 55
```

At a glance, this piece of code may look very complicated to you. But in reality it's very simple. Let's break it down step by step.

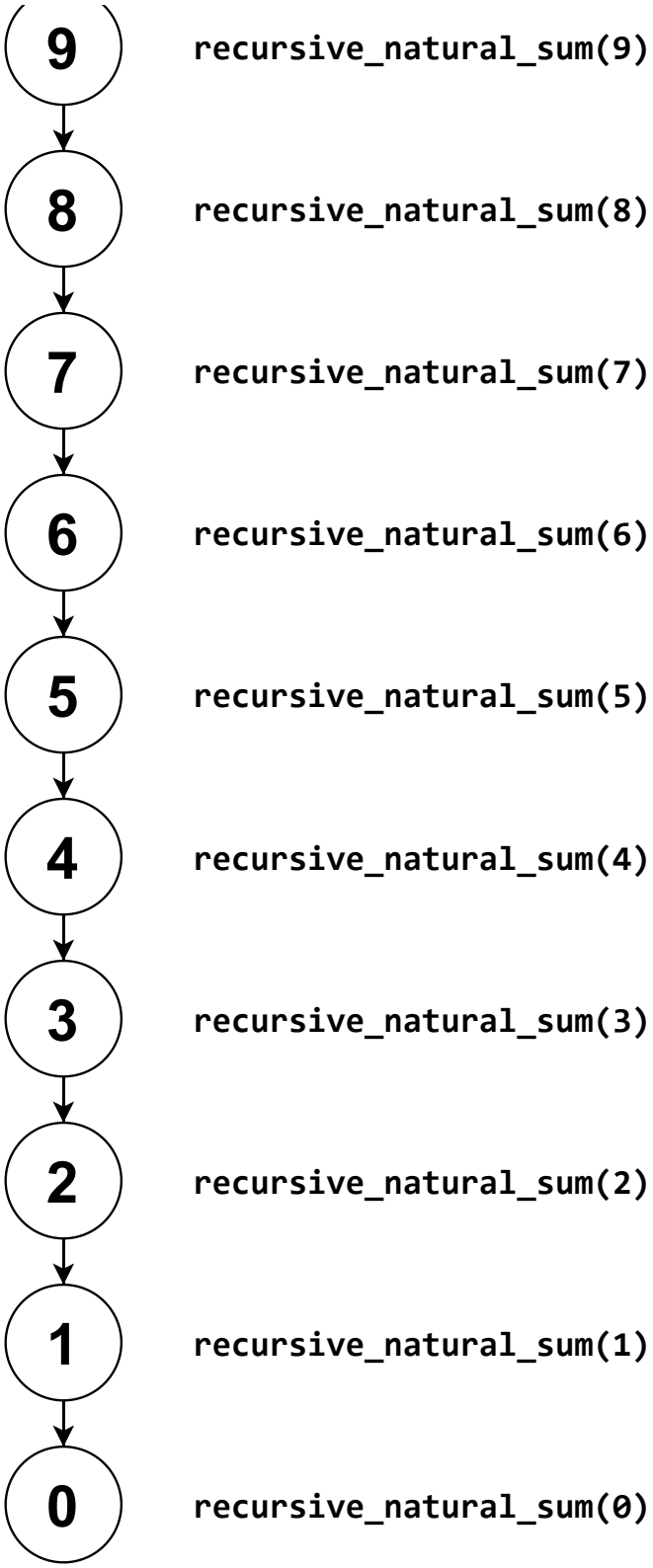
When you call the `recursive_natural_sum()` function with the value of 10 for the first time, you start a chain reaction of sorts.

Since the value is not less than 1, the `if` statement evaluates to `False` and the second `return` statement gets called.

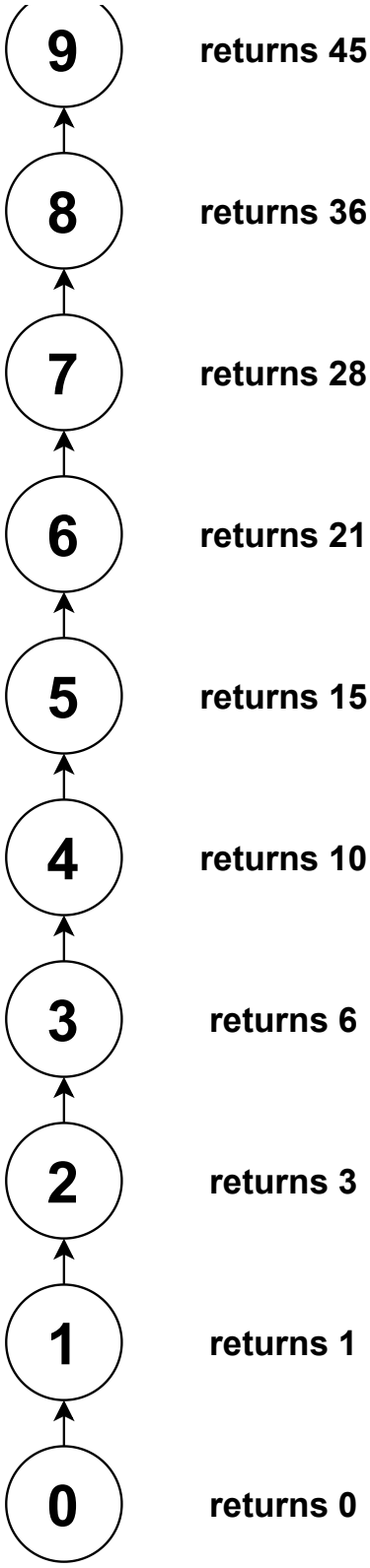
last_number = 1 What is 1 at this point?

You're also adding the returned value from this call to the current value of the `last_number` variable.

But you'll not get a return value because your inner function call will call itself again with `last_number - 1` which will be 8 at that point.



This calling goes on and on until the value of `last_digit` becomes zero. Once it becomes zero, the `if` statement evaluates to `True` and the function calls start to return a value.



The value returned from each function call is `last_digit + (last_digit - 1)` by the end of the recursion chain it adds up to 55.

I'm not sating that recursive functions are easier than loops – but at times, using a recursive function instead of nested loops can be more efficient.

How to Write Anonymous or Lambda Functions in Python

Anonymous or lambda functions are functions without any name. This is not something exclusive to Python and most of the modern programming languages have some sort of lambda implementation.

Instead of beginning the function declaration with `def`, you instead start with writing out `lambda` followed by a colon and the function body.

```
print_hello = lambda: print('Hello, World!')
```

```
if __name__ == '__main__':
    main()

# Hello, World!
```

Since lambdas do not have a name, you have to put it in a variable in order to access it but that's not recommended. If you need a named function, use `def` instead.

Lambda functions are useful when you want to pass a function as an argument to another function call. Take the `filter()` function for example.

```
def check_even(number):
    if number % 2 == 0:
        return True
    else:
        return False


def main():
    numbers = [1, 2, 5, 4, 7, 88, 12, 15, 55, 77, 95]

    even_numbers = filter(check_even, numbers)

    print(list(even_numbers))


if __name__ == '__main__':
    main()

# [2, 4, 88, 12]
```

its two arguments. The function should describe the logic for filtering and the iterable type will contain the values you want to filter from.

In this code, you have a list of numbers and you want to filter out the odd numbers from that list.

The `check_even()` function takes a number as argument. It then returns `True` if the number is divisible by two and `False` if not.

The `filter()` function iterates through the list of numbers and passes each number to the `check_even()` function.

It keeps the number if the `check_even()` function returns `True` or discards the number if the `check_even()` function returns `False`.

Now this `check_even()` function doesn't have any purpose other than checking if a given number is divisible by two or not. So you can write it as a lambda.

```
def main():
    numbers = [1, 2, 5, 4, 7, 88, 12, 15, 55, 77, 95]

    even_numbers = filter(lambda number: True if number % 2 == 0 e

    print(list(even_numbers))


if __name__ == '__main__':
    main()
```

This lambda takes an argument named `number` then returns `True` if that's divisible by two and `False` otherwise.

You can add multiple arguments separating each by a comma. Finally a lambda doesn't need a return statement but you can assume a return.

So `True if number % 2 == 0 else False` is equivalent to `return True if number % 2 == 0 else False`. The `if...else` statement inside the lambda is in short hand form.

How to Work with local, nonlocal and global Variables in Python

Scope of a variable in Python or programming in general refers to region where that variable is accessible.

```
def outside():
    message = 'Hello, World!'


def main():
    print(message)


if __name__ == '__main__':
    main()

# NameError: name 'msg' is not defined
```

else.

As a result, when you try to access that variable from the `main()` function, you get a `NameError` since the variable is scoped to the `outside()` function.

Variables like this are called local variables and they only exist within the block they've been declared.

Global variables, on the other hand, are usually declared outside of any particular block of code.

```
message = 'Hello, World!'


def main():
    print(message)


if __name__ == '__main__':
    main()

# Hello, World!
```

As you can see, now the `message` variable has no indentation and is declared at the top of the function. You could've declared the variable after the `main()` function.

```
message = 'Hello, World!'

if __name__ == '__main__':
    main()

# Hello, World!
```

This works because you don't try to access the variable until you call the `main()` function inside the `if` statement.

Although global variables are accessible pretty much everywhere, it can be a bit tricky to work with if you have a local variable with a similar name.

```
message = 'Hello, {name}!'


def main():
    message = message.format(name='Farhan')
    print(message)


if __name__ == '__main__':
    main()

# UnboundLocalError: local variable 'message' referenced before as
```

In this code, you have a placeholder for a name inside the `message` variable. You can use the `format()` method to put a

`'message' referenced before assignment` message. In simpler words, you're trying to access a local variable named `message` before even assigning anything to it.

So clearly, Python is looking for a local variable with the given name instead of accessing the global one. Since it's asking, try giving it a local variable.

```
message = 'Hello, {name}!'


def main():
    message = str()

    message = message.format(name='Farhan')
    print(message)


if __name__ == '__main__':
    main()
```

This time the error will be gone but you will not get any output in your console. This is because the local `message` variable is empty and there is no placeholder to put a name in.

This is where the `global` keyword comes in. Instead of creating a local variable, you can let Python know that you're trying to access the global `message` variable.

```
def main():
    global message

    message = message.format(name='Farhan')
    print(message)


if __name__ == '__main__':
    main()

# Hello, Farhan!
```

Now, instead of trying to look for a variable named `message` within the local scope, Python will directly reach out to the global scope.

Finally, there is the `nonlocal` keyword usually used in nested functions. It solves a similar problem as the `global` keyword but in a local scope.

```
def greet(name):
    message = 'Hello, {name}!'

    def include_name():
        message = message.format(name=name)

    include_name()
    return message


def main():
    print(greet('Farhan'))
```

In this example, you're dealing with three functions. There is the `main()` function, there is the `greet()` function, and inside that is the `include_name()` function.

The `greet()` function takes a name as an argument but doesn't include that in the message right away.

Instead it calls the `include_name()` function defined within its local scope. That's where the problem begins.

You see, the `message` variable is outside the scope of the `include_message()` function and that's why you're getting the `referenced before assignment` error message.

```
def greet(name):
    message = 'Hello, {name}!'

    def include_name():
        global message

        message = message.format(name=name)

    include_name()
    return message


def main():
    print(greet('Farhan'))
```

```
# NameError: name 'message' is not defined
```

You can't use the `global` keyword either since the `message` variable is not defined in the global scope and that's what the error message dictates.

You can use the `nonlocal` keyword to use variables that are not in the global scope but in the scope of the outer function.

```
def greet(name):
    message = 'Hello, {name}!'

    def include_name():
        nonlocal message
        message = message.format(name=name)

    include_name()
    return message


def main():
    print(greet('Farhan'))


if __name__ == '__main__':
    main()

# Hello, Farhan!
```

Now the `include_name()` function will look for the `message` variable within the scope of the `greet()` function instead of its

How to Pass a Variable Number of Arguments to a Function Using *args and **kwargs in Python

Imagine a function that takes a bunch of numbers as arguments and returns their sum. In a function like this, it'd be nice to have the provision of passing a variable number of arguments.

Surely you can pass the numbers as a tuple or as a list but you may want to pass them as regular arguments separated by commas. You can do that by using `*args` or non key arguments in Python.

```
def total(*args):
    print(type(args))

    t = 0
    for arg in args:
        t += arg

    return t


def main():
    print(total(1, 2, 3, 4, 5))


if __name__ == '__main__':
    main()

# <class 'tuple'>
# 15
```

It's not mandatory to name the argument as `*args`, you can call it something more descriptive like `*numbers` or anything else. As long as you put the asterisk in front, you're good to go.

Like `*args` there is also `**kwargs` or keyword arguments that will allow you to access the function arguments as a dictionary.

```
def items(**kwargs):
    print(type(kwargs))

    for key, value in kwargs.items():
        print(f"{key} : {value}")


def main():
    items(
        Apple=10,
        Orange=8,
        Grape=35
    )


if __name__ == '__main__':
    main()

# <class 'dict'>
# Apple : 10
# Orange : 8
# Grape : 35
```

In this case, you can pass arbitrary number of key-value pairs and access them as a dictionary inside the `items()` function.

As long as you put the double asterisks at the front, you'll be fine. The `items()` method within dictionaries lets you iterate through them.

You can also change the names of the `key` and `value` variables. A more readable version of the function can be as follows:

```
def items(**fruits):
    print(type(fruits))

    for fruit, price in fruits.items():
        print(f"{fruit} : {price}")


def main():
    items(
        Apple=10,
        Orange=8,
        Grape=35
    )


if __name__ == '__main__':
    main()

# <class 'dict'>
# Apple : 10
# Orange : 8
# Grape : 35
```

Keep in mind that the type of the keys in this case has to be a string and the values can be anything you want.

As you project grows, breaking off your code into multiple files becomes a necessity. A module in Python is just a file containing Python code that you can import inside other Python files.

For example, assume that you have a Python project with two files. The first one may be "mathstuff.py" and the other one may be "main.py".

The "mathstuff.py" file can contain stuff related to mathematics, for example a function that sums up all the natural numbers in a range.

```
# mathstuff.py

def natural_sum(last_number):
    if last_number < 1:
        return last_number

    total = 0
    for number in range(1, last_number + 1):
        total += number

    return total
```

Now you can import this function any other file such as the "main.py" file.

```
import mathstuff
```

```
    print(mathstuff.natural_sum(last_number))


if __name__ == '__main__':
    main()

# up to which number would you like to calculate the sum?
# - 10
# 55
```

The `import` statement, as the name suggests, imports bits of code from another file or module.

It's not uncommon to house more than one function, variable, or other object in a Python module and often times you may want to use only few of them.

You can use the `from...import` statement in these situations.

```
from mathstuff import natural_sum


def main():
    last_number = int(input('up to which number would you like to

    print(natural_sum(last_number))


if __name__ == '__main__':
    main()

# up to which number would you like to calculate the sum?
```

It also saves you from having to write the module name everytime you want to access a function or object living inside that module.

Finally, you can use the `as` keyword to change the name of an imported module to make that more easily accessible.

```
import mathstuff as math


def main():
    last_number = int(input('up to which number would you like to

    print(math.natural_sum(last_number))


if __name__ == '__main__':
    main()

# up to which number would you like to calculate the sum?
# - 10
# 55
```

Also works with the `from...import` statement.

```
from mathstuff import natural_sum as nsum


def main():
```

```
if __name__ == '__main__':
    main()

# up to which number would you like to calculate the sum?
# - 10
# 55
```

Importing modules is something that you'll have to do all the time. Apart from modules, there is also the idea of packages.

In these examples, both files are in the same folder. Packages are a nifty little way of keeping related Python modules together in different folders.

For example, in a web framework, you may have a package called `framework` that houses all the code that comes with this web framework.

Now this `framework` package can in turn have multiple subpackages – for example there may be a package named `http` for handling HTTP requests and responses.

```
├──framework
│   └──http
```

```
├──framework
│   │   __init__.py
│   │
│   └──http
│           __init__.py
```

Now these files have turned into packages. These "__init__.py" files will tell the Python import system that these folders are indeed packages.

Finally, to put some code inside the `http` package, create a file named `response.py` with the following content:

```
# framework/http/response.py

from json import dumps


def as_json(message):
    return dumps({
        'message': message
    })
```

First, you're importing the `dumps` function from the `json` package. These are part of the Python standard library.

The `dumps` function can turn a Python object like a dictionary into a JSON string, which means the `as_json()` function

```
{"message": "Hello, World"}
```

Now you can import this function in the "main.py" file.

```
from framework.http.response import as_json


def main():
    print(as_json('Hello, World!'))


if __name__ == '__main__':
    main()

# {"message": "Hello, World"}
```

Instead of putting the `as_json()` function inside another Python file, you can simply put it inside the "framework/http/__init__.py" file.

Then you can update the "main.py" file to use the updated package path.

```
from framework.http import as_json


def main():
    print(as_json('Hello, World!'))
```

```
# {"message": "Hello, World"}
```

If you ever try out a framework like Django, you'll see that the framework contains a huge amount of packages, so understanding how the import system works will help you out immensely.

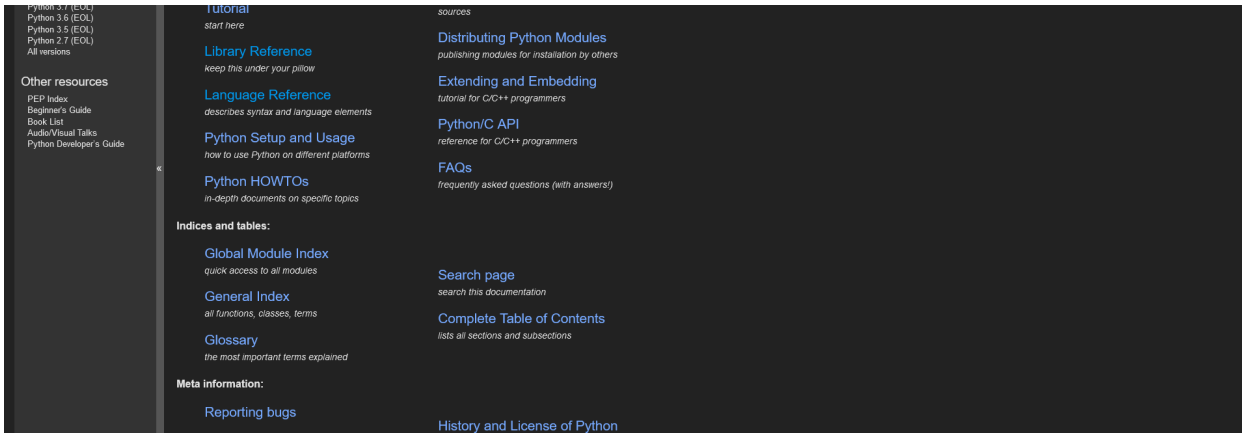
How to Use the Python Documentation Efficiently

Since you're now out of infancy as a Python programmer, I'd like to show you how you can browse through the official Python documentation.

You may think, well browsing documentation is not hard and you'd be absolutely right. But it can be daunting at first.

So what I'm going to do is give you a little primer on how I have used the documentation throughout my career.

The first step is to visit https://docs.python.org/ and you'll automatically land on the documentation for the latest version of Python.



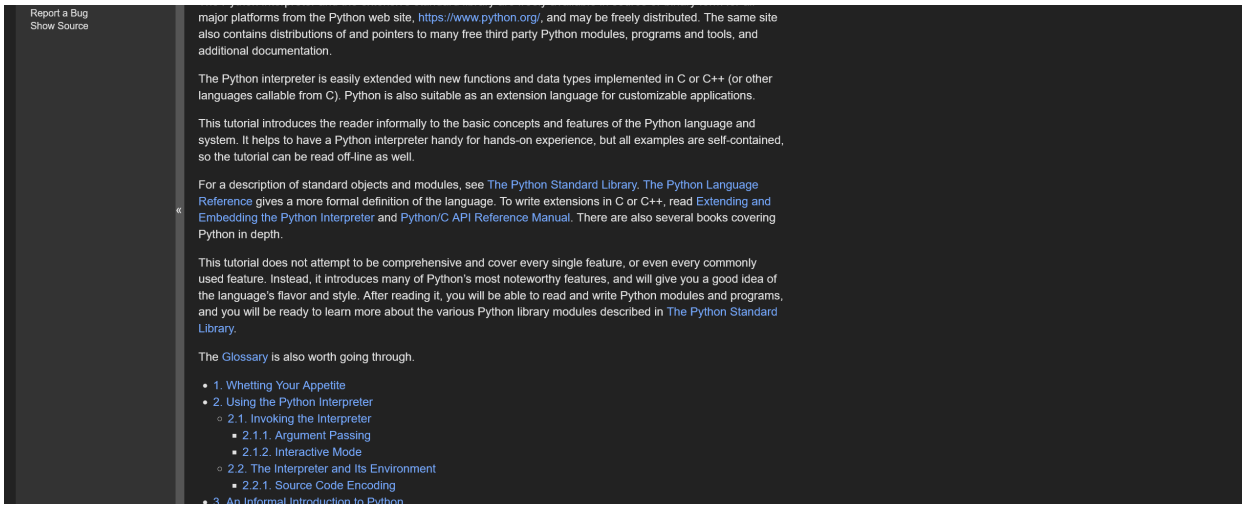
Python Documentation (https://docs.python.org/)

At the time of writing, the latest version of Python is 3.11.4 however I still have version 3.10.11 installed on my computers.

Right from the get go, you can see lots of different links other pages and to be honest, you're not going to need all of them right away.

The best way to find out which link to what page is to to have a look at whichever looks interesting to you.

I'll talk about three links from this page that have helped me immensely. The first one is the "Tutorial" page.

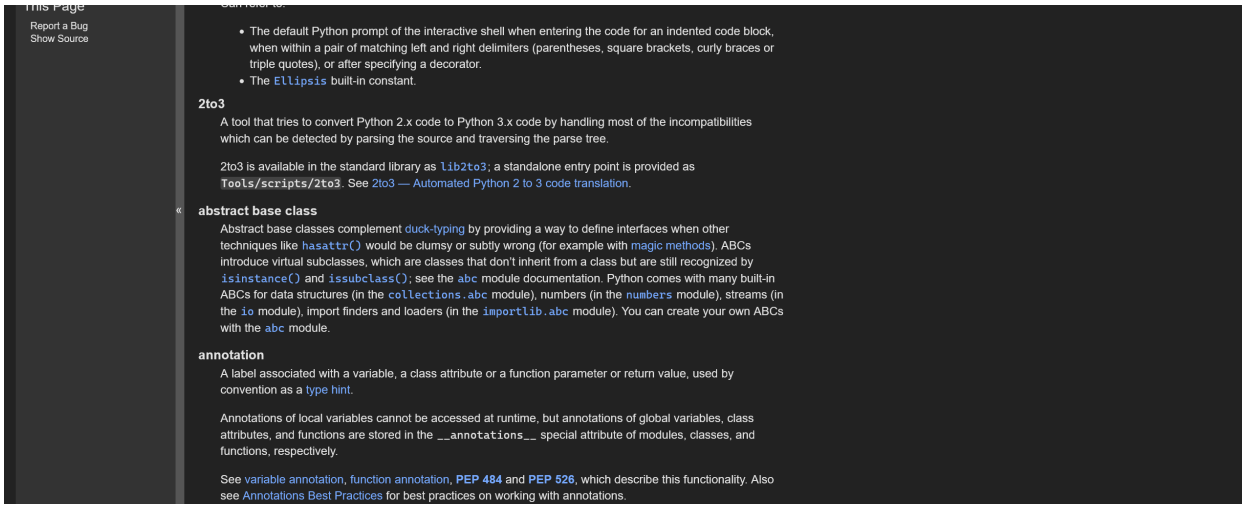


The Python Tutorial (https://docs.python.org/3/tutorial/index.html)

Back when I was making my shift from C to Python, this is the tutorial I went through. The tutorial starts with an introduction to the Python interpreter.

Then it teaches you topics including but not limited to data types, control flow statements, data structures, modules, error handling, the standard library, and even object oriented programming.

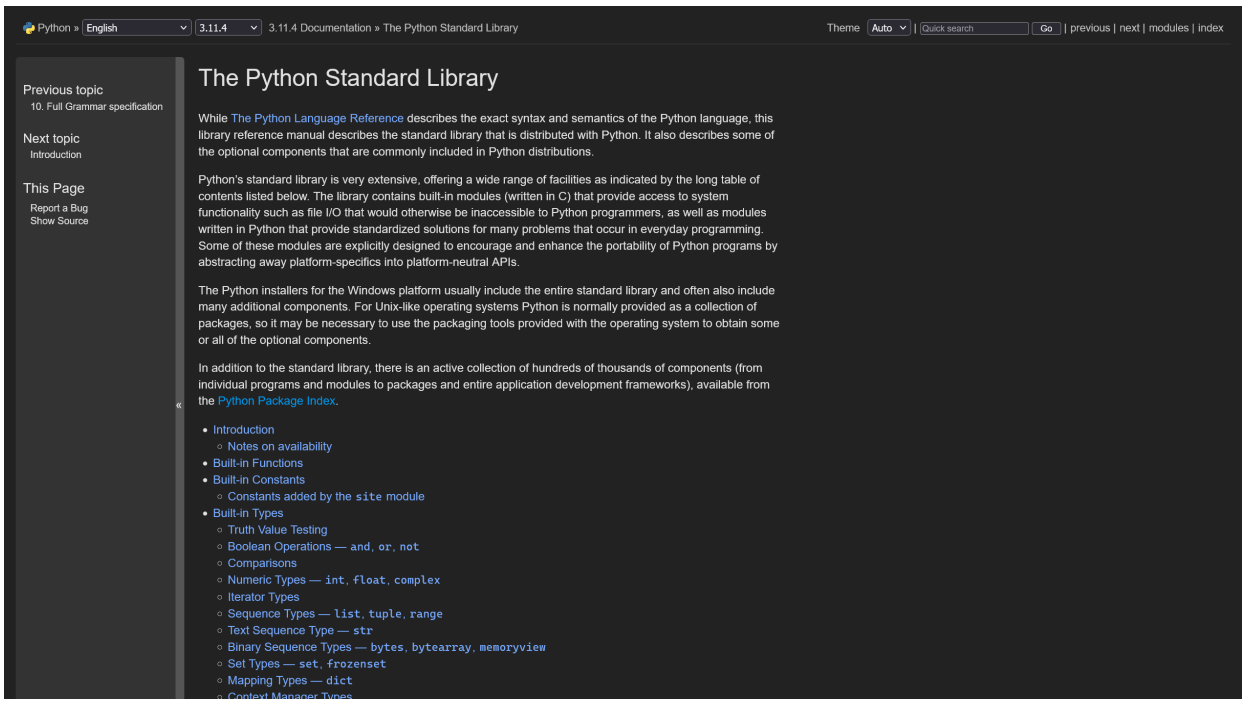
The other page that's extremely useful is the "Glossary" page. It contains a list of all the important terminology that you may come across while working with Python.



Glossary (https://docs.python.org/3/glossary.html)

So at any point if you feel like that you do not know the meaning of a word, take a look at the glossary.

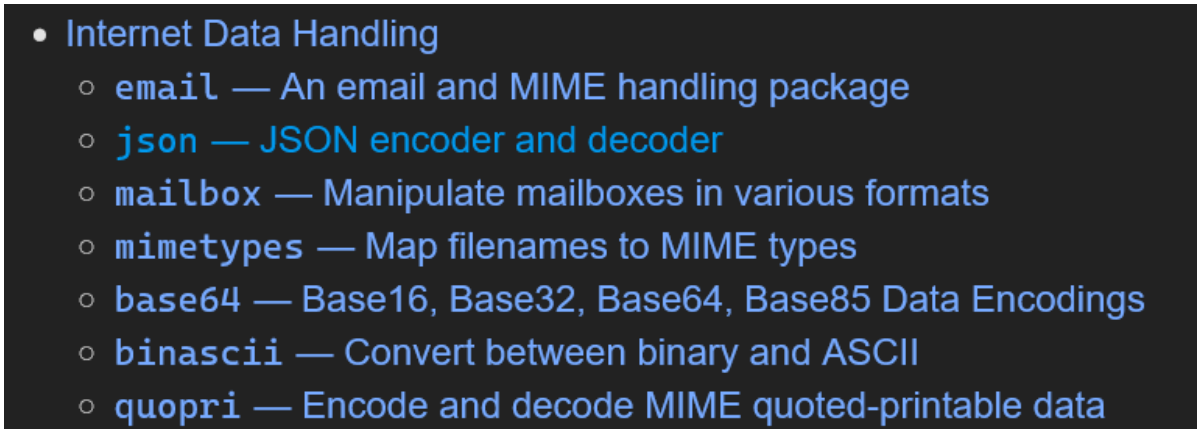
Finally the "Library Reference" page is a detailed description of everything included in the standard Python library.



Library Reference (https://docs.python.org/3/library/index.html)

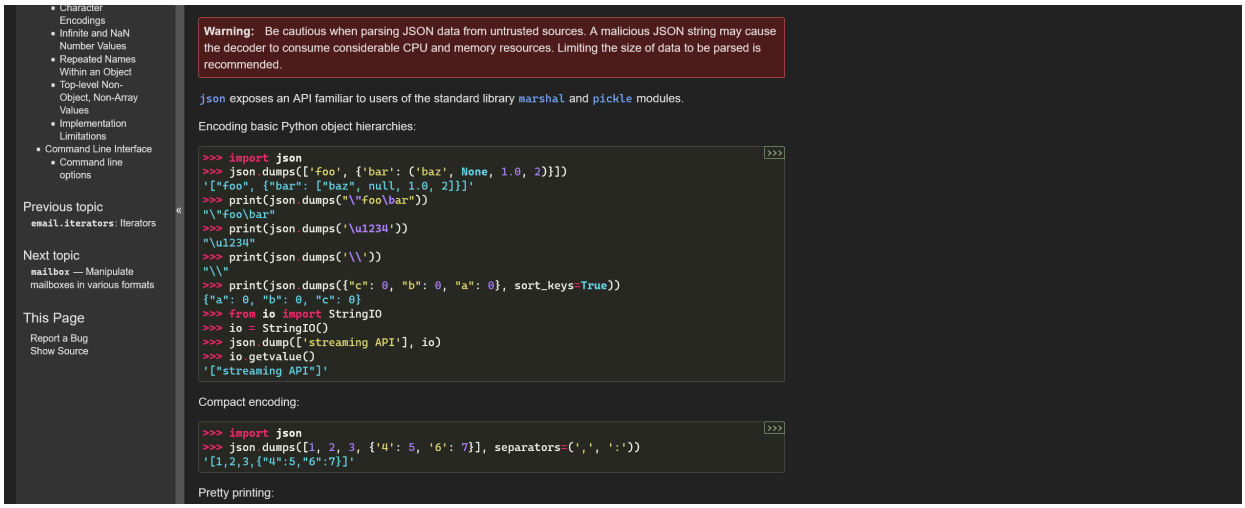
under the "Built-in Types" section.

Or if you want to know about something else such as the JSON package, you can search the library reference for JSON – and sure enough you'll find something on it.



JSON is under the Internet Data Handling section ()

Following the link will land you on the page describing how the JSON package works.



JSON encoder and decoder (https://docs.python.org/3/library/json.html)

The page not only contains text but also contains practical and very useful code examples.

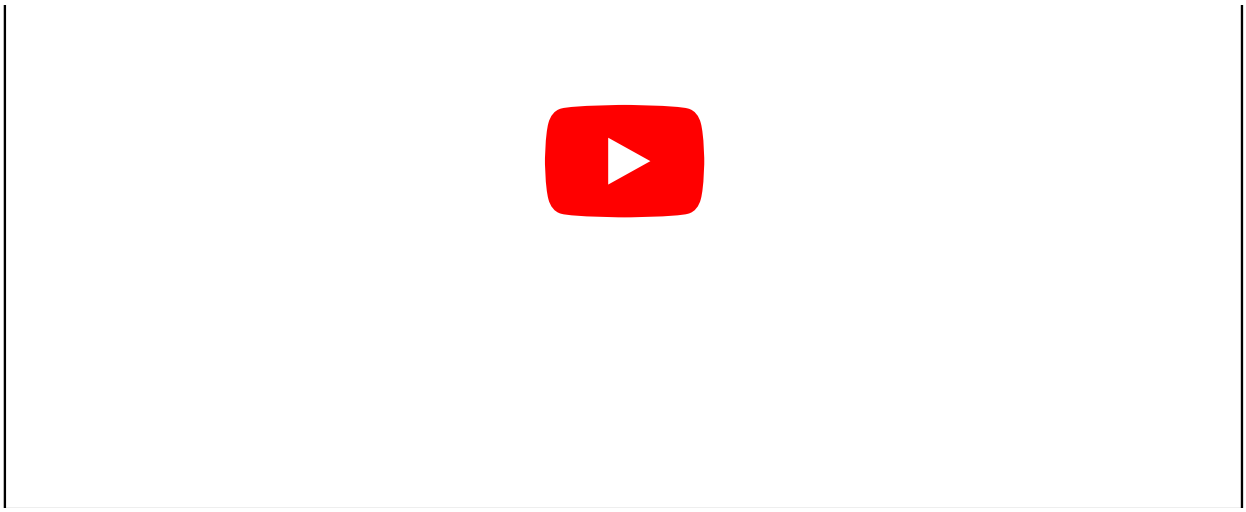
The official documentation is going to be your most reliable and in-depth source of learning, so the sooner you get used to it the better.

What's Next?

As I've said, this text is not a definitive guide to Python – which means there is still a lot to learn. In this section I'll list out a number of different resources.

Object Oriented Programming

The first thing that you may want to learn right after finishing this handbook is object oriented programming with Python.



This comprehensive video course is hosted on the freeCodeCamp YouTube channel. It's a little over 2 hours long and covers the essential concepts nicely.

Object Oriented Programming is not just about learning about concepts like classes, objects, and inheritance.

Writing good object oriented code takes a lot of practice and it all begins with the basics. Take your time with this one and make sure to understand everything.

Algorithms and Data Structures

The second item on the list that you should absolutely learn if you're serious about being an efficient programmer is data structures and algorithms.

Fortunately, the freeCodeCamp YouTube channel hosts a very comprehensive video produced by some of the finest teachers out there on the topic.

The video is a little over 5 hours long and will teach everything you need to know about data structures and algorithms as beginner.

This course is not going to turn you into a better programmer instantly, but it'll teach you a better and more efficient way of thinking about problems.

Django

If you'd like to get into web development using Python, Django is among the most popular choices out there.

The freeCodeCamp YouTube channel hosts this massive 18 hour long course taught by Dr. Chuck, one of the best teachers in the world.

The course not only teaches Django from the ground up but also a long list of concepts around the web itself.

Having a good understanding of object oriented programming is important before you jump into the world of Django, so make sure you have that.

Qt

Python may not be the most popular languages for building graphical user interfaces, but it's surprisingly capable on that end, too.

Qt is a very popular cross-platform UI framework and PySide6 is the official Python bindings for Qt 6.

In this 5 hour long course, you'l learn all the fundamentals of creating user interfaces using Qt and create cross-platform, robust software in no time.

PyGame

Just like cross-platform graphical user interfaces, Python is not the most popular choice when it comes to game programing.

However, the PyGame library is a very powerful and easy to use library for writing 2D games in Python.

In this almost 7 hour long course on game programming with Python, you'll learn about writing a game that mimicks the very popular Stardew Valley.

Undoubtedly, this is a very challenging video to go through but so is making games. So if you're into gamedev and Python, this may be the course you need.

Data Science

Data science is arguably the most popular field where Python plays a huge role. Becoming a data scientist can take years but you gotta start somewhere.

This 12 hour long course on the freeCodeCamp YouTube channel teaches you a lot about how to use your Python knowledge in data science.

Although the course doesn't go very deep into the realm of data science, it teaches you about a number of very important libraries used regularly in data science.

Near the end of the course, you'll also create a project by applying everything you learn throughout the course.

Conclusion

I would like to thank you from the bottom of my heart for the time you've spent reading this article.

Although I've listed only a small number of courses here, the freeCodeCamp YouTube channel is just filled with excellent Python learning resources.

I also have a personal blog where I write about random tech stuff, so if you're interested in something like that, checkout https://farhan.dev.

If you have any questions or are confused about anything – or just want to get in touch – I'm available on Twitter and LinkedIn.

Farhan Hasin Chowdhury
Software developer with a knack for learning new things and writing about them

---

If you read this far, tweet to the author to show them you care.

Tweet a thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) charity organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Trending Guides

Bash If Statement

Align Text in HTML

JS Copy an Object

Tableau vs Power BI

What is rem in CSS?

Quick Sort Algorithm

CS vs IT

JS Append

SQL Select

PHP Explode

PHP Implode

What is YAML?

Declare an Array in JS
Python String Contains
Append String in Python
What is Data Analytics?
Git Revert to Last Commit
What is an IDE in Coding?
Convert Int to String C++
SQL Date Example
JavaScript Minify
Data Analyst vs Scientist
Lowercase a Python String
Stratified Random Sampling
Linux Environment Variables
Which Programming Language?

Our Charity