Python

* What can python do ?

You’re probably wondering what people are building with Python in the real world. So first, let’s take a quick look at how some of the big tech companies are using the language.

Google is a company that has used Python from the start, and it’s gained a place as one of the tech giant’s main server-side languages. Guido van Rossum, Python’s Benevolent Dictator for Life, even worked there for several years, overseeing the language’s development.

Instagram likes Python for its simplicity.

Spotify puts the language to use in its data analysis and back-end services. According to their team, Python’s ease of use leads to a lightning-fast development pipeline. Spotify performs a ton of analyses to give recommendations to their users, so they need something that’s simple but also works well. Python to the rescue!

From web development to data science, machine learning , and more, Python’s real-world applications are limitless. Here are some projects that will assist you in finally putting your Python skills to good use.

#1: Automate the Boring Stuff

This is a resource on “practical programming for total beginners.” Like the title says, this book will teach you how to automate tedious tasks such as updating spreadsheets or renaming files on your computer. It’s the perfect starting point for anyone who’s mastered the basics of Python.

You’ll get a chance to practice what you’ve learned so far by creating dictionaries, scraping the web, working with files , and creating objects and classes. The hands-on applications that you come across in this book will provide you with real-world results that you can see immediately.

#2: Stay on Top of Bitcoin

Everyone seems to be talking about Bitcoin these days. Ever since topping out at a price of almost $20,000 in December 2017, the cryptocurrency has been on the minds of millions. Its price continues to fluctuate, but many would consider it a worthwhile investment.

If you’re looking to cash in on the virtual gold rush and just need to know when to make your move, then you’ll need to stay on top of Bitcoin’s prices. This tutorial can teach you how to use your Python skills to build a Bitcoin price notification service.

The foundation of this project is the creation of IFTTT (“if this, then that”) applets. You’ll learn how to use the [requests](http://docs.python-requests.org/en/master/) library to send HTTP requests and how to use a webhook to connect your app to external services.

This is the perfect starter project for a beginner Pythonista with an interest in crypto. The service you build with this tutorial can be extended to other currencies as well, so don’t worry—Ethereum is fair game, too.

#3: Create a Calculator

This simple project is a solid gateway into GUI programming. Building back-end services is one important part of deployment, but there may be a front-end that needs to be taken into account. Creating applications that users can easily interact with is paramount.

If you’re interested in UX and UI design, then take a look at this tutorial You’ll be working with the tkinter module, the standard graphical user interface package that comes traditionally bundled with Python.

The tkinter module is a wrapper around Tcl/Tk, a combination of the Tcl scripting language and a GUI framework extension, Tk. If you have Python installed, then you should already have the tkinter framework ready to go as well. A simple call will get you started:

#4: Mine Twitter Data

Thanks to the Internet—and, increasingly, the Internet of Things—we now have access to hordes of data that weren’t available even a decade ago. Analytics is a huge part of any field that works with data. What are people talking about? What patterns can we see in their behavior?

Twitter is a great place to get answers to some of these questions. If you’re interested in data analysis, then a Twitter data mining project is a great way to use your Python skills to answer questions about the world around you.

#XX-scale web applications.

#6: Build a Blockchain

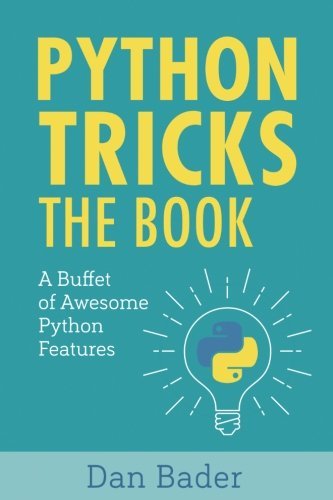
While the blockchain was initially developed as a financial technology, it’s spreading to a variety of other industries. Blockchains can be used for almost any kind of transaction: from real estate dealings to medical record transfers.

You can get a better understanding of how they work by building one yourself. [Hackernoon’s tutorial](https://hackernoon.com/learn-blockchains-by-building-one-117428612f46) will assist you in implementing a blockchain from scratch. At the end of this project, you’ll have gained an in-depth understanding of how this transactional technology works.

You’ll be working with HTTP clients and the [requests](http://docs.python-requests.org/en/master/) library. Once you install the [Flask](https://realpython.com/tutorials/flask/) web framework, you’ll be able to use HTTP requests to communicate with your blockchain over the Internet.

Remember, blockchain isn’t just for crypto enthusiasts. Once you’ve built one for yourself, see if you can’t find a creative way to implement the technology in your field of interest.

7: Get Challenged

[](https://files.realpython.com/media/python-tricks-book-cover.5b0f5df06d94.jpg)

If you’re not sure about taking the plunge with some of the larger projects listed above, but the smaller ones don’t interest you either, then you might be wondering what else there is. How on earth can you find something that excites you?

Coding challenges can help you practice your Python skills and gain a surface-level understanding of all the different things you can do with Python.

To put it simply: you’re presented with a problem, and you have to find a solution that uses Python.

You’ll get a chance to develop implementations that make sense to you, but you’ll also have the opportunity to dive deep into the Python language by way of hints. These give you an idea of which modules you should be importing to help you solve the challenge.

Coding challenges are a great way to learn breadth-first about as many libraries, methods, and frameworks as possible. You’re guaranteed to find something that you’ll want to explore more on your own time. You might even come back to this list and find that something you used in one of your challenges has sparked a new interest for you!

* WHY PYTHON :-

## Why to Learn Python?

**Python** is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

**Python** is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Web Development Domain. I will list down some of the key advantages of learning Python:

* **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
* **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
* **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
* **Python is a Beginner's Language** − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

## Characteristics of Python

Following are important characteristics of **Python Programming** −

* It supports functional and structured programming methods as well as OOP.
* It can be used as a scripting language or can be compiled to byte-code for building large applications.
* It provides very high-level dynamic data types and supports dynamic type checking.
* It supports automatic garbage collection.
* It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.
  + PYTHON SYNTAX COMPARED TO OTHER PROGRAMMING LANGYAGES :-

The Python language has many similarities to Perl, C, and Java. However, there are some definite differences between the languages.

First Python Program

Let us execute programs in different modes of programming.

Interactive Mode Programming

Invoking the interpreter without passing a script file as a parameter brings up the following prompt −

$ python

Python 2.4.3 (#1, Nov 11 2010, 13:34:43)

[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>>

Type the following text at the Python prompt and press the Enter −

>>> print "Hello, Python!"

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!");**. However in Python version 2.4.3, this produces the following result −

Hello, Python!

Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have extension **.py**. Type the following source code in a test.py file −

print "Hello, Python!"

We assume that you have Python interpreter set in PATH variable. Now, try to run this program as follows −

$ python test.py

This produces the following result −

Hello, Python!

Let us try another way to execute a Python script. Here is the modified test.py file −

#!/usr/bin/python

print "Hello, Python!"

We assume that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows −

$ chmod +x test.py # This is to make file executable

$./test.py

This produces the following result −

Hello, Python!

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

* Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
* Starting an identifier with a single leading underscore indicates that the identifier is private.
* Starting an identifier with two leading underscores indicates a strongly private identifier.
* If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

|  |  |  |
| --- | --- | --- |
| and | exec | not |
| assert | finally | or |
| break | for | pass |
| class | from | print |
| continue | global | raise |
| def | if | return |
| del | import | try |
| elif | in | while |
| else | is | with |
| except | lambda | yield |

Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example −

if True:

print "True"

else:

print "False"

However, the following block generates an error −

if True:

print "Answer"

print "True"

else:

print "Answer"

print "False"

Thus, in Python all the continuous lines indented with same number of spaces would form a block. The following example has various statement blocks −

**Note** − Do not try to understand the logic at this point of time. Just make sure you understood various blocks even if they are without braces.

#!/usr/bin/python

import sys

try:

# open file stream

file = open(file\_name, "w")

except IOError:

print "There was an error writing to", file\_name

sys.exit()

print "Enter '", file\_finish,

print "' When finished"

while file\_text != file\_finish:

file\_text = raw\_input("Enter text: ")

if file\_text == file\_finish:

# close the file

file.close

break

file.write(file\_text)

file.write("\n")

file.close()

file\_name = raw\_input("Enter filename: ")

if len(file\_name) == 0:

print "Next time please enter something"

sys.exit()

try:

file = open(file\_name, "r")

except IOError:

print "There was an error reading file"

sys.exit()

file\_text = file.read()

file.close()

print file\_text

Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example −

total = item\_one + \

item\_two + \

item\_three

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example −

days = ['Monday', 'Tuesday', 'Wednesday',

'Thursday', 'Friday']

Quotation in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

word = 'word'

sentence = "This is a sentence."

paragraph = """This is a paragraph. It is

made up of multiple lines and sentences."""

Comments in Python

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

#!/usr/bin/python

# First comment

print "Hello, Python!" # second comment

This produces the following result −

Hello, Python!

You can type a comment on the same line after a statement or expression −

name = "Madisetti" # This is again comment

You can comment multiple lines as follows −

# This is a comment.

# This is a comment, too.

# This is a comment, too.

# I said that already.

Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments:

'''

This is a multiline

comment.

'''

Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Waiting for the User

The following line of the program displays the prompt, the statement saying “Press the enter key to exit”, and waits for the user to take action −

#!/usr/bin/python

raw\_input("\n\nPress the enter key to exit.")

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Multiple Statements on a Single Line

The semicolon ( ; ) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon −

import sys; x = 'foo'; sys.stdout.write(x + '\n')

Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon ( : ) and are followed by one or more lines which make up the suite. For example −

if expression :

suite

elif expression :

suite

else :

suite

Command Line Arguments

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with -h −

$ python -h

usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...

Options and arguments (and corresponding environment variables):

-c cmd : program passed in as string (terminates option list)

-d : debug output from parser (also PYTHONDEBUG=x)

-E : ignore environment variables (such as PYTHONPATH)

-h : print this help message and exit

[ etc. ]

You can also program your script in such a way that it should accept various options. [Command Line Arguments](https://www.tutorialspoint.com/python/python_command_line_arguments.htm) is an advanced topic and should be studied a bit later once you have gone through rest of the Python concepts.

Python installation:-

You may want to print these instructions before proceeding, so that you can refer to them while downloading and installing Python. Or, just keep this document in your browser. You should read each step completely before performing the action that it describes.

This document shows downloading and installing Python 3.7.4 on Windows 10 in Summer 2019. **You should download and install the latest version of Python.** The current latest (as of Winter 2020) is Python 3.8.1.

Remember that you must install Java, Python, and Eclipse as all 64-bit applications.

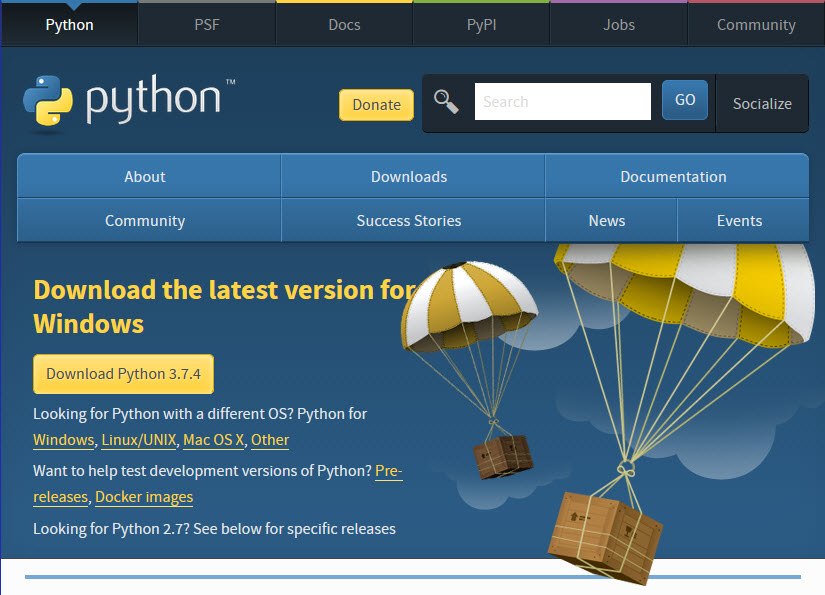
## Python: Version 3.7.4

The Python download requires about 25 Mb of disk space; keep it on your machine, in case you need to re-install Python. When installed, Python requires about an additional 90 Mb of disk space.

### Downloading

1. Click [Python Download](https://www.python.org/downloads/).

The following page will appear in your browser.

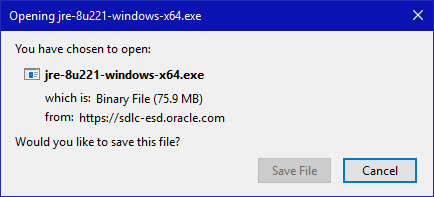


1. Click the **Windows** link (two lines below the **Download Python 3.7.4** button). The following page will appear in your browser.



1. Click on the **Download Windows x86-64 executable installer** link under the top-left **Stable Releases**.

The following pop-up window titled **Opening python-3.74-amd64.exe** will appear.



Click the **Save File** button.

The file named **python-3.7.4-amd64.exe** should start downloading into your standard download folder. This file is about 30 Mb so it might take a while to download fully if you are on a slow internet connection (it took me about 10 seconds over a cable modem).

The file should appear as

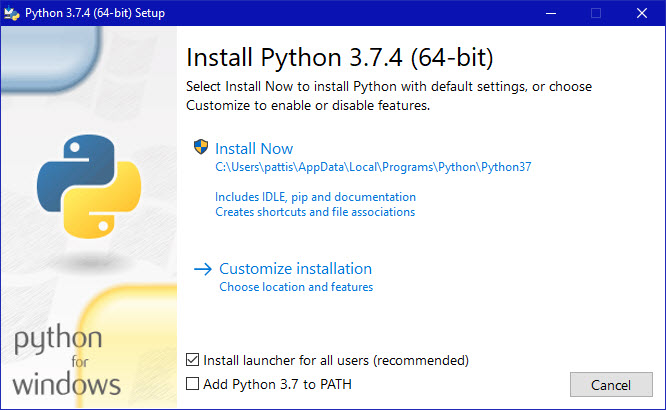


1. Move this file to a more permanent location, so that you can install Python (and reinstall it easily later, if necessary).
2. Feel free to explore this webpage further; if you want to just continue the installation, you can terminate the tab browsing this webpage.
3. Start the **Installing** instructions directly below.

### Installing

1. Double-click the icon labeling the file **python-3.7.4-amd64.exe**.

A **Python 3.7.4 (64-bit) Setup** pop-up window will appear.



Ensure that the **Install launcher for all users (recommended)** and the **Add Python 3.7 to PATH** checkboxes at the bottom are checked.

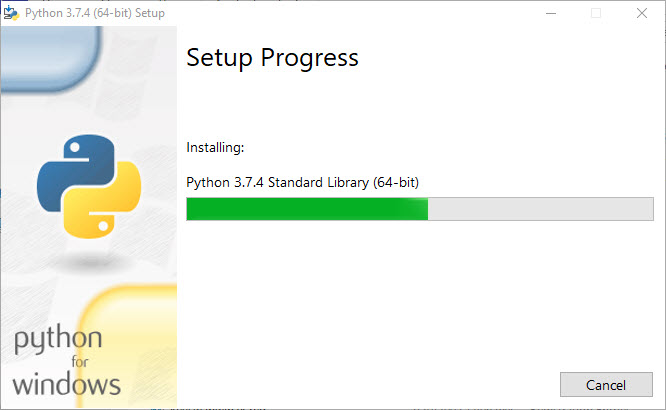
If the Python Installer finds an earlier version of Python installed on your computer, the **Install Now** message may instead appear as **Upgrade Now** (and the checkboxes will not appear).

1. Highlight the **Install Now** (or **Upgrade Now**) message, and then click it.

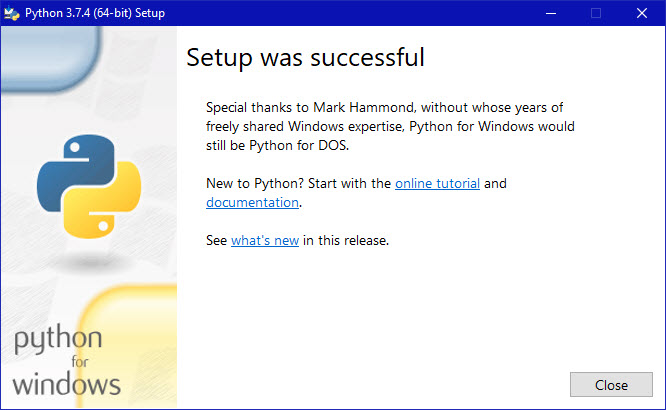
When run, a **User Account Control** pop-up window may appear on your screen. I could not capture its image, but it asks, **Do you want to allow this app to make changes to your device**.

1. Click the **Yes** button.

A new **Python 3.7.4 (64-bit) Setup** pop-up window will appear with a **Setup Progress** message and a progress bar.



During installation, it will show the various components it is installing and move the progress bar towards completion. Soon, a new **Python 3.7.4 (64-bit) Setup** pop-up window will appear with a **Setup was successfuly** message.



1. Click the **Close** button.

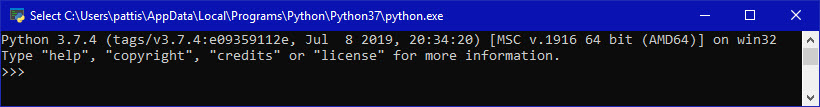
Python should now be installed.

### Verifying

To try to verify installation,

1. Navigate to the directory **C:\Users\Pattis\AppData\Local\Programs\Python\Python37** (or to whatever directory Python was installed: see the pop-up window for Installing step 3).
2. Double-click the icon/file **python.exe**.

The following pop-up window will appear.



A pop-up window with the title **C:\Users\Pattis\AppData\Local\Programs\Python\Python37\python.exe** appears, and inside the window; on the first line is the text **Python 3.7.4 ...** (notice that it should also say 64 bit). Inside the window, at the bottom left, is the prompt **>>>**: type **exit()** to this prompt and press **enter** to terminate Python.

You should keep the file **python-3.7.4.exe** somewhere on your computer in case you need to reinstall Python (not likely necessary).

You may now follow the instructions to download and install Java (you should have already installed Java, but if you haven't, it is OK to do so now, so long as you install both Python and Java before you install Eclipse), and then follows the instruction to download and install the Eclipse IDE. Note: you need to download/install Java even if you are using Eclipse only for Python)

Module 2:-beginning python Basics :-

* The print statements:-

A lot of you, while reading this tutorial, might think that there is nothing undiscovered about a simple python  Print function since you all would have started learning Python with the evergreen example of printing Hello, World!. It's also true that Python or, for that matter, any programming language, the Print function is the most basic and the baby step that you take while learning a particular language. However, while learning a programming language, you sometimes tend to focus on more advanced topics and many times forget the intricacies or capabilities of a simple function that you might almost always use.

The focus of today's tutorial will be entirely on Python's Print function; you will learn about the most underestimated function.

Let's begin this tutorial by printing the Hello, World! example.

print("Hello, World!")

Hello, World!

Unlike Python2, which did not require you to put a parenthesis, in Python3, parenthesis is a must else it will raise a syntax error as shown below.

print "Hello, World!"

File "<ipython-input-6-a1fcabcd869c>", line 1

print "Hello, World!"

^

SyntaxError: Missing parentheses in call to 'print'. Did you mean print("Hello, World!")?

From the above output, you can conclude that in Python3, print() is not a statement but a function.

Having said that, let's check the type/class of the print() function.

type(print)

builtin\_function\_or\_method

It returns builtin\_function\_or\_method, which means it is a predefined or a builtin Python Function.

Now let's say you want to add a new line or a vertical space between the below two outputs, to achieve this you can simply call the print() function without passing any argument in it.

print("Hello, World!");print("Hello, World!")

Hello, World!

Hello, World!

print("Hello, World!")

print()

print("Hello, World!")

Hello, World!

Hello, World!

Let's look at the syntax of the print() function. print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False) As you know by now, the print function Prints the values to a stream, or to sys.stdout by default. The sys.stdout or system standard output means the print function will print the output on the screen, and it can even be changed to stdin or stderr.

#### Optional keyword arguments:

* sep: It can be a string which you would like to insert between values, defaults to space.

Let's put a list of words inside the print function and separate them with a new line. Remember, by default; the separator adds a blank space between each word.

print('datacamp','tutorial','on','python','print','function')

datacamp tutorial on python print function

print('datacamp','tutorial','on','python','print','function',sep='\n') #`\n` will put each word in a new line

datacamp

tutorial

on

python

print

function

Similarly, you can separate them with a comma, or add two \n which will add an empty line in between or even add a plus (+)(+) sign.

print('datacamp','tutorial','on','python','print','function',sep=',')

datacamp,tutorial,on,python,print,function

print('datacamp','tutorial','on','python','print','function',sep='\n\n')

datacamp

tutorial

on

python

print

function

print('datacamp','tutorial','on','python','print','function',sep=',+')

datacamp,+tutorial,+on,+python,+print,+function

Before you jump on to the next argument, i.e., end, remember that you can also pass in a variable to the print function. To understand this, let's take an example by defining a list of integers and pass it to the print function. You would expect it to print the list of integers.

int\_list = [1,2,3,4,5,6]

print(int\_list)

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

* end: It is a string appended after the last value, defaults to a newline. It allows the programmer to define a custom ending character for each print call other than the default newline or \n.

Let's say you have two strings and you want to join them together with space, all you would need to do is in the print function of the first string str1 add end argument with quotes, and you can expect the two strings to be joined by a space.

str1 = 'datacamp tutorial on'

str2 = 'python print function'

print(str1)

print(str2)

datacamp tutorial on

python print function

print(str1,end=' ')

print(str2)

datacamp tutorial on python print function

Let's take another example where you have a function whose job is to print all values within a list on the same line. You could achieve this with the end argument, as shown below:

def value(items):

for item in items:

print(item, end=' ')

value([1,2,3,4])

1 2 3 4

* file: A file-like object (stream); defaults to the current sys.stdout. Here you can mention the file in which you would like to write or append the output of the print function.

By using the file argument, you can store the output of the print function to a file in various formats like .csv, .txt, etc. Let's understand this by taking an example wherein you iterate over each element of the list. It is saved in a text file. To accomplish this, first, you will open a text file in an append state in which you will be saving the output of the print statement. Next, you will define the function whose output will be appended in the text file.

file = open('print.txt','a+')

def value(items):

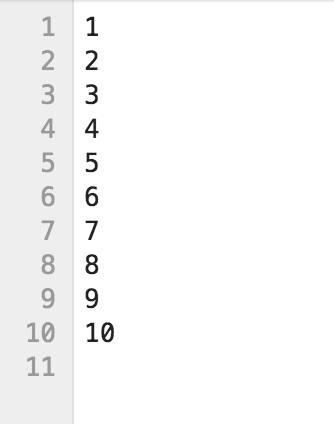
for item in items:

print(item, file=file)

file.close() # close the file ones the function execution completes.

value([1,2,3,4,5,6,7,8,9,10])

Once the above function execution completes, you should see a file named print.txt in your current directory. Let's open it and see its contents.



From the above output, it is clear that you can even save the output to files and not just print the output on the terminal or within the notebook.

* flush: It determines whether to forcibly flush the stream. By default, it is set to False.

Typically, output to a file or the console is buffered, with text output at least until you print a newline ([Source](https://stackoverflow.com/questions/15608229/what-does-prints-flush-do)). Buffer means that the output is stored in some kind of a register where the output resides before it is ready to be stored or the file is finally closed. The job of the flush is to make sure that the output, which is being buffered, goes to the destination safely.

import time

print('Please enter your email-id : ', end=' ')

#print('Please enter your email-id : ', end=' ', flush=True) #run this to see the difference and comment out the above line of code.

time.sleep(5)

Please enter your email-id :

If you run the above lines of code in the terminal, you will notice that the prompt string does not show up until the sleep timer ends and the program exits ([Source](https://stackoverflow.com/questions/15608229/what-does-prints-flush-do)). However, if you add flush=True in the print function, you will see the prompt, and then you will have to wait for 5 seconds for the program to finish.

Somehow the Jupyter Notebook or Jupyter Lab takes care of it and shows the prompt before the 5 seconds timer, so if you want to test this functionality of print, make sure to run it in command line terminal and not on the jupyter notebook.

Let's now see how you can make use of the print function to take input from the user in the jupyter notebook. For this, you will use python's builtin function input().

tutorial\_topic = input()

print("The topic of today's tutorial is: ", end='')

print(tutorial\_topic)

Print Function

The topic of today's tutorial is: Print Function

As soon as you run the above cell, you should see an output as shown below:



[Here](https://www.datacamp.com/community/tutorials/python-print-function?utm_source=adwords_ppc&utm_campaignid=1455363063&utm_adgroupid=65083631748&utm_device=c&utm_keyword=&utm_matchtype=b&utm_network=g&utm_adpostion=&utm_creative=332602034358&utm_targetid=dsa-429603003980&utm_loc_interest_ms=&utm_loc_physical_ms=1007740&gclid=CjwKCAjw19z6BRAYEiwAmo64LeEdTO8g66Autv4RQNNstjz9ual9ZHJxdxjVX-irlioJ6wYy8lBspBoCmy0QAvD_BwE#inp) you defined an optional named argument end that you learned before, which concatenates the static statement inside print statement with the user input.

Let's see some cooler ways of printing variable values inside the print function.

* To display a variable's value along with a predefined string, all you need to do is add a comma in between the two. Here the position of the predefined string and the variable does not matter.

a = 2

b = "Datacamp"

print(a,"is an integer while",b,"is a string.")

2 is an integer while Datacamp is a string.

* You can make use of the format argument wherein you can pass as many variables as you want to print. When you pass the variables in the format function, you need to specify the index numbers (order in which they are placed inside the format argument) in the predefined string. So this print function will now act as a template for you.

Another thing to remember is that here the index numbers are represented with a curly bracket {} representing a placeholder.

Let's understand it with the below example:

a = 2

b = "Datacamp"

print("{0} is an integer while {1} is a string.".format(a,b))

2 is an integer while Datacamp is a string.

If you put the index number of the second variable at both places, as expected, it will print the same values, in this case, b variable for both.

print("{1} is an integer while {1} is a string.".format(a,b))

Datacamp is an integer while Datacamp is a string.

* Similar to a format argument where your print function acts as a template, you have a percentage (%) sign that you can use to print the values of the variables.

Like format argument, this also has a concept of placeholders. However, unlike the format function where you pass in just the index numbers, in this, you also need to specify the datatype the placeholder should expect.

* + %d is used as a placeholder for numeric or decimal values.
  + %s is used as a placeholder for strings.

a = 2

b = "Datacamp"

print("%d is an integer while %s is a string."%(a,b))

2 is an integer while Datacamp is a string.

Let's see what happens if you specify %s for the variable a which is an integer.

print("%s is an integer while %s is a string."%(a,b))

2 is an integer while Datacamp is a string.

As you can see from the above example, it still works. The reason is that the Print function implicitly performed typecasting and converted the integer to a string. However, the opposite is not true. It will fail to convert a string to an integer and will result in a TypeError.

Let's find out.

print("%d is an integer while %d is a string."%(a,b))

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-121-68c55041ecfe> in <module>

----> 1 print("%d is an integer while %d is a string."%(a,b))

TypeError: %d format: a number is required, not str

## Conclusion

Congratulations on finishing the tutorial.

This tutorial was a good starting point for beginners who aspire to become proficient in Python. You might want to play around with the Print function a bit more and explore a few more functionalities of it that might have been missed out in this tutorial.

# **How to use comments in Python**

When working with any programming language, you include comments in the code to notate your work. This details what certain parts of the code are for, and lets other developers – you included – know what you were up to when you wrote the code. This is a necessary practice, and good developers make heavy use of the comment system. Without it, things can get real confusing, real fast.

### How to Write Comments in Python

In Python, there are two ways to annotate your code.

The first is to include comments that detail or indicate what a section of code – or snippet – does.

The second makes use of multi-line comments or paragraphs that serve as documentation for others reading your code.

Think of the first type as a comment for yourself, and the second as a comment for others. There is not right or wrong way to add a comment, however. You can do whatever feels comfortable.

Single-line comments are created simply by beginning a line with the hash (#) character, and they are automatically terminated by the end of line.

For example:

#This would be a comment in Python

Comments that span multiple lines – used to explain things in more detail – are created by adding a delimiter (“””) on each end of the comment.

"""

This would be a multiline comment

in Python that spans several lines and

describes your code, your day, or anything you want it to

"""

* Basic data structures and data types :-

## Basic Data Types

Python is a dynamically typed language, which means that the Python interpreter infers the type of an object at runtime. In comparison, compiled languages like C are generally statically typed. In these cases, the type of an object has to be attached to the object before compile time.[[18](https://www.oreilly.com/library/view/python-for-finance/9781491945360/ch04.html#ftn.id1154611)]

### Integers

One of the most fundamental data types is the integer, or int:

In [1]: a = 10

type(a)

Out[1]: int

The built-in function type provides type information for all objects with standard and built-in types as well as for newly created classes and objects. In the latter case, the information provided depends on the description the programmer has stored with the class. There is a saying that “everything in Python is an object.” This means, for example, that even simple objects like the int object we just defined have built-in methods. For example, you can get the number of bits needed to represent the int object in-memory by calling the method bit\_length:

In [2]: a.bit\_length()

Out[2]: 4

You will see that the number of bits needed increases the higher the integer value is that we assign to the object:

In [3]: a = 100000

a.bit\_length()

Out[3]: 17

In general, there are so many different methods that it is hard to memorize all methods of all classes and objects. Advanced Python environments, like IPython, provide tab completion capabilities that show all methods attached to an object. You simply type the object name followed by a dot (e.g., a.) and then press the Tab key, e.g., a.*tab*. This then provides a collection of methods you can call on the object. Alternatively, the Python built-in function dir gives a complete list of attributes and methods of any object.

A specialty of Python is that integers can be arbitrarily large. Consider, for example, the googol number 10100. Python has no problem with such large numbers, which are technically long objects:

In [4]: googol = 10 \*\* 100

googol

Out[4]: 100000000000000000000000000000000000000000000000000000000000000000000000

00000000000000000000000000000L

In [5]: googol.bit\_length()

Out[5]: 333

### LARGE INTEGERS

Python integers can be arbitrarily large. The interpreter simply uses as many bits/bytes as needed to represent the numbers.

It is important to note that mathematical operations on int objects return int objects. This can sometimes lead to confusion and/or hard-to-detect errors in mathematical routines. The following expression yields the expected result:

In [6]: 1 + 4

Out[6]: 5

However, the next case may return a somewhat surprising result:

In [7]: 1 / 4

Out[7]: 0

In [8]: type(1 / 4)

Out[8]: int

### Floats

For the last expression to return the generally desired result of 0.25, we must operate on float objects, which brings us naturally to the next basic data type. Adding a dot to an integer value, like in 1. or 1.0, causes Python to interpret the object as a float. Expressions involving a float also return a float object in general:[[19](https://www.oreilly.com/library/view/python-for-finance/9781491945360/ch04.html#ftn.id972374)]

In [9]: 1. / 4

Out[9]: 0.25

In [10]: type (1. / 4)

Out[10]: float

A float is a bit more involved in that the computerized representation of rational or real numbers is in general not exact and depends on the specific technical approach taken. To illustrate what this implies, let us define another float object:

In [11]: b = 0.35

type(b)

Out[11]: float

float objects like this one are always represented internally up to a certain degree of accuracy only. This becomes evident when adding 0.1 to b:

In [12]: b + 0.1

Out[12]: 0.44999999999999996

The reason for this is that floats are internally represented in binary format; that is, a decimal number 0 < n < 1 is represented by a series of the form . For certain floating-point numbers the binary representation might involve a large number of elements or might even be an infinite series. However, given a fixed number of bits used to represent such a number—i.e., a fixed number of terms in the representation series—inaccuracies are the consequence. Other numbers can be represented perfectly and are therefore stored exactly even with a finite number of bits available. Consider the following example:

In [13]: c = 0.5

c.as\_integer\_ratio()

Out[13]: (1, 2)

One half, i.e., 0.5, is stored exactly because it has an exact (finite) binary representation as . However, for b = 0.35 we get something different than the expected rational number :

In [14]: b.as\_integer\_ratio()

Out[14]: (3152519739159347, 9007199254740992)

The precision is dependent on the number of bits used to represent the number. In general, all platforms that Python runs on use the IEEE 754 double-precision standard (i.e., 64 bits), for internal representation.[[20](https://www.oreilly.com/library/view/python-for-finance/9781491945360/ch04.html#ftn.id965021)] This translates into a 15-digit relative accuracy.

Since this topic is of high importance for several application areas in finance, it is sometimes necessary to ensure the exact, or at least best possible, representation of numbers. For example, the issue can be of importance when summing over a large set of numbers. In such a situation, a certain kind and/or magnitude of representation error might, in aggregate, lead to significant deviations from a benchmark value.

The module decimal provides an arbitrary-precision object for floating-point numbers and several options to address precision issues when working with such numbers:

In [15]: **import** **decimal**

**from** **decimal** **import** Decimal

In [16]: decimal.getcontext()

Out[16]: Context(prec=28, rounding=ROUND\_HALF\_EVEN, Emin=-999999999, Emax=999999

999, capitals=1, flags=[], traps=[Overflow, InvalidOperation, DivisionB

yZero])

In [17]: d = Decimal(1) / Decimal (11)

d

Out[17]: Decimal('0.09090909090909090909090909091')

You can change the precision of the representation by changing the respective attribute value of the Context object:

In [18]: decimal.getcontext().prec = 4 *# lower precision than default*

In [19]: e = Decimal(1) / Decimal (11)

e

Out[19]: Decimal('0.09091')

In [20]: decimal.getcontext().prec = 50 *# higher precision than default*

In [21]: f = Decimal(1) / Decimal (11)

f

Out[21]: Decimal('0.090909090909090909090909090909090909090909090909091')

If needed, the precision can in this way be adjusted to the exact problem at hand and one can operate with floating-point objects that exhibit different degrees of accuracy:

In [22]: g = d + e + f

g

Out[22]: Decimal('0.27272818181818181818181818181909090909090909090909')

### ARBITRARY-PRECISION FLOATS

The module decimal provides an arbitrary-precision floating-point number object. In finance, it might sometimes be necessary to ensure high precision and to go beyond the 64-bit double-precision standard.

### Strings

Now that we can represent natural and floating-point numbers, we turn to text. The basic data type to represent text in Python is the string. The string object has a number of really helpful built-in methods. In fact, Python is generally considered to be a good choice when it comes to working with text files of any kind and any size. A string object is generally defined by single or double quotation marks or by converting another object using the str function (i.e., using the object’s standard or user-defined string representation):

In [23]: t = 'this is a string object'

With regard to the built-in methods, you can, for example, capitalize the first word in this object:

In [24]: t.capitalize()

Out[24]: 'This is a string object'

Or you can split it into its single-word components to get a list object of all the words (more on list objects later):

In [25]: t.split()

Out[25]: ['this', 'is', 'a', 'string', 'object']

You can also search for a word and get the position (i.e., index value) of the first letter of the word back in a successful case:

In [26]: t.find('string')

Out[26]: 10

If the word is not in the string object, the method returns -1:

In [27]: t.find('Python')

Out[27]: -1

Replacing characters in a string is a typical task that is easily accomplished with the replace method:

In [28]: t.replace(' ', '|')

Out[28]: 'this|is|a|string|object'

The stripping of strings—i.e., deletion of certain leading/lagging characters—is also often necessary:

In [29]: 'http://www.python.org'.strip('htp:/')

Out[29]: 'www.python.org'

[Table 4-1](https://www.oreilly.com/library/view/python-for-finance/9781491945360/ch04.html#string_methods) lists a number of helpful methods of the string object.

*Table 4-1. Selected string methods*

| **Method** | **Arguments** | **Returns/result** |
| --- | --- | --- |
| capitalize | () | Copy of the string with first letter capitalized |
| count | (*sub*[, *start*[, *end*]]) | Count of the number of occurrences of substring |
| decode | ([*encoding*[, *errors*]]) | Decoded version of the string, using *encoding* (e.g., UTF-8) |
| encode | ([*encoding*[, *errors*]]) | Encoded version of the string |
| find | (*sub*[, *start*[, *end*]]) | (Lowest) index where substring is found |
| join | (*seq*) | Concatenation of strings in sequence *seq* |
| replace | (*old*, *new*[, *count*]) | Replaces *old* by *new* the first *count* times |
| split | ([*sep*[, *maxsplit*]]) | List of words in string with *sep* as separator |
| splitlines | ([*keepends*]) | Separated lines with line ends/breaks if *keepends* is *True* |
| strip | (*chars*) | Copy of string with leading/lagging characters in *chars* removed |
| upper | () | Copy with all letters capitalized |

A powerful tool when working with string objects is regular expressions. Python provides such functionality in the module re:

In [30]: **import** **re**

Suppose you are faced with a large text file, such as a comma-separated value (CSV) file, which contains certain time series and respective date-time information. More often than not, the date-time information is delivered in a format that Python cannot interpret directly. However, the date-time information can generally be described by a regular expression. Consider the following string object, containing three date-time elements, three integers, and three strings. Note that triple quotation marks allow the definition of strings over multiple rows:

In [31]: series = """

'01/18/2014 13:00:00', 100, '1st';

'01/18/2014 13:30:00', 110, '2nd';

'01/18/2014 14:00:00', 120, '3rd'

"""

The following regular expression describes the format of the date-time information provided in the string object:[[21](https://www.oreilly.com/library/view/python-for-finance/9781491945360/ch04.html#ftn.id1322653)]

In [32]: dt = re.compile("'[0-9/:\s]+'") *# datetime*

Equipped with this regular expression, we can go on and find all the date-time elements. In general, applying regular expressions to string objects also leads to performance improvements for typical parsing tasks:

In [33]: result = dt.findall(series)

result

Out[33]: ["'01/18/2014 13:00:00'", "'01/18/2014 13:30:00'", "'01/18/2014 14:00:0

0'"]

### REGULAR EXPRESSIONS

When parsing string objects, consider using regular expressions, which can bring both convenience and performance to such operations.

The resulting string objects can then be parsed to generate Python datetime objects (cf. [Appendix C](https://www.oreilly.com/library/view/python-for-finance/9781491945360/apc.html) for an overview of handling date and time data with Python). To parse the string objects containing the date-time information, we need to provide information of how to parse—again as a string object:

In [34]: **from** **datetime** **import** datetime

pydt = datetime.strptime(result[0].replace("'", ""),

'%m/%d/%Y %H:%M:%S')

pydt

Out[34]: datetime.datetime(2014, 1, 18, 13, 0)

In [35]: **print** pydt

Out[35]: 2014-01-18 13:00:00

In [36]: **print** type(pydt)

Out[36]: <type 'datetime.datetime'>

Later chapters provide more information on date-time data, the handling of such data, and datetime objects and their methods. This is just meant to be a teaser for this important topic in finance.

## Basic Data Structures

As a general rule, data structures are objects that contain a possibly large number of other objects. Among those that Python provides as built-in structures are:

tuple

A collection of arbitrary objects; only a few methods available

list

A collection of arbitrary objects; many methods available

dict

A key-value store object

set

An unordered collection object for other unique objects

### Tuples

A tuple is an advanced data structure, yet it’s still quite simple and limited in its applications. It is defined by providing objects in parentheses:

In [37]: t = (1, 2.5, 'data')

type(t)

Out[37]: tuple

You can even drop the parentheses and provide multiple objects separated by commas:

In [38]: t = 1, 2.5, 'data'

type(t)

Out[38]: tuple

Like almost all data structures in Python the tuple has a built-in index, with the help of which you can retrieve single or multiple elements of the tuple. It is important to remember that Python uses zero-based numbering, such that the third element of a tuple is at index position 2:

In [39]: t[2]

Out[39]: 'data'

In [40]: type(t[2])

Out[40]: str

### ZERO-BASED NUMBERING

In contrast to some other programming languages like Matlab, Python uses zero-based numbering schemes. For example, the first element of a tuple object has index value 0.

There are only two special methods that this object type provides: count and index. The first counts the number of occurrences of a certain object and the second gives the index value of the first appearance of it:

In [41]: t.count('data')

Out[41]: 1

In [42]: t.index(1)

Out[42]: 0

tuple objects are not very flexible since, once defined, they cannot be changed easily.

### Lists

Objects of type list are much more flexible and powerful in comparison to tuple objects. From a finance point of view, you can achieve a lot working only with list objects, such as storing stock price quotes and appending new data. A list object is defined through brackets and the basic capabilities and behavior are similar to those of tuple objects:

In [43]: l = [1, 2.5, 'data']

l[2]

Out[43]: 'data'

list objects can also be defined or converted by using the function list. The following code generates a new list object by converting the tuple object from the previous example:

In [44]: l = list(t)

l

Out[44]: [1, 2.5, 'data']

In [45]: type(l)

Out[45]: list

In addition to the characteristics of tuple objects, list objects are also expandable and reducible via different methods. In other words, whereas string and tuple objects are immutable sequence objects (with indexes) that cannot be changed once created, list objects are mutable and can be changed via different operations. You can append list objects to an existing list object, and more:

In [46]: l.append([4, 3]) *# append list at the end*

l

Out[46]: [1, 2.5, 'data', [4, 3]]

In [47]: l.extend([1.0, 1.5, 2.0]) *# append elements of list*

l

Out[47]: [1, 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]

In [48]: l.insert(1, 'insert') *# insert object before index position*

l

Out[48]: [1, 'insert', 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]

In [49]: l.remove('data') *# remove first occurrence of object*

l

Out[49]: [1, 'insert', 2.5, [4, 3], 1.0, 1.5, 2.0]

In [50]: p = l.pop(3) *# removes and returns object at index*

**print** l, p

Out[50]: [1, 'insert', 2.5, 1.0, 1.5, 2.0] [4, 3]

Slicing is also easily accomplished. Here, slicing refers to an operation that breaks down a data set into smaller parts (of interest):

In [51]: l[2:5] *# 3rd to 5th elements*

Out[51]: [2.5, 1.0, 1.5]

* String operations in python:-

# **Basic String Operations**

Strings are bits of text. They can be defined as anything between quotes:

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

astring = "Hello world!"

astring2 = 'Hello world!'

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

Run

[Powered by DataCamp](https://www.datacamp.com/)

As you can see, the first thing you learned was printing a simple sentence. This sentence was stored by Python as a string. However, instead of immediately printing strings out, we will explore the various things you can do to them. You can also use single quotes to assign a string. However, you will face problems if the value to be assigned itself contains single quotes.For example to assign the string in these bracket(single quotes are ' ') you need to use double quotes only like this

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

3

4

astring = "Hello world!"

print("single quotes are ' '")

print(len(astring))

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

Run

[Powered by DataCamp](https://www.datacamp.com/)

That prints out 12, because "Hello world!" is 12 characters long, including punctuation and spaces.

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

astring = "Hello world!"

print(astring.index("o"))

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

Run

[Powered by DataCamp](https://www.datacamp.com/)

That prints out 4, because the location of the first occurrence of the letter "o" is 4 characters away from the first character. Notice how there are actually two o's in the phrase - this method only recognizes the first.

But why didn't it print out 5? Isn't "o" the fifth character in the string? To make things more simple, Python (and most other programming languages) start things at 0 instead of 1. So the index of "o" is 4.

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

astring = "Hello world!"

print(astring.count("l"))

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

Run

[Powered by DataCamp](https://www.datacamp.com/)

For those of you using silly fonts, that is a lowercase L, not a number one. This counts the number of l's in the string. Therefore, it should print 3.

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

astring = "Hello world!"

print(astring[3:7])

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

Run

[Powered by DataCamp](https://www.datacamp.com/)

This prints a slice of the string, starting at index 3, and ending at index 6. But why 6 and not 7? Again, most programming languages do this - it makes doing math inside those brackets easier.

If you just have one number in the brackets, it will give you the single character at that index. If you leave out the first number but keep the colon, it will give you a slice from the start to the number you left in. If you leave out the second number, it will give you a slice from the first number to the end.

You can even put negative numbers inside the brackets. They are an easy way of starting at the end of the string instead of the beginning. This way, -3 means "3rd character from the end".

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

astring = "Hello world!"

print(astring[3:7:2])

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

Run

[Powered by DataCamp](https://www.datacamp.com/)

This prints the characters of string from 3 to 7 skipping one character. This is extended slice syntax. The general form is [start:stop:step].

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

3

astring = "Hello world!"

print(astring[3:7])

print(astring[3:7:1])

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

Run

[Powered by DataCamp](https://www.datacamp.com/)

Note that both of them produce same output

There is no function like strrev in C to reverse a string. But with the above mentioned type of slice syntax you can easily reverse a string like this

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

astring = "Hello world!"

print(astring[::-1])

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

Run

[Powered by DataCamp](https://www.datacamp.com/)

This

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

3

astring = "Hello world!"

print(astring.upper())

print(astring.lower())

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

Run

[Powered by DataCamp](https://www.datacamp.com/)

These make a new string with all letters converted to uppercase and lowercase, respectively.

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

3

astring = "Hello world!"

print(astring.startswith("Hello"))

print(astring.endswith("asdfasdfasdf"))

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

Run

[Powered by DataCamp](https://www.datacamp.com/)

This is used to determine whether the string starts with something or ends with something, respectively. The first one will print True, as the string starts with "Hello". The second one will print False, as the string certainly does not end with "asdfasdfasdf".

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

astring = "Hello world!"

afewwords = astring.split(" ")

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

Run

[Powered by DataCamp](https://www.datacamp.com/)

This splits the string into a bunch of strings grouped together in a list. Since this example splits at a space, the first item in the list will be "Hello", and the second will be "world!".

## Exercise

Try to fix the code to print out the correct information by changing the string.

* [script.py](https://www.learnpython.org/en/Basic_String_Operations)



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

s = "Hey there! what should this string be?"

# Length should be 20

print("Length of s = %d" % len(s))

# First occurrence of "a" should be at index 8

print("The first occurrence of the letter a = %d" % s

.index("a"))

# Number of a's should be 2

print("a occurs %d times" % s.count("a"))

# Slicing the string into bits

print("The first five characters are '%s'" % s[:5]) #

Start to 5

print("The next five characters are '%s'" % s[5:10]) #

5 to 10

print("The thirteenth character is '%s'" % s[12]) #

Just number 12

print("The characters with odd index are '%s'" %s[1

::2]) #(0-based indexing)

print("The last five characters are '%s'" % s[-5:]) #

5th-from-last to end

* [IPython Shell](https://www.learnpython.org/en/Basic_String_Operations)



In [1]:

SolutionRun

* Simple input and out put :-

## Python Input and Output

With this topic, we begin our series of Python Practice tutorials. Every tutorial describes a specific topic with examples. A problem statement at the end of each tutorial will assess your understanding.

## Introduction

Like all high-level languages, Python is easy to read, takes less time to write, and is portable. This versatile programming language has two versions: Python 2 and Python 3. Wiki says: Python 2.x is legacy, Python 3.x is the present and future of the language. That is, Python 2 is no longer in development and all new features will be added in Python 3. Note that, keeping this in mind, the code examples in this tutorial are in Python 3. Wherever Python 2.x code is shown, it will be highlighted.

## Execution

Python executes code top to bottom, when written in the correct syntax. To execute the code in our python tutorials, you will need to install python in your machine as a prerequisite. A small description on how to install Python and get the interpreter running is given here. Once the interpreter is running you can start typing in commands to get the result.

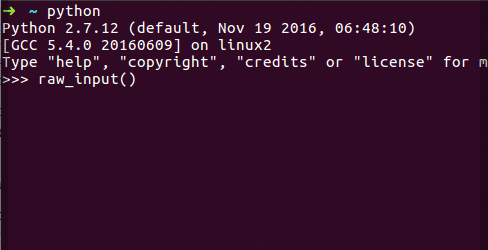
## Input using the input( ) function

A function is defined as a block of organized, reusable code used to perform a single, related action. Python has many built-in functions; you can also create your own. Python has an input function which lets you ask a user for some text input. You call this function to tell the program to stop and wait for the user to key in the data. In Python 2, you have a built-in function raw\_input(), whereas in Python 3, you have input(). The program will resume once the user presses the ENTER or RETURN key. Look at this example to get input from the keyboard using Python 2 in the interactive mode. Your output is displayed in quotes once you hit the ENTER key.

>>>raw\_input()

I am learning at hackerearth (This is where you type in)

'I am learning at hackerearth' (The interpreter showing you how the input is captured.)

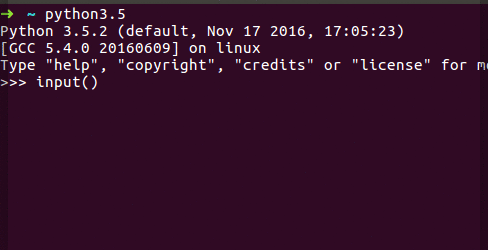


In Python 3.x, you need to use input().

>>> input()

I am learning at hackerearth.

'I am learning at hackerearth.'



You can always tell your user what to input by printing a prompt. There is no difference between input in Python 3 and raw\_input in Python 2 except for the keywords.

## Output using the print() function

To output your data to the screen, use the print() function. You can write print(argument) and this will print the argument in the next line when you press the ENTER key.

**Definitions to remember:** An argument is a value you pass to a function when calling it. A value is a letter or a number. A variable is a name that refers to a value. It begins with a letter. An assignment statement creates new variables and gives them values.

This syntax is valid in both Python 3.x and Python 2.x. For example, if your data is "Guido," you can put "Guido" inside the parentheses ( ) after print.

>>> print("Guido")

Guido

## More on using input

To capture the input in your program, you will need a variable. A variable is a container to hold data. (You will learn more on variables in a later tutorial.) You can take the input and assign it to a variable. This is done using the = operator before the input keyword and then putting the variable name before the = operator. For example, when you give a sentence "generic input" as the input, this gets assigned to a variable, say, my\_var. You can then print the value stored in my\_var. Let us understand this with the following example:

>>> # take an input and assign it to a variable

>>> beautiful\_number = input() # The data you key in the next line which is 6 will be assigned to beautiful\_number

6

>>> print(beautiful\_number) # the next line will print the value in beautiful\_number after you press enter or return

'6'

## Give a helpful hint during the prompt

It is often a good idea to tell the user what to input. You can do this by putting the hint in quotes inside the input parentheses. The hint will come in the next line and will wait for the user input. You can then type the input and when you hit the ENTER key, it will capture the input. In this example, "tell me a beautiful number" is the hint. This gets printed in the next line when asking for the input. If you type 6, this will be assigned to the variable beautiful\_number which we can print later.

>>> beautiful\_number = input("tell me a beautiful number ")

tell me a beautiful number 6

>>> print(beautiful\_number)

'6'

## More on using print

Say you want to print a specific string (a sequence of characters such as letters, punctuation marks, numbers, and letters) N number of times. The (asterisk) \* operator performs repetition on strings. You can print "5" six times. Inside the print parentheses, put â€œ5â€ followed by \* and the number of times you want â€œ5â€ repeated.

>>> print("5"\*6)

555555

You can separate the output using the comma delimiter. By default, this adds a space between the output items. For example, the sequence of numbers 5,6,7 separated by comma , gets printed with a space between one number and the next.

>>> print(5,6,7)

5 6 7

To change the output to what you want, use the keyword arguments sep and end to print ( ). When separating the output with a comma delimiter, you can also define the separation format using the â€œsepâ€ keyword.

>>> print('LOVE', 30, 82.2)

LOVE 30 82.2

>>> print('LOVE', 30, 82.2, sep=',')

'LOVE', 30, 82.2

By default, print goes to a new line at the end. You can change this by using the keyword â€œendâ€ as shown in the example below.

>>> print('LOVE', 30, 82.2, sep=',', end='!!\n')

'LOVE', 30, 82.2!!

For example, you can print the letters in the word "python" and all the letters will come in a new line.

>>> for i in "python":

... print(i)

...

p

y

t

h

o

n

You can change this default implementation. You can have a colon : between the letters instead of a new line.

>>> for i in "python":

... print(i, end=":")

...

p:y:t:h:o:n:

## Printing the result of a calculation

Say you can assign the number 7 to a variable population and if you write the logic population \* 7 inside the parentheses of print, it will just do the calculation up front and print the result.

>>> population = 7

>>> print("Population in 2050: ", population \* 1.28) # making the calculation in place

Population in 2050: 8.96

* + Simple output formatting :-

# **Python | Output Formatting**

Last Updated: 02-04-2020

There are several ways to present the output of a program, data can be printed in a human-readable form, or written to a file for future use. Sometimes user often wants more control the formatting of output than simply printing space-separated values. There are several ways to format output.

* To use formatted string literals , begin a string with f or F before the opening quotation mark or triple quotation mark.
* The str format  method of strings help a user to get a fancier Output
* User can do all the string handling by using string slicing and concatenation operations to create any layout that user wants. The string type has some methods that perform useful operations for padding strings to a given column width.

**Formatting output using String modulo operator(%) :**  
The % operator can also be used for string formatting. It interprets the left argument much like a printf()-style format string to be applied to the right argument. In Python, there is no printf() function but the functionality of the ancient printf is contained in Python. To this purpose, the modulo operator % is overloaded by the string class to perform string formatting. Therefore, it is often called string modulo (or sometimes even called modulus) operator.  
String modulo operator ( % ) is still available in Python(3.x) and user is using it widely. But nowadays the old style of formatting is removed from the language.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python program showing how to use  # string modulo operator(%) to print  # fancier output    # print integer and float value  print("Geeks : % 2d, Portal : % 5.2f" %(1, 05.333))    # print integer value  print("Total students : % 3d, Boys : % 2d" %(240, 120))    # print octal value  print("% 7.3o"% (25))    # print exponential value  print("% 10.3E"% (356.08977)) |

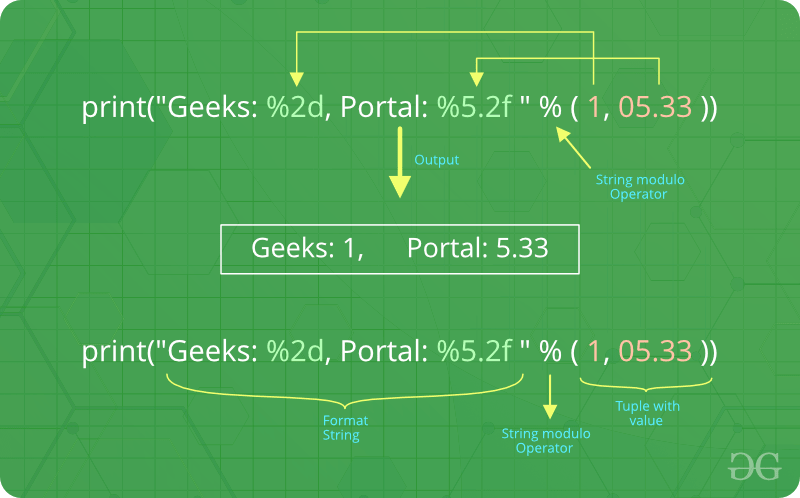
**Output :**

Geeks : 1, Portal : 5.33

Total students : 240, Boys : 120

031

3.561E+02

  
There are two of those in our example: “%2d” and “%5.2f”. The general syntax for a format placeholder is:

%[flags][width][.precision]type

Let’s take a look at the placeholders in our example.

* The first placeholder “%2d” is used for the first component of our tuple, i.e. the integer 1. The number will be printed with 2 characters. As 1 consists only of one digits, the output is padded with 1 leading blanks.
* The second one “%8.2f” is a format description for a float number. Like other placeholders, it is introduced with the % character. This is followed by the total number of digits the string should contain. This number includes the decimal point and all the digits, i.e. before and after the decimal point.
* Our float number 05.333 has to be formatted with 5 characters. The decimal part of the number or the precision is set to 2, i.e. the number following the “.” in our placeholder. Finally, the last character “f” of our placeholder stands for “float”.

**Formatting output using format method :**  
The format() method was added in Python(2.6). Format method of strings requires more manual effort. User use {} to mark where a variable will be substituted and can provide detailed formatting directives, but user also needs to provide the information to be formatted. This method lets us concatenate elements within an output through positional formatting. For Example –  
**Code 1:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python program showing  # use of format() method    # using format() method  print('I love {} for "{}!"'.format('Geeks', 'Geeks'))    # using format() method and refering  # a position of the object  print('{0} and {1}'.format('Geeks', 'Portal'))    print('{1} and {0}'.format('Geeks', 'Portal')) |

**Output :**

I love Geeks for "Geeks!"

Geeks and Portal

Portal and Geeks

The brackets and characters within them (called **format fields**) are replaced with the objects passed into the format() method. A number in the brackets can be used to refer to the position of the object passed into the format() method.  
   
**Code 2:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python program showing  # a use of format() method    # combining positional and keyword arguments  print('Number one portal is {0}, {1}, and {other}.'       .format('Geeks', 'For', other ='Geeks'))    # using format() method with number  print("Geeks :{0:2d}, Portal :{1:8.2f}".        format(12, 00.546))    # Changing positional argument  print("Second argument: {1:3d}, first one: {0:7.2f}".        format(47.42, 11))    print("Geeks: {a:5d},  Portal: {p:8.2f}".       format(a = 453, p = 59.058)) |

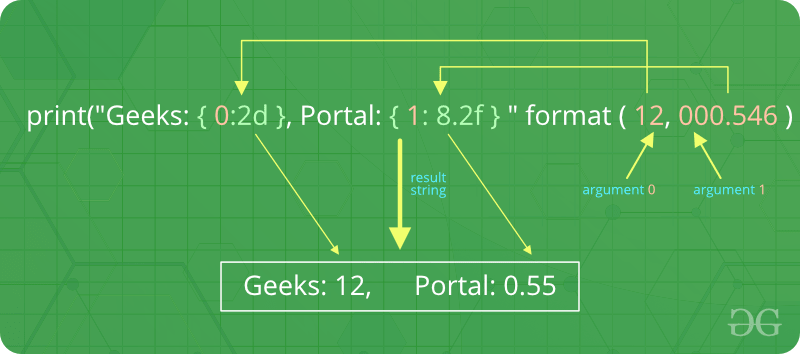
**Output:**

Number one portal is Geeks, For, and Geeks.

Geeks :12, Portal : 0.55

Second argument: 11, first one: 47.42

Geeks: 453, Portal: 59.06

The following diagram with an example usage depicts how the format method works for positional parameters:  
  
   
**Code 3:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python program to  # show format () is  # used in dictionary    tab = {'geeks': 4127, 'for': 4098, 'geek': 8637678}    # using format() in dictionary  print('Geeks: {0[geeks]:d}; For: {0[for]:d}; '      'Geeks: {0[geek]:d}'.format(tab))    data = dict(fun ="GeeksForGeeks", adj ="Portal")    # using format() in dictionary  print("I love {fun} computer {adj}".format(\*\*data)) |

**Output:**

Geeks: 4127; For: 4098; Geeks: 8637678

I love GeeksForGeeks computer Portal

**Formatting output using String method :**  
In this output is formatted by using string slicing and concatenation operations. The string type has some methods that help in formatting a output in an fancier way. Some of method which help in formatting a output are [str.ljust()](https://www.geeksforgeeks.org/python-string-ljust-rjust-center/), [str.rjust()](https://www.geeksforgeeks.org/python-string-ljust-rjust-center/), [str.centre()](https://www.geeksforgeeks.org/python-string-ljust-rjust-center/)

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python program to  # format a output using  # string() method    cstr = "I love geeksforgeeks"    # Printing the center aligned  # string with fillchr  print ("Center aligned string with fillchr: ")  print (cstr.center(40, '#'))    # Printing the left aligned  # string with "-" padding  print ("The left aligned string is : ")  print (cstr.ljust(40, '-'))    # Printing the right aligned string  # with "-" padding  print ("The right aligned string is : ")  print (cstr.rjust(40, '-')) |

**Output:**

Center aligned string with fillchr:

##########I love geeksforgeeks##########

The left aligned string is :

I love geeksforgeeks--------------------

The right aligned string is :

--------------------I love geeksforgeeks

Python operaters :-

Python Operators

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators
* Identity operators
* Membership operators
* Bitwise operators

Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** | **Try it** |
| + | Addition | x + y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_add) |
| - | Subtraction | x - y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_sub) |
| \* | Multiplication | x \* y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_mult) |
| / | Division | x / y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_div) |
| % | Modulus | x % y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_mod) |
| \*\* | Exponentiation | x \*\* y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_exp) |
| // | Floor division | x // y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_floordiv) |

Python Assignment Operators

Assignment operators are used to assign values to variables:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Example** | **Same As** | **Try it** |
| = | x = 5 | x = 5 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass1) |
| += | x += 3 | x = x + 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass2) |
| -= | x -= 3 | x = x - 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass3) |
| \*= | x \*= 3 | x = x \* 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass4) |
| /= | x /= 3 | x = x / 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass5) |
| %= | x %= 3 | x = x % 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass6) |
| //= | x //= 3 | x = x // 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass7) |
| \*\*= | x \*\*= 3 | x = x \*\* 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass8) |
| &= | x &= 3 | x = x & 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass9) |
| |= | x |= 3 | x = x | 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass10) |
| ^= | x ^= 3 | x = x ^ 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass11) |
| >>= | x >>= 3 | x = x >> 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass12) |
| <<= | x <<= 3 | x = x << 3 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_ass13) |

Python Comparison Operators

Comparison operators are used to compare two values:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** | **Try it** |
| == | Equal | x == y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare1) |
| != | Not equal | x != y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare2) |
| > | Greater than | x > y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare4) |
| < | Less than | x < y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare5) |
| >= | Greater than or equal to | x >= y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare6) |
| <= | Less than or equal to | x <= y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare7) |

Python Logical Operators

Logical operators are used to combine conditional statements:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Try it** |
| and | Returns True if both statements are true | x < 5 and  x < 10 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_logical1) |
| or | Returns True if one of the statements is true | x < 5 or x < 4 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_logical2) |
| not | Reverse the result, returns False if the result is true | not(x < 5 and x < 10) | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_logical3) |

Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Try it** |
| is | Returns True if both variables are the same object | x is y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_identity1) |
| is not | Returns True if both variables are not the same object | x is not y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_identity2) |

Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Try it** |
| in | Returns True if a sequence with the specified value is present in the object | x in y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_membership1) |
| not in | Returns True if a sequence with the specified value is not present in the object | x not in y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_membership2) |

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Description** |
| & | AND | Sets each bit to 1 if both bits are 1 |
| | | OR | Sets each bit to 1 if one of two bits is 1 |
| ^ | XOR | Sets each bit to 1 if only one of two bits is 1 |
| ~ | NOT | Inverts all the bits |
| << | Zero fill left shift | Shift left by pushing zeros in from the right and let the leftmost  bits fall off |
| >> | Signed right shift | Shift right by pushing copies of the leftmost bit in from the  left,  and let the rightmost bits fall off |

MODULE 3:-PYTHON PROGRAM FLOW

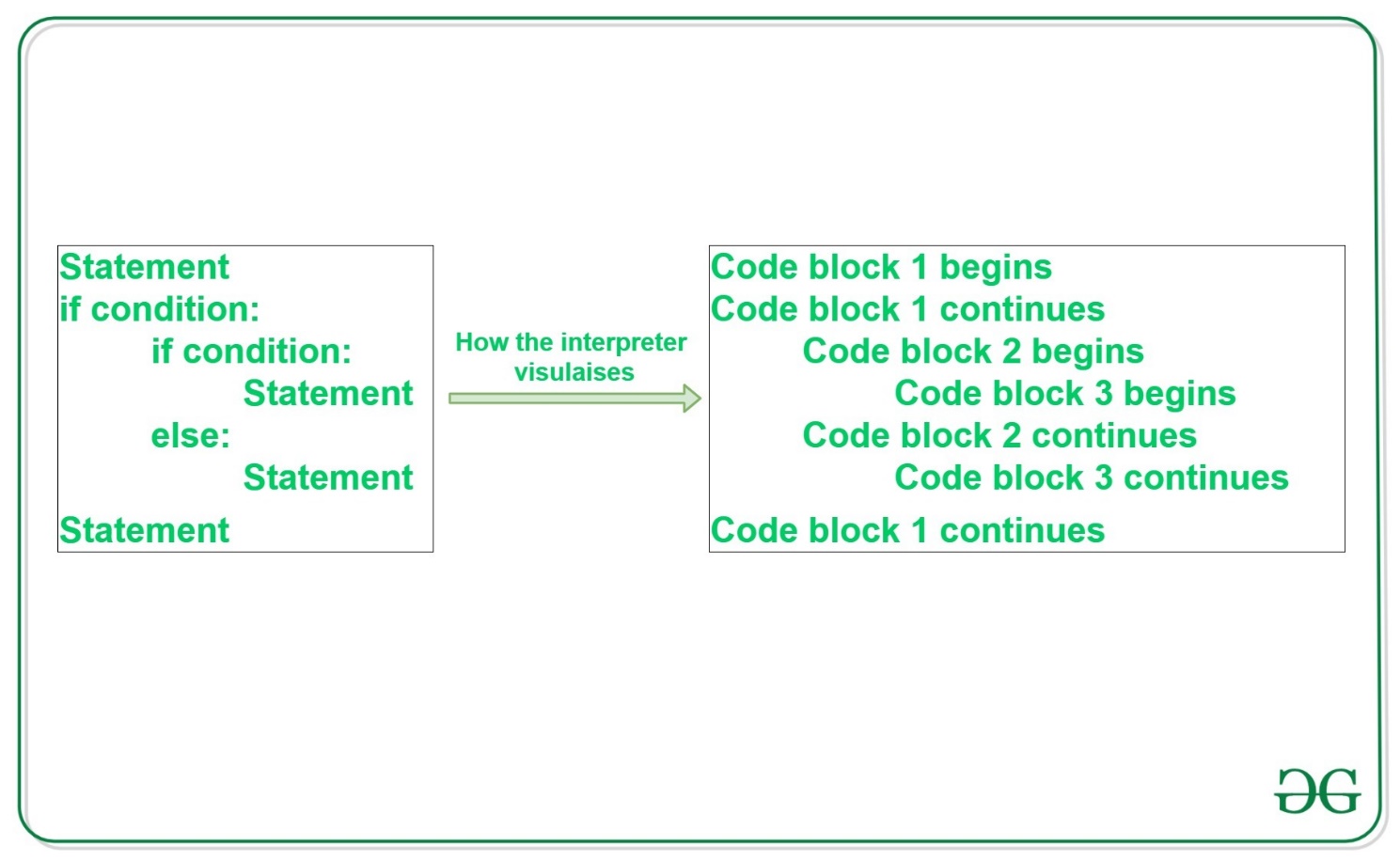
# **Indentation in Python**

Last Updated: 26-11-2019

Indentation is a very important concept of Python because without proper indenting the Python code, you will end up seeing IndentationError and the code will not get compiled.

## Indentation

In simple terms indentation refers to adding white space before a statement. But the question arises is it even necessary?  
To understand this consider a situation where you are reading a book and all of a sudden all the page numbers from the book went missing. So you don’t know, where to continue reading and you will get confused. This situation is similar with Python. Without indentation, Python does not know which statement to execute next or which statement belongs to which block. This will lead to IndentationError.



In the above example,

* Statement (line 1), if condition (line 2), and statement (last line) belongs to the same block which means that after statement 1, if condition will be executed. and suppose the if condition becomes False then the Python will jump to the last statement for execution.
* The nested if-else belongs to block 2 which means that if nested if becomes False, then Python will execute the statements inside the else condition.
* Statements inside nested if-else belongs to block 3 and only one statement will be executed depending on the if-else condition.

Python indentation is a way of telling a Python interpreter that the group of statements belongs to a particular block of code. A block is a combination of all these statements. Block can be regarded as the grouping of statements for a specific purpose. Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation to highlight the blocks of code. Whitespace is used for indentation in Python. All statements with the same distance to the right belong to the same block of code. If a block has to be more deeply nested, it is simply indented further to the right. You can understand it better by looking at the following lines of code.

**Example #1:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python program showing  # indentation    site = 'gfg'    if site == 'gfg':      print('Logging on to geeksforgeeks...')  else:      print('retype the URL.')  print('All set !') |

**Output:**

Logging on to geeksforgeeks...

All set !

The lines print(‘Logging on to geeksforgeeks…’) and print(‘retype the URL.’) are two separate code blocks. The two blocks of code in our example if-statement are both indented four spaces. The final print(‘All set!’) is not indented, and so it does not belong to the else-block.

**Example #2:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| j = 1    while(j<= 5):       print(j)       j = j + 1 |

**Output:**

1

2

3

4

5

To indicate a block of code in Python, you must indent each line of the block by the same whitespace. The two lines of code in the while loop are both indented four spaces. It is required for indicating what block of code a statement belongs to. For example, j=1 and while(j<=5): is not indented, and so it is not within while block. So, Python code structures by indentation.

* If statements and its related statements :
* Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.
* Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.
* Following is the general form of a typical decision making structure found in most of the programming languages −
* 
* Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.
* Python programming language provides following types of decision making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Sr.No.** | **Statement & Description** |
| 1 | [if statements](https://www.tutorialspoint.com/python/python_if_statement.htm)  An **if statement** consists of a boolean expression followed by one or more statements. |
| 2 | [if...else statements](https://www.tutorialspoint.com/python/python_if_else.htm)  An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE. |
| 3 | [nested if statements](https://www.tutorialspoint.com/python/nested_if_statements_in_python.htm)  You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |

* Let us go through each decision making briefly −

## Single Statement Suites

* If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.
* Here is an example of a **one-line if** clause −
* #!/usr/bin/python
* var = 100
* if ( var == 100 ) : print "Value of expression is 100"
* print "Good bye!"
* When the above code is executed, it produces the following result −
* Value of expression is 100
* Good bye!
* In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.
* Programming languages provide various control structures that allow for more complicated execution paths.
* A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −
* 
* Python programming language provides following types of loops to handle looping requirements.

|  |  |
| --- | --- |
| **Sr.No.** | **Loop Type & Description** |
| 1 | [while loop](https://www.tutorialspoint.com/python/python_while_loop.htm)  Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| 2 | [for loop](https://www.tutorialspoint.com/python/python_for_loop.htm)  Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | [nested loops](https://www.tutorialspoint.com/python/python_nested_loops.htm)  You can use one or more loop inside any another while, for or do..while loop. |

## Loop Control Statements

* Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
* Python supports the following control statements. Click the following links to check their detail.
* Let us go through the loop control statements briefly

|  |  |
| --- | --- |
| **Sr.No.** | **Control Statement & Description** |
| 1 | [break statement](https://www.tutorialspoint.com/python/python_break_statement.htm)  Terminates the loop statement and transfers execution to the statement immediately following the loop. |
| 2 | [continue statement](https://www.tutorialspoint.com/python/python_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | [pass statement](https://www.tutorialspoint.com/python/python_pass_statement.htm)  The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. |

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



Python programming language provides following types of loops to handle looping requirements.

|  |  |
| --- | --- |
| **Sr.No.** | **Loop Type & Description** |
| 1 | [while loop](https://www.tutorialspoint.com/python/python_while_loop.htm)  Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| 2 | [for loop](https://www.tutorialspoint.com/python/python_for_loop.htm)  Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | [nested loops](https://www.tutorialspoint.com/python/python_nested_loops.htm)  You can use one or more loop inside any another while, for or do..while loop. |
| 3 | [pass statement](https://www.tutorialspoint.com/python/python_pass_statement.htm)  The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. |

Examples for if statements and its related statemennts :-

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the if keyword.

### Example

If statement:

a = 33  
b = 200  
if b > a:  
  print("b is greater than a")

## Elif

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

### Example

a = 33  
b = 33  
if b > a:  
  print("b is greater than a")  
elif a == b:  
  print("a and b are equal")

## Else

The else keyword catches anything which isn't caught by the preceding conditions.

### Example

a = 200  
b = 33  
if b > a:  
  print("b is greater than a")  
elif a == b:  
  print("a and b are equal")  
else:  
  print("a is greater than b")

 this example a is greater than b, so the first condition is not true, also the elif condition is not true, so we go to the else condition and print to screen that "a is greater than b".

You can also have an else without the elif:

### Example

a = 200  
b = 33  
if b > a:  
  print("b is greater than a")  
else:  
  print("b is not greater than a")

## Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

### Example

One line if statement:

if a > b: print("a is greater than b")

## Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

### Example

One line if else statement:

a = 2  
b = 330  
print("A") if a > b else print("B")

You can also have multiple else statements on the same line:

### Example

One line if else statement, with 3 conditions:

a = 330  
b = 330  
print("A") if a > b else print("=") if a == b else print("B")

## And

The and keyword is a logical operator, and is used to combine conditional statements:

### Example

Test if a is greater than b, AND if c is greater than a:

a = 200  
b = 33  
c = 500  
if a > b and c > a:  
  print("Both conditions are True")

## Or

The or keyword is a logical operator, and is used to combine conditional statements:

### Example

Test if a is greater than b, OR if a is greater than c:a = 200  
b = 33  
c = 500  
if a > b or a > c:  
  print("At least one of the conditions is True")

**Nested If**

You can have if statements inside if statements, this is called nested if statements.

### Example

x = 41  
  
if x > 10:  
  print("Above ten,")  
  if x > 20:  
    print("and also above 20!")  
  else:  
    print("but not above 20.")

## The pass Statement

if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

### Example

a = 33  
b = 200  
  
if b > a:  
  pass

Top of Form

## Test Yourself With Exercises

## Exercise:

Print "Hello World if a is greater than b.

a = 50

b = 10

 a  b

print("Hello World")

## The break Statement

With the break statement we can stop the loop even if the while condition is true:

### Example

Exit the loop when i is 3:

i = 1  
while i < 6:  
  print(i)  
  if i == 3:

the while loop statemens:-

## The while Loop

With the while loop we can execute a set of statements as long as a condition is true.

### Example

Print i as long as i is less than 6:

i = 1  
while i < 6:  
  print(i)  
  i += 1

**Note:** remember to increment i, or else the loop will continue forever.

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

## The break Statement

With the break statement we can stop the loop even if the while condition is true:

### Example

Exit the loop when i is 3:

i = 1  
while i < 6:  
  print(i)  
  if i == 3:  
    break  
  i += 1

for loop statements:-

# **Python For Loops**

## Python For Loops

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

### Example

Print each fruit in a fruit list:

fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
  print(x)

The for loop does not require an indexing variable to set beforehand.

## Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

### Example

Loop through the letters in the word "banana":

for x in "banana":  
  print(x)

## The break Statement

With the break statement we can stop the loop before it has looped through all the items:

### Example

Exit the loop when x is "banana":

fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
  print(x)  
  if x == "banana":  
    break

### Example

Exit the loop when x is "banana", but this time the break comes before the print:

fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
  if x == "banana":  
    break  
  print(x)

the range statements:-

## range() function syntax and arguments

range(start, stop[, step])

It takes three arguments. Out of the three 2 arguments are optional. I.e., start and step are the optional arguments.

1. A **start** argument is a starting number of the sequence. i.e., lower limit. By default, it starts with 0 if not specified.
2. A **stop** argument is an upper limit. i.e., generate numbers up to this number, The range() doesn’t include this number in the result.
3. The **step** is a difference between each number in the result. The default value of the step is 1 if not specified.

### range() function Examples

Let see all the possible scenarios now. Below are the **three variant of range()** function.

**Example one**– Using only one argument

# Print first 5 numbers using range function

for i in range(5):

print(i, end=', ')

 Copy

**Output**:

0, 1, 2, 3, 4,

Only a stop argument is passed to range() . So by default, it takes start = 0 and step = 1.

**Example Two –**using two arguments (i.e., start and stop)

# Print integers within given start and stop number using range()

for i in range(5, 10):

print(i, end=', ')

 Copy

**Output**:

5, 6, 7, 8, 9,

**Note**: By default, it took step value as 1.

**Example Three** – using all three arguments

# using start, stop, and step arguments in range()

print("Printing All even numbers between 2 and 10 using range()")

for i in range(2, 10, 2):

print(i, end=', ')

 Copy

**Output**:

Printing All even numbers between and 10 using range()

2, 4, 6, 8,

All three arguments are specified i.e., start = 2, stop = 10, step = 2.  The step value is 2 so the difference between each number is 2.

**Practice Problem**

Generate a range of numbers from 9 to 100 divisible by 3 in Python using range() function.

### Points to remember about range() function arguments

* range() only works with the integers. **All arguments must be integers**. You can not use float number or any other type in a start, stop and step argument of a **range()**.
* All three arguments can be positive or negative.
* The step value must not be zero. If a step is zero Python raises a **ValueError** exception.

## for i in range – for loop with range()

As you know for loop executes a block of code or statement repeatedly for the fixed number of times. Using for loop we can iterate over a sequence of numbers produced by the range() function. Let’s see how to use for loop and range() function to **print the odd numbers between 1 and 10**. Using this example, we can understand how i is getting its value when we use range() and for loop together.

for i in range(1, 10, 2):

print("Current value of i is:", i)

 Copy

**Output**:

Current value of i is: 1

Current value of i is: 3

Current value of i is: 5

Current value of i is: 7

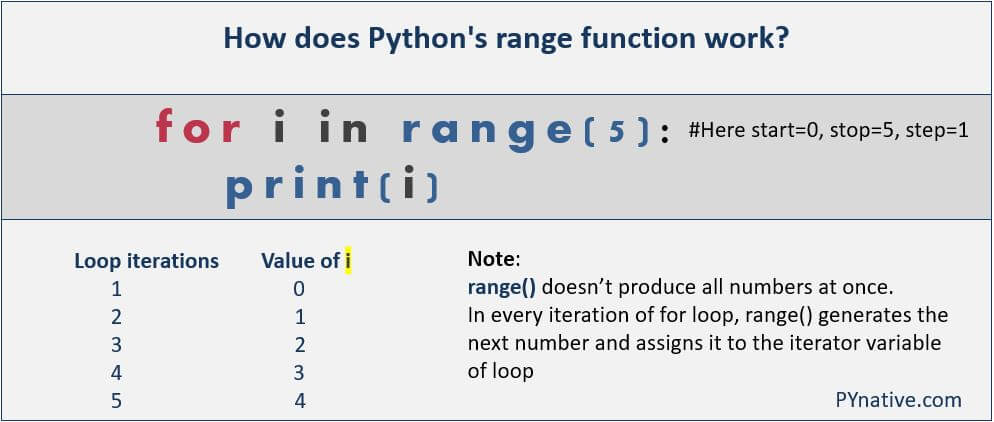
Current value of i is: 9

**In for i in range() i is the iterator variable**. To understand what does for i in range() mean in Python, first, we need to understand the working of range() function.  The range() function uses the generator to produce numbers within a range, i.e., it doesn’t produce all numbers at once. It generates the next value only when for loop iteration asked for it. In each loop iteration, Python generates the next value and assign it to the iterator variable i.

**Program execution**

* As you can see in the output, the variable i is not getting the value 1, 3, 5, 7, 9 at the same time.
* In the first iteration of for loop value of i is the value of start.  i.e., The first value of i is the starting number of a range. Here the range starts at 1.
* Next, In every subsequent iteration of for loop, the value of i incremented sequentially. The value of i is determined by the formula i = i + step. i.e., in the second iteration, i become 3, and so on.

As you know, In every iteration of for loop, range() generates the next number and assigns it to the iterator variable i. i.e., We get numbers on demand ( range() produces number one by one as the loop moves to the next iteration). Because of this behavior range() is faster and saves memory.

Working of Python range function with for loop

**Practice Problem**

Print the following number pattern using Python range() and for loop.

1

2 2

3 3 3

## Inclusive range

In this section, we will learn how to generate an inclusive range. The **range(n) is of exclusive nature** that is why it doesn’t include the last number in the output.  i.e., The given endpoint is never part of the generated result. For example, range(0, 5) = [0, 1, 2, 3, 4]. The result contains numbers from 0 to up to 5 but not 5 and the total count is 5. The range(start, stop) not include stop number in the output because the index (i) always starts with 0 in Python.

If you want to include the last number in the output i.e., If you want an inclusive range then set stop argument value as stop+step.

**Inclusive range() example**.

# Printing inclusive range

start = 1

stop = 5

step = 1

stop +=step #now stop is 6

for i in range(start, stop, step):

print(i, end=', ')

 Copy

**Output**:

1, 2, 3, 4, 5,

**Example 2**

# Printing inclusive range

start = 2

stop = 10

step = 2

stop +=step #now stop is 12

for i in range(start, stop, step):

print(i, end=', ')

 Copy

**Output**:

2, 4, 6, 8, 10,

## Python range step

A step is an optional argument of a range(). The step is a **difference between each number** in the result sequence.  If the step size is 2, then the difference between each number is 2. The default size of a step is **1** if not specified. We can perform lots of operations by effectively using step arguments such as reversing a sequence, printing negative ranges.

**Decrementing with range() using a negative step**

We can use negative values in all the arguments of range() function i.e., start, stop, and step.

start = -2

stop = -10

step = -2

print("Negative number range")

for number in range(start, stop, step):

print(number, end=', ')

 Copy

**Output**:

Negative number range

-2, -4, -6, -8,

Let’s understand the above program, we set, start = -2, stop = -10, step = -2.

* In the 1st iteration of for loop, the result is -2
* In the 2nd iteration of for loop, the result is -2, -4 because -2+(-2) = -4 and so on.
* And Last iteration output is -2, -4, -6,-8

**Decrementing with the range from Negative to Positive number**

Here in this example, we will learn how to use a step argument to display a range of numbers from negative to positive. Range of negative numbers.

# printing range from negative to positive

for num in range(-2, 5, 1):

print(num, end=", ")

 Copy

The output of the above program

-2, -1, 0, 1, 2, 3, 4,

**Python range from Positive to Negative number**

Here in this example, we can learn how to use step argument effectively to display numbers from positive to negative.

print (" printing range from Positive to Negative")

for num in range(2,-5,-1):

print(num, end=", ")

 Copy

**Output**:

printing range from Positive to Negative

2, 1, 0, -1, -2, -3, -4,

## Convert range() to List

If you execute print( type( range(10) ) ) you will get <class 'range'> as output. Python range() function doesn’t return a list type. It returns a range object, i.e., sequence object of type range, So as a result, we get an immutable sequence object of integers.

We can convert the output of a range() to the Python list. **Use list class to convert range output to list**. Let’s understand this with the following example.

print("Converting python range() to list")

even\_list = list( range(2, 10, 2))

print("printing list", even\_list)

 Copy

**Output**:

Converting python range() to list

printing list [2, 4, 6, 8]

We can also use range() function to access Python list items using its index number.

print("Use of range() to access Python list using index number")

sample\_list = [10, 20, 30, 40, 50]

for i in range(len(sample\_list)):

print("List item at index ", i, "is ", sample\_list[i])

 Copy

**Output**:

Use of range() to access Python list using index number

List item at index 0 is 10

List item at index 1 is 20

List item at index 2 is 30

List item at index 3 is 40

List item at index 4 is 50

**Note**: Using a len(list), we can get a count of list items, We used this count in range() to iterate for loop fixed number of times.

## Reverse range

If you want to print the sequence of numbers within range by descending order or reverse order in Python then its possible, there are two ways to do this.

**The first is to use a negative or down step value**. i.e., set the step argument of a range() to -1. For example, if you want to display a number sequence like [5, 4, 3, 2, 1, 0] i.e., we want reverse iteration or backward iteration of for loop with range() function.

Let’s see how to **loop backward using indices** in Python to display a range of numbers from 5 to 0.

print ("Displaying a range of numbers by reverse order")

for i in range(5, -1, -1):

print (i, end=', ')

 Copy

**Output**:

Displaying a range of numbers by reverse order

5, 4, 3, 2, 1, 0

**Use the reversed function to reverse range in Python**

Alternatively, using The reversed() function, we can reverse any sequence. If we use the reversed() function with range(), that will return a **range\_iterator** that accesses the given range of numbers in the reverse order. The below example will let you know how to make a reverse for loop in Python.

print("Printing reverse range using reversed()")

for i in reversed(range(0, 5)):

print(i)

 Copy

**Output**:

Printing reverse range using reversed()

4

3

2

1

0

Check the **output type** if  we use range() with reversed()

print("Checking the type")

print(type(range(0, 5)))

print(type(reversed(range(0,5))))

 Copy

Output:

Checking the type

<class 'range'>

<class 'range\_iterator'>

Also, If you need the list out of it, you need to convert the output of the reversed() function to list. So you can get the reverse list of ranges.

**Print a list in reverse order with range().**

print("Printing list in reverse order with range")

reverseed\_list = list(reversed(range(0, 5)))

print(reverseed\_list)

print("Second example to reverse list with range")

reverse\_list2 = list(range(5, -1, -1))

print(reverse\_list2)

print("Third Example to reverse list with range")

reverse\_list3 = list(range(2, 20, 2)[::-1])

print(reverse\_list3)

 Copy

**Output**:

Printing list in reverse order with range

[4, 3, 2, 1, 0]

Second example to reverse list with range

[5, 4, 3, 2, 1, 0]

Third Example to reverse list with range

[18, 16, 14, 12, 10, 8, 6, 4, 2]

## Using float Numbers in range()

**Python’s range() function doesn’t support the float numbers**.  i.e., we cannot use floating-point or non-integer numbers in any of its arguments. we can use only integer numbers. However, we can create a custom range function where we can use float numbers like 0.1 or 1.6 in any of its arguments. I have demonstrated this in the below example.

def frange(start, stop=None, step=None):

if stop == None:

stop = start + 0.0

start = 0.0

if step == None:

step = 1.0

while True:

if step > 0 and start >= stop:

break

elif step < 0 and start <= stop:

break

yield ("%g" % start) # return float number

start = start + step

print ("Printing float range")

floatList = frange(0.5, 1.0, 0.1)

for num in floatList:

print (num)

 Copy

**Output**:

Printing float range

0.5

0.6

0.7

0.8

0.9

Also, see [**All other ways to use float numbers in range() function.**](https://pynative.com/python-range-for-float-numbers/)

## Concatenating the result of two range() function

Let say you want to add range(5) + range(10,15). (**Note**: this code is a pseudo-code.)  And you want the concatenated range like [0, 1, 2, 3, 4, 10, 11, 12, 13, 14].

We can concatenate the output of two range functions using the ***itertools’s chain()*** function.

Program: Concatenating two range function results.

from itertools import chain

print ("Concatinated two range() function")

concatenated\_range = chain(range(10), range(50, 75))

for num in concatenated\_range:

print(num,end=", ")

 Copy

**Output**:

Concatinated two range() function

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,

## Access range() output with its index value

range() is the constructor returns a range object which is nothing but a sequence of numbers, this range object can also be accessed by its index number using **slice notation**. It supports both positive and negative indices. The below example explains the same.

print("accessing Python range objet with its index")

first\_number = range(0,10)[0] #printing 0th position number i.e. index ZERO means first number

print("First number in given range is: ", first\_number)

fifth\_number = range(0,10)[4]

print("fifth number in given range is: ", fifth\_number)

 Copy

**Output**:

accessing Python range objet with its index

First number in given range is: 0

fifth number in given range is: 4

* break and continue:-

# Python break and continue

#### **In this article, you will learn to use break and continue statements to alter the flow of a loop.**

## What is the use of break and continue in Python?

In Python, break and continue statements can alter the flow of a normal loop.

Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.

The break and continue statements are used in these cases.

## Python break statement

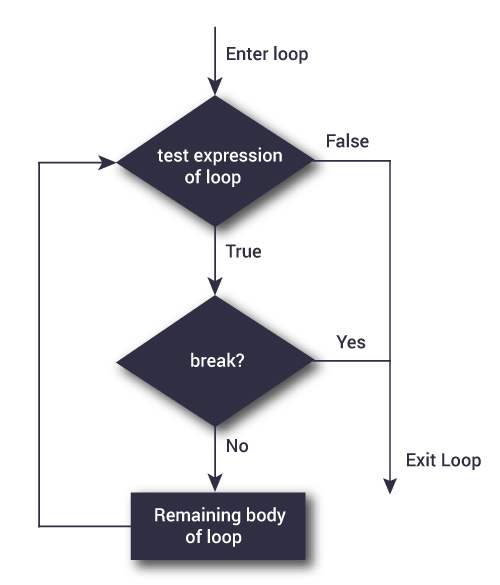
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

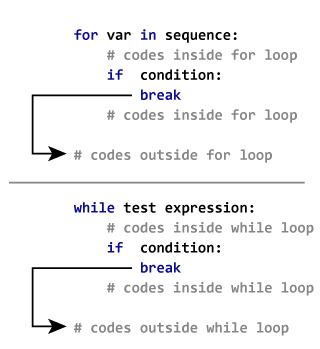
### Syntax of break

break

### Flowchart of break

Flowchart of break statement in Python

The working of break statement in [for loop](https://www.programiz.com/python-programming/for-loop) and [while loop](https://www.programiz.com/python-programming/while-loop) is shown below.

Working of the break statement

### Example: Python break

# Use of break statement inside the loop

for val in "string":

if val == "i":

break

print(val)

print("The end")

**Output**

s

t

r

The end

In this program, we iterate through the "string" sequence. We check if the letter is i, upon which we break from the loop. Hence, we see in our output that all the letters up till i gets printed. After that, the loop terminates.

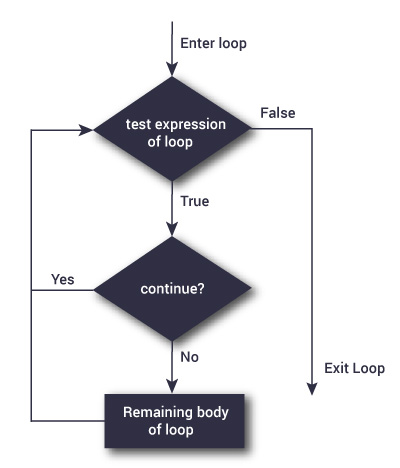
## Python continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

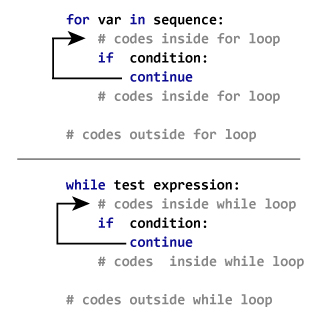
### Syntax of Continue

continue

### Flowchart of continue

Flowchart of continue statement in Python

The working of continue statement in for and while loop is shown below.

How continue statement works in python

### Example: Python continue

# Program to show the use of continue statement inside loops

for val in "string":

if val == "i":

continue

print(val)

print("The end")

**Output**

s

t

r

n

g

The end

This program is same as the above example except the break statement has been replaced with continue.

We continue with the loop, if the string is i, not executing the rest of the block. Hence, we see in our output that all the letters except i gets printed.

* ASSERT:-

# Python Assert Statement

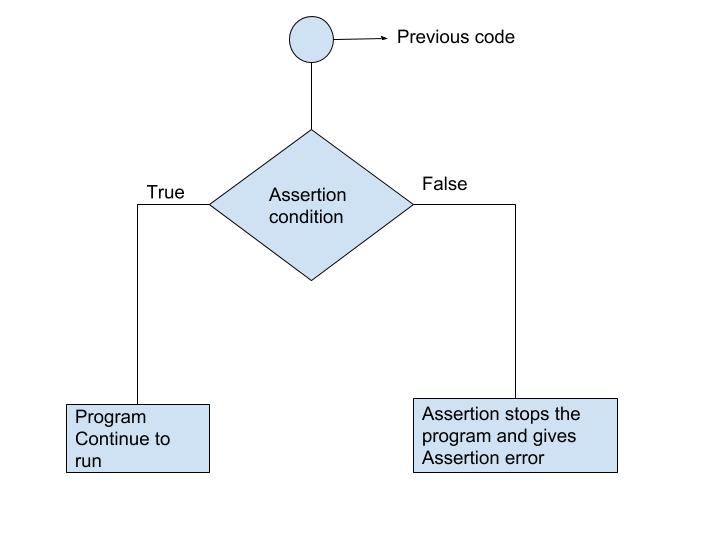
## What is Assertion?

Assertions are statements that assert or state a fact confidently in your program. For example, while writing a division function, you're confident the divisor shouldn't be zero, you assert divisor is not equal to zero.

Assertions are simply boolean expressions that check if the conditions return true or not. If it is true, the program does nothing and move to the next line of code. However, if it's false, the program stops and throws an error.

It is also a debugging tool as it brings the program on halt as soon as any error is occurred and shows on which point of the program error has occurred.

We can be clear by looking at the flowchart below:



## Python assert Statement

Python has built-in assert statement to use assertion condition in the program. assert statement has a condition or expression which is supposed to be always true. If the condition is false assert halts the program and gives an AssertionError.

#### **Syntax for using Assert in Pyhton:**

assert <condition>

assert <condition>,<error message>

In Python we can use assert statement in two ways as mentioned above.

1. assert statement has a condition and if the condition is not satisfied the program will stop and give AssertionError.
2. assert statement can also have a condition and a optional error message. If the condition is not satisfied assert stops the program and gives AssertionError along with the error message.

Let's take an example, where we have a function which will calculate the average of the values passed by the user and the value should not be an empty list. We will use assert statement to check the parameter and if the length is of the passed list is zero, program halts.

### Example 1: Using assert without Error Message

def avg(marks):

assert len(marks) != 0

return sum(marks)/len(marks)

mark1 = []

print("Average of mark1:",avg(mark1))

When we run the above program, the output will be:

AssertionError

We got an error as we passed an empty list mark1 to assert statement, the condition became false and assert stops the program and give AssertionError.

Now let's pass another list which will satisfy the assert condition and see what will be our output.

### Example 2: Using assert with error message

def avg(marks):

assert len(marks) != 0,"List is empty."

return sum(marks)/len(marks)

mark2 = [55,88,78,90,79]

print("Average of mark2:",avg(mark2))

mark1 = []

print("Average of mark1:",avg(mark1))

When we run the above program, the output will be:

Average of mark2: 78.0

AssertionError: List is empty.

We passed a non-empty list mark2 and also an empty list mark1 to the avg() function and we got output for mark2 list but after that we got an error AssertionError: List is empty. The assert condition was satisfied by the mark2 list and program to continue to run. However, mark1 doesn't satisfy the condition and gives an AssertionError.

## Key Points to Remember

* Assertions are the condition or boolean expression which are always supposed to be true in the code.
* assert statement takes an expression and optional message.
* assert statement is used to check types, values of argument and the output of the function.
* assert statement is used as debugging tool as it halts the program at the point where an error occurs.
* EXAMPLES FOR LOOPING:-

# Python for Loop explained with examples

A loop is a used for iterating over a set of statements repeatedly. In Python we have three types of loops **for**, **while** and **do-while**. In this guide, we will learn **for loop** and the other two loops are covered in the separate tutorials.

## Syntax of For loop in Python

for <variable> in <sequence>:

# body\_of\_loop that has set of statements

# which requires repeated execution

Here <variable> is a variable that is used for iterating over a <sequence>. On every iteration it takes the **next value** from <sequence>  until the end of sequence is reached.

Lets take few examples of for loop to understand the usage.

## Python – For loop example

The following example shows the use of for loop to iterate over a list of numbers. In the body of for loop we are calculating the square of each number present in list and displaying the same.

# Program to print squares of all numbers present in a list

# List of integer numbers

numbers = [1, 2, 4, 6, 11, 20]

# variable to store the square of each num temporary

sq = 0

# iterating over the given list

for val in numbers:

# calculating square of each number

sq = val \* val

# displaying the squares

print(sq)

Output:

1

4

16

36

121

400

## Function range()

In the above example, we have iterated over a list using for loop. However we can also use a range() function in for loop to iterate over numbers defined by range().

**range(n)**: generates a set of whole numbers starting from 0 to (n-1).  
For example:  
range(8) is equivalent to [0, 1, 2, 3, 4, 5, 6, 7]

**range(start, stop)**: generates a set of whole numbers starting from start to stop-1.  
For example:  
range(5, 9) is equivalent to [5, 6, 7, 8]

**range(start, stop, step\_size)**: The default step\_size is 1 which is why when we didn’t specify the step\_size, the numbers generated are having difference of 1. However by specifying step\_size we can generate numbers having the difference of step\_size.  
For example:  
range(1, 10, 2) is equivalent to [1, 3, 5, 7, 9]

Lets use the **range() function** in for loop:

## Python for loop example using range() function

Here we are using **range() function** to calculate and display the sum of first 5 natural numbers.

# Program to print the sum of first 5 natural numbers

# variable to store the sum

sum = 0

# iterating over natural numbers using range()

for val in range(1, 6):

# calculating sum

sum = sum + val

# displaying sum of first 5 natural numbers

print(sum)

Output:

15

## For loop with else block

Unlike [Java](https://beginnersbook.com/2015/03/for-loop-in-java-with-example/), In Python we can have an optional ‘else’ block associated with the loop. The ‘else’ block executes only when the loop has completed all the iterations. Lets take an example:

for val in range(5):

print(val)

else:

print("The loop has completed execution")

Output:

0

1

2

3

4

The loop has completed execution

**Note:** The else block only executes when the loop is finished.

## Nested For loop in Python

When a for loop is present inside another for loop then it is called a nested for loop. Lets take an example of nested for loop.

for num1 in range(3):

for num2 in range(10, 14):

print(num1, ",", num2)

Output:

0 , 10

0 , 11

0 , 12

0 , 13

1 , 10

1 , 11

1 , 12

1 , 13

2 , 10

2 , 11

2 , 12

2 , 13

# Python While Loop

BY CHAITANYA SINGH | FILED UNDER: [PYTHON TUTORIAL](https://beginnersbook.com/category/python-tutorial/)

**While loop** is used to iterate over a block of code repeatedly until a given condition returns false. In the last tutorial, we have seen [for loop in Python](https://beginnersbook.com/2018/01/python-for-loop/), which is also used for the same purpose. The main difference is that we use **while loop** when we are **not** certain of the number of times the loop requires execution, on the other hand when we exactly know how many times we need to run the loop, we use for loop.

## Syntax of while loop

while condition:

#body\_of\_while

The body\_of\_while is set of Python statements which requires repeated execution. These set of statements execute repeatedly until the given condition returns false.

## Flow of while loop

1. First the given condition is checked, if the condition returns false, the loop is terminated and the control jumps to the next statement in the program after the loop.  
2. If the condition returns true, the set of statements inside loop are executed and then the control jumps to the beginning of the loop for next iteration.

These two steps happen repeatedly as long as the condition specified in while loop remains true.

## Python – While loop example

Here is an example of while loop. In this example, we have a variable num and we are displaying the value of num in a loop, the loop has a increment operation where we are increasing the value of num. This is very important step, the while loop must have a increment or decrement operation, else the loop will run indefinitely, we will cover this later in infinite while loop.

num = 1

# loop will repeat itself as long as

# num < 10 remains true

while num < 10:

print(num)

#incrementing the value of num

num = num + 3

Output:

1

4

7

## Infinite while loop

**Example 1:**  
This will print the word ‘hello’ indefinitely because the condition will always be true.

while True:

print("hello")

Example 2:

num = 1

while num<5:

print(num)

This will print ‘1’ indefinitely because inside loop we are not updating the value of num, so the value of num will always remain 1 and the condition num < 5 will always return true.

## Nested while loop in Python

When a while loop is present inside another while loop then it is called nested while loop. Lets take an example to understand this concept.

i = 1

j = 5

while i < 4:

while j < 8:

print(i, ",", j)

j = j + 1

i = i + 1

Output:

1 , 5

2 , 6

3 , 7

## Python – while loop with else block

We can have a ‘else’ block associated with while loop. The ‘else’ block is optional. It executes only after the loop finished execution.

num = 10

while num > 6:

print(num)

num = num-1

else:

print("loop is finished")

Output:

10

9

8

7

loop is finished

MODULE 4:-FUNCTIONS & MODULES

* CREATING YOUR OWN FUNCTIONS:-

Throughout the previous tutorials in this series, you’ve seen many examples demonstrating the use of [built-in Python functions](https://realpython.com/python-data-types/#built-in-functions). In this tutorial, you’ll learn how to **define your own Python function**. You’ll learn when to divide your program into separate user-defined functions and what tools you’ll need to do this.

**Here’s what you’ll learn in this tutorial:**

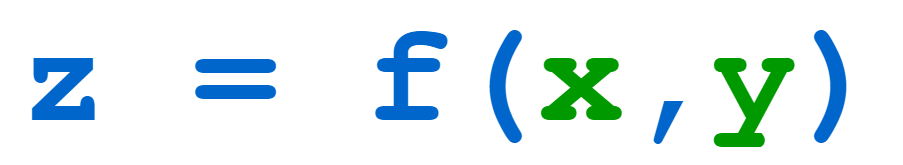
* How **functions** work in Python and why they’re beneficial
* How to **define and call** your own Python function
* Mechanisms for **passing arguments** to your function
* How to **return data** from your function back to the calling environment

**Free PDF Download:** [**Python 3 Cheat Sheet**](https://realpython.com/bonus/python-cheat-sheet-short/)

[Remove ads](https://realpython.com/account/join/)

## Functions in Python

You may be familiar with the mathematical concept of a **function**. A function is a relationship or mapping between one or more inputs and a set of outputs. In mathematics, a function is typically represented like this:

[](https://files.realpython.com/media/t.74ec5430f457.png)

Here, f is a function that operates on the inputs x and y. The output of the function is z. However, programming functions are much more generalized and versatile than this mathematical definition. In fact, appropriate function definition and use is so critical to proper software development that virtually all modern programming languages support both built-in and user-defined functions.

In programming, a **function** is a self-contained block of code that encapsulates a specific task or related group of tasks. In previous tutorials in this series, you’ve been introduced to some of the built-in functions provided by Python. [id()](https://realpython.com/python-variables/#object-identity), for example, takes one argument and returns that object’s unique integer identifier:

>>>

>>> s = 'foobar'

>>> id(s)

56313440

len() returns the length of the argument passed to it:

>>>

>>> a = ['foo', 'bar', 'baz', 'qux']

>>> len(a)

4

[any()](https://realpython.com/any-python/) takes an iterable as its argument and returns True if any of the items in the iterable are [truthy](https://realpython.com/python-data-types/#boolean-type-boolean-context-and-truthiness) and False otherwise:

>>>

>>> any([False, False, False])

False

>>> any([False, True, False])

True

>>> any(['bar' == 'baz', len('foo') == 4, 'qux' in {'foo', 'bar', 'baz'}])

False

>>> any(['bar' == 'baz', len('foo') == 3, 'qux' in {'foo', 'bar', 'baz'}])

True

Each of these built-in functions performs a specific task. The code that accomplishes the task is defined somewhere, but you don’t need to know where or even how the code works. All you need to know about is the function’s [interface](https://realpython.com/python-interface/):

1. What **arguments** (if any) it takes
2. What **values** (if any) it returns

Then you call the function and pass the appropriate arguments. Program execution goes off to the designated body of code and does its useful thing. When the function is finished, execution returns to your code where it left off. The function may or may not return data for your code to use, as the examples above do.

When you define your own Python function, it works just the same. From somewhere in your code, you’ll call your Python function and program execution will transfer to the body of code that makes up the function.

**Note:** In this case, you will know where the code is and exactly how it works because you wrote it!

When the function is finished, execution returns to the location where the function was called. Depending on how you designed the function’s interface, data may be passed in when the function is called, and return values may be passed back when it finishes.

## The Importance of Python Functions

Virtually all programming languages used today support a form of user-defined functions, although they aren’t always called functions. In other languages, you may see them referred to as one of the following:

* **Subroutines**
* **Procedures**
* **Methods**
* **Subprograms**

So, why bother defining functions? There are several very good reasons. Let’s go over a few now.

### Abstraction and Reusability

Suppose you write some code that does something useful. As you continue development, you find that the task performed by that code is one you need often, in many different locations within your application. What should you do? Well, you could just replicate the code over and over again, using your editor’s copy-and-paste capability.

Later on, you’ll probably decide that the code in question needs to be modified. You’ll either find something wrong with it that needs to be fixed, or you’ll want to enhance it in some way. If copies of the code are scattered all over your application, then you’ll need to make the necessary changes in every location.

**Note:** At first blush, that may seem like a reasonable solution, but in the long term, it’s likely to be a maintenance nightmare! While your code editor may help by providing a search-and-replace function, this method is error-prone, and you could easily introduce bugs into your code that will be difficult to find.

A better solution is to **define a Python function that performs the task**. Anywhere in your application that you need to accomplish the task, you simply call the function. Down the line, if you decide to change how it works, then you only need to change the code in one location, which is the place where the function is defined. The changes will automatically be picked up anywhere the function is called.

The **abstraction of functionality** into a function definition is an example of the Don’t repeat yourself function (DRY) of software development. This is arguably the strongest motivation for using functions.

### Modularity

Functions allow **complex processes** to be broken up into smaller steps. Imagine, for example, that you have a program that reads in a file, processes the file contents, and then writes an output file. Your code could look like this:

# Main program

# Code to read file in

<statement>

<statement>

<statement>

<statement>

# Code to process file

<statement>

<statement>

<statement>

<statement>

# Code to write file out

<statement>

<statement>

<statement>

<statement>

In this example, the main program is a bunch of code strung together in a long sequence, with whitespace and comments to help organize it. However, if the code were to get much lengthier and more complex, then you’d have an increasingly difficult time wrapping your head around it.

Alternatively, you could structure the code more like the following:

def read\_file():

# Code to read file in

<statement>

<statement>

<statement>

<statement>

def process\_file():

# Code to process file

<statement>

<statement>

<statement>

<statement>

def write\_file():

# Code to write file out

<statement>

<statement>

<statement>

<statement>

# Main program

read\_file()

process\_file()

write\_file()

This example is **modularized**. Instead of all the code being strung together, it’s broken out into separate functions, each of which focuses on a specific task. Those tasks are read, process, and write. The main program now simply needs to call each of these in turn.

**Note:** The def keyword introduces a new Python function definition. You’ll learn all about this very soon.

In life, you do this sort of thing all the time, even if you don’t explicitly think of it that way. If you wanted to move some shelves full of stuff from one side of your garage to the other, then you hopefully wouldn’t just stand there and aimlessly think, “Oh, geez. I need to move all that stuff over there! How do I do that???” You’d divide the job into manageable steps:

1. **Take** all the stuff off the shelves.
2. **Take** the shelves apart.
3. **Carry** the shelf parts across the garage to the new location.
4. **Re-assemble** the shelves.
5. **Carry** the stuff across the garage.
6. **Put** the stuff back on the shelves.

Breaking a large task into smaller, bite-sized sub-tasks helps make the large task easier to think about and manage. As programs become more complicated, it becomes increasingly beneficial to modularize them in this way.

### Namespace Separation

A **namespace** is a region of a program in which **identifiers** have meaning. As you’ll see below, when a Python function is called, a new namespace is created for that function, one that is distinct from all other namespaces that already exist.

The practical upshot of this is that variables can be defined and used within a Python function even if they have the same name as variables defined in other functions or in the main program. In these cases, there will be no confusion or interference because they’re kept in separate namespaces.

This means that when you write code within a function, you can use variable names and identifiers without worrying about whether they’re already used elsewhere outside the function. This helps minimize errors in code considerably.

Hopefully, you’re sufficiently convinced of the virtues of functions and eager to create some! Let’s see how.

## Function Calls and Definition

The usual syntax for defining a Python function is as follows:

def <function\_name>([<parameters>]):

<statement(s)>

The components of the definition are explained in the table below:

| **Component** | **Meaning** |
| --- | --- |
| def | The keyword that informs Python that a function is being defined |
| <function\_name> | A valid Python identifier that names the function |
| <parameters> | An optional, comma-separated list of parameters that may be passed to the function |
| : | Punctuation that denotes the end of the Python function header (the name and parameter list) |
| <statement(s)> | A block of valid Python statements |

The final item, <statement(s)>, is called the **body** of the function. The body is a block of statements that will be executed when the function is called. The body of a Python function is defined by indentation in accordance with the OFF-SIDE-RULE. This is the same as code blocks associated with a control structure, like an if or while statement.

The syntax for calling a Python function is as follows:

<function\_name>([<arguments>])

<arguments> are the values passed into the function. They correspond to the <parameters> in the Python function definition. You can define a function that doesn’t take any arguments, but the parentheses are still required. Both a function definition and a function call must always include parentheses, even if they’re empty.

As usual, you’ll start with a small example and add complexity from there. Keeping the time-honored mathematical tradition in mind, you’ll call your first Python function f(). Here’s a script file, foo.py, that defines and calls f():

1 def f():

2 s = '-- Inside f()'

3 print(s)

4

5 print('Before calling f()')

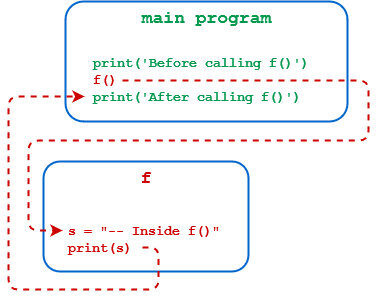
6 f()

7 print('After calling f()')

Here’s how this code works:

1. **Line 1** uses the def keyword to indicate that a function is being defined. Execution of the def statement merely creates the definition of f(). All the following lines that are indented (lines 2 to 3) become part of the body of f() and are stored as its definition, but they aren’t executed yet.
2. **Line 4** is a bit of whitespace between the function definition and the first line of the main program. While it isn’t syntactically necessary, it is nice to have. To learn more about whitespace around top-level Python function definitions, check out writing beautifull pythonic code with pep
3. **Line 5** is the first statement that isn’t indented because it isn’t a part of the definition of f(). It’s the start of the main program. When the main program executes, this statement is executed first.
4. **Line 6** is a call to f(). Note that empty parentheses are always required in both a function definition and a function call, even when there are no parameters or arguments. Execution proceeds to f() and the statements in the body of f() are executed.
5. **Line 7** is the next line to execute once the body of f() has finished. Execution returns to this print f() statement

The sequence of execution (or **control flow**) for foo.py is shown in the following diagram:

[](https://files.realpython.com/media/t.f3e2000ecb56.png)

When foo.py is run from a Windows command prompt, the result is as follows:

C:\Users\john\Documents\Python\doc>python foo.py

Before calling f()

-- Inside f()

After calling f()

Occasionally, you may want to define an empty function that does nothing. This is referred to as a **stub**, which is usually a temporary placeholder for a Python function that will be fully implemented at a later time. Just as a block in a control structure can’t be empty, neither can the body of a function. To define a stub function, use the pass  statement:

>>>

>>> def f():

... pass

...

>>> f()

As you can see above, a call to a stub function is syntactically valid but doesn’t do anything.

## Argument Passing

So far in this tutorial, the functions you’ve defined haven’t taken any arguments. That can sometimes be useful, and you’ll occasionally write such functions. More often, though, you’ll want to **pass data into a function** so that its behavior can vary from one invocation to the next. Let’s see how to do that.

* Functions parameters:-

## Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

* Function blocks begin with the keyword **def** followed by the function name and parentheses ( ( ) ).
* Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
* The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
* The code block within every function starts with a colon (:) and is indented.
* The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

## Syntax

def functionname( parameters ):

"function\_docstring"

function\_suite

return [expression]

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

## Example

The following function takes a string as input parameter and prints it on standard screen.

def printme( str ):

"This prints a passed string into this function"

print str

return

## Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

[Live Demo](http://tpcg.io/3mrnYY)

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme("I'm first call to user defined function!")

printme("Again second call to the same function")

When the above code is executed, it produces the following result −

I'm first call to user defined function!

Again second call to the same function

## Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

[Live Demo](http://tpcg.io/ZwGczd)

#!/usr/bin/python

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

mylist.append([1,2,3,4]);

print "Values inside the function: ", mylist

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

[Live Demo](http://tpcg.io/xgrZcs)

#!/usr/bin/python

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

mylist = [1,2,3,4]; # This would assig new reference in mylist

print "Values inside the function: ", mylist

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result −

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

* Variable arguments:-

## Function Arguments

You can call a function by using the following types of formal arguments −

* Required arguments
* Keyword arguments
* Default arguments
* Variable-length arguments

## Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme()

When the above code is executed, it produces the following result −

Traceback (most recent call last):

File "test.py", line 11, in <module>

printme();

TypeError: printme() takes exactly 1 argument (0 given)

## Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme( str = "My string")

When the above code is executed, it produces the following result −

My string

The following example gives more clear picture. Note that the order of parameters does not matter.

#!/usr/bin/python

# Function definition is here

def printinfo( name, age ):

"This prints a passed info into this function"

print "Name: ", name

print "Age ", age

return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

## Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

#!/usr/bin/python

# Function definition is here

def printinfo( name, age = 35 ):

"This prints a passed info into this function"

print "Name: ", name

print "Age ", age

return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

Name: miki

Age 35

## Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

def functionname([formal\_args,] \*var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

#!/usr/bin/python

# Function definition is here

def printinfo( arg1, \*vartuple ):

"This prints a variable passed arguments"

print "Output is: "

print arg1

for var in vartuple:

print var

return;

# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )

When the above code is executed, it produces the following result −

Output is:

10

Output is:

70

60

50

## The *Anonymous* Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

* Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
* An anonymous function cannot be a direct call to print because lambda requires an expression
* Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
* Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

## Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

lambda [arg1 [,arg2,.....argn]]:expression

Following is the example to show how *lambda* form of function works −

#!/usr/bin/python

# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function

print "Value of total : ", sum( 10, 20 )

print "Value of total : ", sum( 20, 20 )

When the above code is executed, it produces the following result −

Value of total : 30

Value of total : 40

## The *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows −

#!/usr/bin/python

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2

print "Inside the function : ", total

return total;

# Now you can call sum function

total = sum( 10, 20 );

print "Outside the function : ", total

When the above code is executed, it produces the following result −

Inside the function : 30

Outside the function : 30

## Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

* Global variables
* Local variables

## Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

#!/usr/bin/python

total = 0; # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2; # Here total is local variable.

print "Inside the function local total : ", total

return total;

# Now you can call sum function

sum( 10, 20 );

print "Outside the function global total : ", total

When the above code is executed, it produces the following result −

Inside the function local total : 30

Outside the function global total : 0

* Scope of a function:-

## Local Scope

A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

### Example

A variable created inside a function is available inside that function:

def myfunc():  
  x = 300  
  print(x)  
  
myfunc()

### Function Inside Function

As explained in the example above, the variable x is not available outside the function, but it is available for any function inside the function:

### Example

The local variable can be accessed from a function within the function:

def myfunc():  
  x = 300  
  def myinnerfunc():  
    print(x)  
  myinnerfunc()  
  
myfunc()

## Global Scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

### Example

A variable created outside of a function is global and can be used by anyone:

x = 300  
  
def myfunc():  
  print(x)  
  
myfunc()  
  
print(x)

### Naming Variables

If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

### Example

The function will print the local x, and then the code will print the global x:

x = 300  
  
def myfunc():  
  x = 200  
  print(x)  
  
myfunc()  
  
print(x)

## Global Keyword

If you need to create a global variable, but are stuck in the local scope, you can use the global keyword.

The global keyword makes the variable global.

### Example

If you use the global keyword, the variable belongs to the global scope:

def myfunc():  
  global x  
  x = 300  
  
myfunc()  
  
print(x)

Also, use the global keyword if you want to make a change to a global variable inside a function.

### Example

To change the value of a global variable inside a function, refer to the variable by using the global keyword:

x = 300  
  
def myfunc():  
  global x  
  x = 200  
  
myfunc()  
  
print(x)

* Function documentation:-

## Documenting Your Python Code Base Using Docstrings[#](https://realpython.com/documenting-python-code/#documenting-your-python-code-base-using-docstrings)

Now that we’ve learned about commenting, let’s take a deep dive into documenting a Python code base. In this section, you’ll learn about docstrings and how to use them for documentation. This section is further divided into the following sub-sections:

1. **Docstrings background :** A background on how docstrings work internally within Python
2. **Docstring backtype :** The various docstring “types” (function, class, class method, module, package, and script)
3. **Docstrings formats :** The different docstring “formats” (Google, NumPy/SciPy, reStructured Text, and Epytext)

### Docstrings Background

Documenting your Python code is all centered on docstrings. These are built-in strings that, when configured correctly, can help your users and yourself with your project’s documentation. Along with docstrings, Python also has the built-in function help() that prints out the objects docstring to the console. Here’s a quick example:

>>>

>>> help(str)

Help on class str in module builtins:

class str(object)

| str(object='') -> str

| str(bytes\_or\_buffer[, encoding[, errors]]) -> str

|

| Create a new string object from the given object. If encoding or

| errors are specified, then the object must expose a data buffer

| that will be decoded using the given encoding and error handler.

| Otherwise, returns the result of object.\_\_str\_\_() (if defined)

| or repr(object).

| encoding defaults to sys.getdefaultencoding().

| errors defaults to 'strict'.

# Truncated for readability

How is this output generated? Since everything in Python is an object, you can examine the directory of the object using the dir() command. Let’s do that and see what find:

>>>

>>> dir(str)

['\_\_add\_\_', ..., '\_\_doc\_\_', ..., 'zfill'] # Truncated for readability

Within that directory output, there’s an interesting property, \_\_doc\_\_. If you examine that property, you’ll discover this:

>>>

>>> print(str.\_\_doc\_\_)

str(object='') -> str

str(bytes\_or\_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or

errors are specified, then the object must expose a data buffer

that will be decoded using the given encoding and error handler.

Otherwise, returns the result of object.\_\_str\_\_() (if defined)

or repr(object).

encoding defaults to sys.getdefaultencoding().

errors defaults to 'strict'.

Voilà! You’ve found where docstrings are stored within the object. This means that you can directly manipulate that property. However, there are restrictions for builtins:

>>>

>>> str.\_\_doc\_\_ = "I'm a little string doc! Short and stout; here is my input and print me for my out"

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: can't set attributes of built-in/extension type 'str'

Any other custom object can be manipulated:

def say\_hello(name):

print(f"Hello {name}, is it me you're looking for?")

say\_hello.\_\_doc\_\_ = "A simple function that says hello... Richie style"

>>>

>>> help(say\_hello)

Help on function say\_hello in module \_\_main\_\_:

say\_hello(name)

A simple function that says hello... Richie style

Python has one more feature that simplifies docstring creation. Instead of directly manipulating the \_\_doc\_\_ property, the strategic placement of the string literal directly below the object will automatically set the \_\_doc\_\_ value. Here’s what happens with the same example as above:

def say\_hello(name):

"""A simple function that says hello... Richie style"""

print(f"Hello {name}, is it me you're looking for?")

>>>

>>> help(say\_hello)

Help on function say\_hello in module \_\_main\_\_:

say\_hello(name)

A simple function that says hello... Richie style

There you go! Now you understand the background of docstrings. Now it’s time to learn about the different types of docstrings and what information they should contain.

[Remove ads](https://realpython.com/account/join/)

### Docstring Types

Docstring conventions are described within PEP 257. Their purpose is to provide your users with a brief overview of the object. They should be kept concise enough to be easy to maintain but still be elaborate enough for new users to understand their purpose and how to use the documented object.

In all cases, the docstrings should use the triple-double quote (""") string format. This should be done whether the docstring is multi-lined or not. At a bare minimum, a docstring should be a quick summary of whatever is it you’re describing and should be contained within a single line:

"""This is a quick summary line used as a description of the object."""

Multi-lined docstrings are used to further elaborate on the object beyond the summary. All multi-lined docstrings have the following parts:

* A one-line summary line
* A blank line proceeding the summary
* Any further elaboration for the docstring
* Another blank line

"""This is the summary line

This is the further elaboration of the docstring. Within this section,

you can elaborate further on details as appropriate for the situation.

Notice that the summary and the elaboration is separated by a blank new

line.

"""

# Notice the blank line above. Code should continue on this line.

All docstrings should have the same max character length as comments (72 characters). Docstrings can be further broken up into three major categories:

* **Class Docstrings:** Class and class methods
* **Package and Module Docstrings:** Package, modules, and functions
* **Script Docstrings:** Script and functions

#### **Class Docstrings**

Class Docstrings are created for the class itself, as well as any class methods. The docstrings are placed immediately following the class or class method indented by one level:

class SimpleClass:

"""Class docstrings go here."""

def say\_hello(self, name: str):

"""Class method docstrings go here."""

print(f'Hello {name}')

Class docstrings should contain the following information:

* A brief summary of its purpose and behavior
* Any public methods, along with a brief description
* Any class properties (attributes)
* Anything related to the interface  for subclassers, if the class is intended to be subclassed

The class constructor parameters should be documented within the \_\_init\_\_ class method docstring. Individual methods should be documented using their individual docstrings. Class method docstrings should contain the following:

* A brief description of what the method is and what it’s used for
* Any arguments (both required and optional) that are passed including keyword arguments
* Label any arguments that are considered optional or have a default value
* Any side effects that occur when executing the method
* Any exceptions that are raised
* Any restrictions on when the method can be called

Let’s take a simple example of a data class that represents an Animal. This class will contain a few class properties, instance properties, a \_\_init\_\_, and a single instance method:

class Animal:

"""

A class used to represent an Animal

...

Attributes

----------

says\_str : str

a formatted string to print out what the animal says

name : str

the name of the animal

sound : str

the sound that the animal makes

num\_legs : int

the number of legs the animal has (default 4)

Methods

-------

says(sound=None)

Prints the animals name and what sound it makes

"""

says\_str = "A {name} says {sound}"

def \_\_init\_\_(self, name, sound, num\_legs=4):

"""

Parameters

----------

name : str

The name of the animal

sound : str

The sound the animal makes

num\_legs : int, optional

The number of legs the animal (default is 4)

"""

self.name = name

self.sound = sound

self.num\_legs = num\_legs

def says(self, sound=None):

"""Prints what the animals name is and what sound it makes.

If the argument `sound` isn't passed in, the default Animal

sound is used.

Parameters

----------

sound : str, optional

The sound the animal makes (default is None)

Raises

------

NotImplementedError

If no sound is set for the animal or passed in as a

parameter.

"""

if self.sound is None and sound is None:

raise NotImplementedError("Silent Animals are not supported!")

out\_sound = self.sound if sound is None else sound

print(self.says\_str.format(name=self.name, sound=out\_sound))

#### **Package and Module Docstrings**

Package docstrings should be placed at the top of the package’s \_\_init\_\_.py file. This docstring should list the modules and sub-packages that are exported by the package.

Module docstrings are similar to class docstrings. Instead of classes and class methods being documented, it’s now the module and any functions found within. Module docstrings are placed at the top of the file even before any imports. Module docstrings should include the following:

* A brief description of the module and its purpose
* A list of any classes, exception, functions, and any other objects exported by the module

The docstring for a module function should include the same items as a class method:

* A brief description of what the function is and what it’s used for
* Any arguments (both required and optional) that are passed including keyword arguments
* Label any arguments that are considered optional
* Any side effects that occur when executing the function
* Any exceptions that are raised
* Any restrictions on when the function can be called

#### **Script Docstrings**

Scripts are considered to be single file executables run from the console. Docstrings for scripts are placed at the top of the file and should be documented well enough for users to be able to have a sufficient understanding of how to use the script. It should be usable for its “usage” message, when the user incorrectly passes in a parameter or uses the -h option.

If you use argparse, then you can omit parameter-specific documentation, assuming it’s correctly been documented within the help parameter of the argparser.parser.add\_argument function. It is recommended to use the \_\_doc\_\_ for the description parameter within argparse.ArgumentParser’s constructor. Check out our tutorial on Command-line parsing  for more details on how to use argparse and other common command line parsers.

Finally, any custom or third-party imports should be listed within the docstrings to allow users to know which packages may be required for running the script. Here’s an example of a script that is used to simply print out the column headers of a spreadsheet:

"""Spreadsheet Column Printer

This script allows the user to print to the console all columns in the

spreadsheet. It is assumed that the first row of the spreadsheet is the

location of the columns.

This tool accepts comma separated value files (.csv) as well as excel

(.xls, .xlsx) files.

This script requires that `pandas` be installed within the Python

environment you are running this script in.

This file can also be imported as a module and contains the following

functions:

\* get\_spreadsheet\_cols - returns the column headers of the file

\* main - the main function of the script

"""

import argparse

import pandas as pd

def get\_spreadsheet\_cols(file\_loc, print\_cols=False):

"""Gets and prints the spreadsheet's header columns

Parameters

----------

file\_loc : str

The file location of the spreadsheet

print\_cols : bool, optional

A flag used to print the columns to the console (default is

False)

Returns

-------

list

a list of strings used that are the header columns

"""

file\_data = pd.read\_excel(file\_loc)

col\_headers = list(file\_data.columns.values)

if print\_cols:

print("\n".join(col\_headers))

return col\_headers

def main():

parser = argparse.ArgumentParser(description=\_\_doc\_\_)

parser.add\_argument(

'input\_file',

type=str,

help="The spreadsheet file to pring the columns of"

)

args = parser.parse\_args()

get\_spreadsheet\_cols(args.input\_file, print\_cols=True)

if \_\_name\_\_ == "\_\_main\_\_":

main()

### Docstring Formats

You may have noticed that, throughout the examples given in this tutorial, there has been specific formatting with common elements: Arguments, Returns, and Attributes. There are specific docstrings formats that can be used to help docstring parsers and users have a familiar and known format. The formatting used within the examples in this tutorial are NumPy/SciPy-style docstrings. Some of the most common formats are the following:

| **Formatting Type** | **Description** | **Supported by Sphynx** | **Formal Specification** |
| --- | --- | --- | --- |
| Google docstring | Google’s recommended form of documentation | Yes | No |
| Restructured docstrings | Official Python documentation standard; Not beginner friendly but feature rich | Yes | Yes |
| numPy\sciPY DOCSTRINGS | NumPy’s combination of reStructured and Google Docstrings | Yes | Yes |
| Epytext | A Python adaptation of Epydoc; Great for Java developers | Not officially | Yes |

The selection of the docstring format is up to you, but you should stick with the same format throughout your document/project. The following are examples of each type to give you an idea of how each documentation format looks.

#### **Google Docstrings Example**

"""Gets and prints the spreadsheet's header columns

Args:

file\_loc (str): The file location of the spreadsheet

print\_cols (bool): A flag used to print the columns to the console

(default is False)

Returns:

list: a list of strings representing the header columns

"""

#### **reStructured Text Example**

"""Gets and prints the spreadsheet's header columns

:param file\_loc: The file location of the spreadsheet

:type file\_loc: str

:param print\_cols: A flag used to print the columns to the console

(default is False)

:type print\_cols: bool

:returns: a list of strings representing the header columns

:rtype: list

"""

#### **NumPy/SciPy Docstrings Example**

"""Gets and prints the spreadsheet's header columns

Parameters

----------

file\_loc : str

The file location of the spreadsheet

print\_cols : bool, optional

A flag used to print the columns to the console (default is False)

Returns

-------

list

a list of strings representing the header columns

"""

#### **Epytext Example**

"""Gets and prints the spreadsheet's header columns

@type file\_loc: str

@param file\_loc: The file location of the spreadsheet

@type print\_cols: bool

@param print\_cols: A flag used to print the columns to the console

(default is False)

@rtype: list

@returns: a list of strings representing the header columns

"""

## Documenting Your Python Projects

Python projects come in all sorts of shapes, sizes, and purposes. The way you document your project should suit your specific situation. Keep in mind who the users of your project are going to be and adapt to their needs. Depending on the project type, certain aspects of documentation are recommended. The general layout of the project and its documentation should be as follows:

project\_root/

│

├── project/ # Project source code

├── docs/

├── README

├── HOW\_TO\_CONTRIBUTE

├── CODE\_OF\_CONDUCT

├── examples.py

Projects can be generally subdivided into three major types: Private, Shared, and Public/Open Source.

### Private Projects

Private projects are projects intended for personal use only and generally aren’t shared with other users or developers. Documentation can be pretty light on these types of projects. There are some recommended parts to add as needed:

* **Readme:** A brief summary of the project and its purpose. Include any special requirements for installation or operating the project.
* **examples.py:** A Python script file that gives simple examples of how to use the project.

Remember, even though private projects are intended for you personally, you are also considered a user. Think about anything that may be confusing to you down the road and make sure to capture those in either comments, docstrings, or the readme.

### Shared Projects

Shared projects are projects in which you collaborate with a few other people in the development and/or use of the project. The “customer” or user of the project continues to be yourself and those limited few that use the project as well.

Documentation should be a little more rigorous than it needs to be for a private project, mainly to help onboard new members to the project or alert contributors/users of new changes to the project. Some of the recommended parts to add to the project are the following:

* **Readme:** A brief summary of the project and its purpose. Include any special requirements for installing or operating the project. Additionally, add any major changes since the previous version.
* **examples.py:** A Python script file that gives simple examples of how to use the projects.
* **How to Contribute:** This should include how new contributors to the project can start contributing.

### Public and Open Source Projects

Public and Open Source projects are projects that are intended to be shared with a large group of users and can involve large development teams. These projects should place as high of a priority on project documentation as the actual development of the project itself. Some of the recommended parts to add to the project are the following:

* **Readme:** A brief summary of the project and its purpose. Include any special requirements for installing or operating the projects. Additionally, add any major changes since the previous version. Finally, add links to further documentation, bug reporting, and any other important information for the project. Dan Bader has put together a great tutorial on what all should be included in your readme.
* **How to Contribute:** This should include how new contributors to the project can help. This includes developing new features, fixing known issues, adding documentation, adding new tests, or reporting issues.
* **Code of Conduct:** Defines how other contributors should treat each other when developing or using your software. This also states what will happen if this code is broken. If you’re using Github, a Code of Conduct template can be generated with recommended wording. For Open Source projects especially, consider adding this.
* **License:** A plaintext file that describes the license your project is using. For Open Source projects especially, consider adding this.
* **docs:** A folder that contains further documentation. The next section describes more fully what should be included and how to organize the contents of this folder.

#### **The Four Main Sections of the docs Folder**

Daniele Procida gave a wonderful PYcon 207 and subsequent blog post  about documenting Python projects. He mentions that all projects should have the following four major sections to help you focus your work:

* **Tutorials**: Lessons that take the reader by the hand through a series of steps to complete a projects (or meaningful exercise). Geared towards the users learning.
* **How-To Guides**: Guides that take the reader through the steps required to solve a common problem (problem-oriented recipes).
* **References**: Explanations that clarify and illuminate a particular topic. Geared towards understanding.
* **Explanations**: Technical descriptions of the machinery and how to operate it (key classes, functions, APIs, and so forth). Think Encyclopedia article.

The following table shows how all of these sections relates to each other as well as their overall purpose:

|  | **Most Useful When We’re Studying** | **Most Useful When We’re Coding** |
| --- | --- | --- |
| **Practical Step** | Tutorials | How-To Guides |
| **Theoretical Knowledge** | Explanation | Reference |

In the end, you want to make sure that your users have access to the answers to any questions they may have. By organizing your project in this manner, you’ll be able to answer those questions easily and in a format they’ll be able to navigate quickly.

### Documentation Tools and Resources

Documenting your code, especially large projects, can be daunting. Thankfully there are some tools out and references to get you started:

| **Tool** | **Description** |
| --- | --- |
| sphinx | A collection of tools to auto-generate documentation in multiple formats |
| epydoc | A tool for generating API documentation for Python modules based on their docstrings |
| Read the docs | Automatic building, versioning, and hosting of your docs for you |
| Doxygen | A tool for generating documentation that supports Python as well as multiple other languages |
| Mkdocs | A static site generator to help build project documentation using the Markdown language |
| pycco | A “quick and dirty” documentation generator that displays code and documentation side by side.. |

. Here are some great examples of projects that use documentation well:

* **Django:**
* **Requests:**
* **Click:**
* **Pandas:**

## Where Do I Start?

The documentation of projects have a simple progression:

1. No Documentation
2. Some Documentation
3. Complete Documentation
4. Good Documentation
5. Great Documentation

If you’re at a loss about where to go next with your documentation, look at where your project is now in relation to the progression above. Do you have any documentation? If not, then start there. If you have some documentation but are missing some of the key project files, get started by adding those.

In the end, don’t get discouraged or overwhelmed by the amount of work required for documenting code. Once you get started documenting your code, it becomes easier to keep going. Feel free to comment if you have questions or reach out to the Real Python Team on social media, and we’ll help.

* Lambda functions and maps:-

# **Python lambda (Anonymous Functions) | filter, map, reduce**

Last Updated: 20-10-2018

In Python, anonymous function means that a function is without a name. As we already know that def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions. It has the following syntax:

**lambda arguments: expression**

* This function can have any number of arguments but only one expression, which is evaluated and returned.
* One is free to use lambda functions wherever function objects are required.
* You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
* It has various uses in particular fields of programming besides other types of expressions in functions.

Let’s look at this example and try to understand the **difference between a normal def defined function and lambda function**. This is a program that returns the cube of a given value:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to illustrate cube of a number  # showing difference between def() and lambda().  def cube(y):      return y\*y\*y;    g = lambda x: x\*x\*x  print(g(7))    print(cube(5)) |

Output:

343

125

* **Without using Lambda :** Here, both of them returns the cube of a given number. But, while using def, we needed to define a function with a name cube and needed to pass a value to it. After execution, we also needed to return the result from where the function was called using the return keyword.
* **Using Lambda :** Lambda definition does not include a “return” statement, it always contains an expression which is returned. We can also put a lambda definition anywhere a function is expected, and we don’t have to assign it to a variable at all. This is the simplicity of lambda functions.

Lambda functions can be used along with built-in functions like filter(), map() and reduce().

**Use of lambda() with filter()**

The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence “sequence”, for which the function returns True. Here is a small program that returns the odd numbers from an input list:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to illustrate  # filter() with lambda()  li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]  final\_list = list(filter(lambda x: (x%2 != 0) , li))  print(final\_list) |

Output:

[5, 7, 97, 77, 23, 73, 61]

**Use of lambda() with map()**

The map() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item. Example:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to illustrate  # map() with lambda()  # to get double of a list.  li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]  final\_list = list(map(lambda x: x\*2 , li))  print(final\_list) |

Output:

[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]

**Use of lambda() with reduce()**

The reduce() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new reduced result is returned. This performs a repetitive operation over the pairs of the list. This is a part of functools module. Example:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to illustrate  # reduce() with lambda()  # to get sum of a list  from functools import reduce  li = [5, 8, 10, 20, 50, 100]  sum = reduce((lambda x, y: x + y), li)  print (sum) |

Output:

193

Here the results of previous two elements are added to the next element and this goes on till the end of the list like (((((5+8)+10)+20)+50)+100).

iven function

to each item of a given iterable (list, tuple etc.)

**NOTE :** The returned value from map() (map object) then can be passed to functions like list() (to create a list), set() (to create a set) .  
   
**CODE 1**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python program to demonstrate working  # of map.    # Return double of n  def addition(n):      return n + n    # We double all numbers using map()  numbers = (1, 2, 3, 4)  result = map(addition, numbers)  print(list(result)) |

Output :

[2, 4, 6, 8]

**CODE 2**  
We can also use lambda expressions with map to achieve above result.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Double all numbers using map and lambda    numbers = (1, 2, 3, 4)  result = map(lambda x: x + x, numbers)  print(list(result)) |

Output :

[2, 4, 6, 8]

**CODE 3**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Add two lists using map and lambda    numbers1 = [1, 2, 3]  numbers2 = [4, 5, 6]    result = map(lambda x, y: x + y, numbers1, numbers2)  print(list(result)) |

Output :

[5, 7, 9]

**CODE 4**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # List of strings  l = ['sat', 'bat', 'cat', 'mat']    # map() can listify the list of strings individually  test = list(map(list, l))  print(test) |

Output :

[['s', 'a', 't'], ['b', 'a', 't'], ['c', 'a', 't'], ['m', ':-a', 't']]

* EXERCISE WITH FUNCTION:-

Python function is a code block or group of statements that perform a particular task. We use Python functions for code reuse so we can use it repeatedly in a program. This Python functions exercise aims to help Python developers to **learn and practice** how to create a function, nested functions, and use the function arguments effectively. All questions are tested on **Python 3**.

**What included in this Python functions exercise?**

The exercise contains **10** questions and solutions provided for each question. This Python function **coding** exercise is nothing but Python function assignments to solve, where you can solve and practice different function programs, questions, problems, and challenges.

Each question includes a specific function related topic you need to learn. This function exercise covers questions on **topics**such as function creation, function calls, function arguments, inner functions, built-in functions. Let us know if you have any alternative solutions. It will help other developers.

#### Exercise Question 1: Create a function that can accept two arguments name and age and print its value

**Solution**:

def demo(name, age):

print(name, age)

demo("Ben", 25)

 Copy

#### Exercise Question 2: Write a function func1() such that it can **accept a variable length of  argument**and print all arguments value

func1(20, 40, 60)

func1(80, 100)

**Expected Output**:

After func1(20, 40, 60):

20  
40  
60

After func1(80, 100):

80  
100

**Solution**:

def func1(\*args):

for i in args:

print(i)

func1(20, 40, 60)

func1(80, 100)

 Copy

**Explanation**:

To **accept Variable Length of Positional Arguments**, i.e., To create functions that take n number of Positional arguments we use \*args(prefix a parameter name with an asterisk \* ).

#### Exercise Question 3: Write a function calculation() such that it can accept two variables and calculate the addition and subtraction of it. And also it must return both addition and subtraction in a single return call

For example:

def calculation(a, b):

# Your Code

res = calculation(40, 10)

print(res)

A res should produce result 50, 30

**Expected Output**:

**Solution**:

In Python, **we can return multiple values from a function**. You can do this by separating return values with a comma.

def calculation(a, b):

return a+b, a-b

res = calculation(40, 10)

print(res)

 Copy

**Or**

def calculation(a, b):

return a+b, a-b

add, sub = calculation(40, 10)

print(add)

print(sub)

 Copy

#### Exercise Question 4: Create a function showEmployee() in such a way that it should accept employee name, and it’s salary and display both, and if the salary is missing in function call it should show it as 9000

**Expected Output**:

showEmployee("Ben", 9000)

showEmployee("Ben")

Should Produce:

Employee Ben salary is: 9000  
Employee Ben salary is: 9000

**Solution**:

In Python, we can **specify default values** for arguments when defining a function.

def showEmployee(name, salary=9000):

print("Employee", name, "salary is:", salary)

showEmployee("Ben", 9000)

showEmployee("Ben")

 Copy

#### Exercise Question 5: Create an inner function to calculate the addition in the following way

* Create an outer function that will accept two parameters a and b
* Create an inner function inside an outer function that will calculate the addition of a and b
* At last, an outer function will add 5 into addition and return it

**Solution**:

In Python, we can create a nested function inside a function. We can use the nested function to perform complex tasks multiple times within another function or avoid loop and code duplication.

def outerFun(a, b):

square = a\*\*2

def innerFun(a,b):

return a+b

add = innerFun(a, b)

return add+5

result = outerFun(5, 10)

print(result)

 Copy

#### Exercise Question 6: Write a recursive function to calculate the sum of numbers from 0 to 10

**Expected Output**:

55

**Solution**:

def calculateSum(num):

if num:

return num + calculateSum(num-1)

else:

return 0

res = calculateSum(10)

print(res)

 Copy

#### Exercise Question 7: Assign a different name to function and call it through the new name

Below is the function displayStudent(name, age). Assign a new name showStudent(name, age)  to it and call through the new name

def displayStudent(name, age):

print(name, age)

displayStudent("Emma", 26)

You should be able to call the same function using

showStudent(name, age)

**Solution**:

def displayStudent(name, age):

print(name, age)

displayStudent("Emma", 26)

showStudent = displayStudent

showStudent("Emma", 26)

 Copy

#### Exercise Question 8: Generate a Python list of all the even numbers between 4 to 30

**Expected Output**:

[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]

**Solution**:

Use built-in function range().

print(list( range(4, 30, 2)))

 Copy

#### Exercise Question 9: Return the largest item from the given list

aList = [4, 6, 8, 24, 12, 2]

**Expected Output**:

24

**Solution**:

aList = [4, 6, 8, 24, 12, 2]

print(max(aList))

* CREATE A MODULE:-

Modules refer to a file containing Python statements and definitions.

A file containing Python code, for example: example.py, is called a module, and its module name would be example.

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as example.py.

# Python Module example

def add(a, b):

"""This program adds two

numbers and return the result"""

result = a + b

return result

Here, we have defined a FUNCTION  add() inside a module named example. The function takes in two numbers and returns their sum.

## How to import modules in Python?

We can import the definitions inside a module to another module or the interactive interpreter in Python.

We use the import keyword to do this. To import our previously defined module example, we type the following in the Python prompt.

>>> import example

This does not import the names of the functions defined in example directly in the current symbol table. It only imports the module name example there.

Using the module name we can access the function using the dot . operator. For example:

>>> example.add(4,5.5)

9.5

Python has tons of standard modules. You can check out the full list of PYTHON STANDERD MODULES and their use cases. These files are in the Lib directory inside the location where you installed Python.

Standard modules can be imported the same way as we import our user-defined modules.

There are various ways to import modules. They are listed below..

### Python import statement

We can import a module using the import statement and access the definitions inside it using the dot operator as described above. Here is an example.

# import statement example

# to import standard module math

import math

print("The value of pi is", math.pi)

When you run the program, the output will be:

The value of pi is 3.141592653589793

### Import with renaming

We can import a module by renaming it as follows:

# import module by renaming it

import math as m

print("The value of pi is", m.pi)

We have renamed the math module as m. This can save us typing time in some cases.

Note that the name math is not recognized in our scope. Hence, math.pi is invalid, and m.pi is the correct implementation.

### Python from...import statement

We can import specific names from a module without importing the module as a whole. Here is an example.

# import only pi from math module

from math import pi

print("The value of pi is", pi)

Here, we imported only the pi attribute from the math module.

In such cases, we don't use the dot operator. We can also import multiple attributes as follows:

>>> from math import pi, e

>>> pi

3.141592653589793

>>> e

2.718281828459045

### Import all names

We can import all names(definitions) from a module using the following construct:

# import all names from the standard module math

from math import \*

print("The value of pi is", pi)

Here, we have imported all the definitions from the math module. This includes all names visible in our scope except those beginning with an underscore(private definitions).

Importing everything with the asterisk (\*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

## Python Module Search Path

While importing a module, Python looks at several places. Interpreter first looks for a built-in module. Then(if built-in module not found), Python looks into a list of directories defined in sys.path. The search is in this order.

* The current directory.
* PYTHONPATH (an environment variable with a list of directories).
* The installation-dependent default directory.

>>> import sys

>>> sys.path

['',

'C:\\Python33\\Lib\\idlelib',

'C:\\Windows\\system32\\python33.zip',

'C:\\Python33\\DLLs',

'C:\\Python33\\lib',

'C:\\Python33',

'C:\\Python33\\lib\\site-packages']

We can add and modify this list to add our own path.

## Reloading a module

The Python interpreter imports a module only once during a session. This makes things more efficient. Here is an example to show how this works.

Suppose we have the following code in a module named my\_module.

# This module shows the effect of

# multiple imports and reload

print("This code got executed")

Now we see the effect of multiple imports.

>>> import my\_module

This code got executed

>>> import my\_module

>>> import my\_module

We can see that our code got executed only once. This goes to say that our module was imported only once.

Now if our module changed during the course of the program, we would have to reload it.One way to do this is to restart the interpreter. But this does not help much.

Python provides a more efficient way of doing this. We can use the reload() function inside the imp module to reload a module. We can do it in the following ways:

>>> import imp

>>> import my\_module

This code got executed

>>> import my\_module

>>> imp.reload(my\_module)

This code got executed

<module 'my\_module' from '.\\my\_module.py'>

## The dir() built-in function

We can use the dir() function to find out names that are defined inside a module.

For example, we have defined a function add() in the module example that we had in the beginning.

We can use dir in example module in the following way:

>>> dir(example)

['\_\_builtins\_\_',

'\_\_cached\_\_',

'\_\_doc\_\_',

'\_\_file\_\_',

'\_\_initializing\_\_',

'\_\_loader\_\_',

'\_\_name\_\_',

'\_\_package\_\_',

'add']

Here, we can see a sorted list of names (along with add). All other names that begin with an underscore are default Python attributes associated with the module (not user-defined).

For example, the \_\_name\_\_ attribute contains the name of the module.

>>> import example

>>> example.\_\_name\_\_

'example'

All the names defined in our current namespace can be found out using the dir() function without any arguments.

>>> a = 1

>>> b = "hello"

>>> import math

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', 'a', 'b', 'math', 'pyscripter']

* STANDERD MODULES:-
* Python comes with a library of standard modules, described in a separate document, the Python Library Reference (“Library Reference” hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the [winreg](https://docs.python.org/3/library/winreg.html" \l "module-winreg" \o "winreg: Routines and objects for manipulating the Windows registry. (Windows)) module is only provided on Windows systems. One particular module deserves some attention: [sys](https://docs.python.org/3/library/sys.html#module-sys), which is built into every Python interpreter. The variables sys.ps1 and sys.ps2 define the strings used as primary and secondary prompts:
* >>>
* **>>> import** **sys**
* **>>>** sys.ps1
* '>>> '
* **>>>** sys.ps2
* '... '
* **>>>** sys.ps1 = 'C> '
* C> print('Yuck!')
* Yuck!
* C>
* These two variables are only defined if the interpreter is in interactive mode.
* The variable sys.path is a list of strings that determines the interpreter’s search path for modules. It is initialized to a default path taken from the environment variable [PYTHONPATH](https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH), or from a built-in default if [PYTHONPATH](https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH) is not set. You can modify it using standard list operations:
* >>>
* **>>> import** **sys**
* **>>>** sys.path.append('/ufs/guido/lib/python')

MODULE 5:- EXCEPTIONS HANDLING

* ERRORS:-

## Python Logical Errors (Exceptions)

Errors that occur at runtime (after passing the syntax test) are called **exceptions** or **logical errors**.

For instance, they occur when we try to open a file(for reading) that does not exist (FileNotFoundError), try to divide a number by zero (ZeroDivisionError), or try to import a module that does not exist (ImportError).

Whenever these types of runtime errors occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Let's look at how Python treats these errors:

>>> 1 / 0

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

ZeroDivisionError: division by zero

>>> open("imaginary.txt")

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'

* EXCEPTION HANDLING HANDLING WITH TRY:-

# Python Exception Handling Using try, except and finally statement

## Exceptions in Python

Python has many built in exceptions that are raised when your program encounters an error (something in the program goes wrong).

When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.

For example, let us consider a program where we have a function  A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.

If never handled, an error message is displayed and our program comes to a sudden unexpected halt.

## Catching Exceptions in Python

In Python, exceptions can be handled using a try statement.

The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.

We can thus choose what operations to perform once we have caught the exception. Here is a simple example.

# import module sys to get the type of exception

import sys

randomList = ['a', 0, 2]

for entry in randomList:

try:

print("The entry is", entry)

r = 1/int(entry)

break

except:

print("Oops!", sys.exc\_info()[0], "occurred.")

print("Next entry.")

print()

print("The reciprocal of", entry, "is", r)

**Output**

The entry is a

Oops! <class 'ValueError'> occurred.

Next entry.

The entry is 0

Oops! <class 'ZeroDivisionError'> occured.

Next entry.

The entry is 2

The reciprocal of 2 is 0.5

In this program, we loop through the values of the randomList list. As previously mentioned, the portion that can cause an exception is placed inside the try block.

If no exception occurs, the except block is skipped and normal flow continues(for last value). But if any exception occurs, it is caught by the except block (first and second values).

Here, we print the name of the exception using the exc\_info() function inside sys module. We can see that a causes ValueError and 0 causes ZeroDivisionError.

Since every exception in Python inherits from the base Exception class, we can also perform the above task in the following way:

# import module sys to get the type of exception

import sys

randomList = ['a', 0, 2]

for entry in randomList:

try:

print("The entry is", entry)

r = 1/int(entry)

break

except Exception as e:

print("Oops!", e.\_\_class\_\_, "occurred.")

print("Next entry.")

print()

print("The reciprocal of", entry, "is", r)

This program has the same output as the above program.

## Catching Specific Exceptions in Python

In the above example, we did not mention any specific exception in the except clause.

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause should catch.

A try clause can have any number of except clauses to handle different exceptions, however, only one will be executed in case an exception occurs.

We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

try:

# do something

pass

except ValueError:

# handle ValueError exception

pass

except (TypeError, ZeroDivisionError):

# handle multiple exceptions

# TypeError and ZeroDivisionError

pass

except:

# handle all other exceptions

pass

## Raising Exceptions in Python

In Python programming, exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the raise keyword.

We can optionally pass values to the exception to clarify why that exception was raised.

>>> raise KeyboardInterrupt

Traceback (most recent call last):

...

KeyboardInterrupt

>>> raise MemoryError("This is an argument")

Traceback (most recent call last):

...

MemoryError: This is an argument

>>> try:

... a = int(input("Enter a positive integer: "))

... if a <= 0:

... raise ValueError("That is not a positive number!")

... except ValueError as ve:

... print(ve)

...

Enter a positive integer: -2

That is not a positive number!

## Python try with else clause

In some situations, you might want to run a certain block of code if the code block inside try ran without any errors. For these cases, you can use the optional else keyword with the try statement.

**Note**: Exceptions in the else clause are not handled by the preceding except clauses.

Let's look at an example:

# program to print the reciprocal of even numbers

try:

num = int(input("Enter a number: "))

assert num % 2 == 0

except:

print("Not an even number!")

else:

reciprocal = 1/num

print(reciprocal)

**Output**

If we pass an odd number:

Enter a number: 1

Not an even number!

If we pass an even number, the reciprocal is computed and displayed.

Enter a number: 4

0.25

However, if we pass 0, we get ZeroDivisionError as the code block inside else is not handled by preceding except.

Enter a number: 0

Traceback (most recent call last):

File "<string>", line 7, in <module>

reciprocal = 1/num

ZeroDivisionError: division by zero

## Python try...finally

The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.

For example, we may be connected to a remote data center through the network or working with a file or a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resource before the program comes to a halt whether it successfully ran or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee the execution.

Here is an example of file operations  to illustrate this.

try:

f = open("test.txt",encoding = 'utf-8')

# perform file operations

finally:

f.close()

* multiple exceptional handling :-

# **Multiple Exception Handling in Python**

Last Updated: 12-06-2019

Given a piece of code that can throw any of several different exceptions, and one needs to account for all of the potential exceptions that could be raised without creating duplicate code or long, meandering code passages.

If you can handle different exceptions all using a single block of code, they can be grouped together in a tuple as shown in the code given below :

**Code #1 :**

filter\_none

brightness\_4

|  |
| --- |
| try:      client\_obj.get\_url(url)  except (URLError, ValueError, SocketTimeout):      client\_obj.remove\_url(url) |

The remove\_url() method will be called if any of the listed exceptions occurs. If, on the other hand, if one of the exceptions has to be handled differently, then put it into its own except clause as shown in the code given below :

**Code #2 :**

filter\_none

brightness\_4

|  |
| --- |
| try:      client\_obj.get\_url(url)  except (URLError, ValueError):      client\_obj.remove\_url(url)  except SocketTimeout:      client\_obj.handle\_url\_timeout(url) |

Many exceptions are grouped into an inheritance hierarchy. For such exceptions, all of the exceptions can be caught by simply specifying a base class. For example, instead of writing code as shown in the code given below –

**Code #3 :**

filter\_none

brightness\_4

|  |
| --- |
| try:      f = open(filename)  except (FileNotFoundError, PermissionError):      ... |

Except statement can be re-written as in the code given below. This works because OSError is a base class that’s common to both the FileNotFoundError and **PermissionError** exceptions.

**Code #4 :**

filter\_none

brightness\_4

|  |
| --- |
| try:      f = open(filename)  except OSError:      ... |

Although it’s not specific to handle multiple exceptions per **se**, it is worth noting that one can get a handle to the thrown exception using them as a keyword as shown in the code given below.

**Code #5 :**

filter\_none

brightness\_4

|  |
| --- |
| try:      f = open(filename)    except OSError as e:      if e.errno == errno.ENOENT:          logger.error('File not found')      elif e.errno == errno.EACCES:          logger.error('Permission denied')      else:          logger.error('Unexpected error: % d', e.errno) |

The **e** variable holds an instance of the raised OSError. This is useful if the exception has to be invested further, such as processing it based on the value of the additional status code. The except clauses are checked in the order listed and the first match executes.

**Code #6 : Create situations where multiple except clauses might match**

filter\_none

brightness\_4

|  |
| --- |
| f = open('missing') |

**Output :**

Traceback (most recent call last):

File "", line 1, in

FileNotFoundError: [Errno 2] No such file or directory: 'miss

filter\_none

brightness\_4

|  |
| --- |
| try:      f = open('missing')      except OSError:          print('It failed')      except FileNotFoundError:          print('File not found') |

**Output :**

Failed

Here the except FileNotFoundError clause doesn’t execute because the OSError is more general, matches the FileNotFoundError exception, and was listed first.

* Writing own exception :-

By looking at the names of the exceptions to see if its meaning is appropriate or not. For example, the IllegalArgumentException is appropriate to throw when checking parameters of a method; the IOException is appropriate to throw when reading/writing files.

From my experience, most of the cases we need custom exceptions for representing business exceptions which are, at a level higher than technical exceptions defined by JDK. For example: InvalidAgeException, LowScoreException, TooManyStudentsException, etc.

Now, let’s see how to create a custom exception in action. Here are the steps:

* Create a new class whose name should end with Exception like ClassNameException. This is a convention to differentiate an exception class from regular ones.
* Make the class extends one of the exceptions which are subtypes of the java.lang.Exception class. Generally, a custom exception class always extends directly from the Exception class.
* Create a constructor with a String parameter which is the detail message of the exception. In this constructor, simply call the super constructor and pass the message.

That’s it. The following is a custom exception class which is created by following the above steps:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class StudentNotFoundException extends Exception {        public StudentNotFoundException(String message) {          super(message);      }  } |

And the following example shows the way a custom exception is used is nothing different than built-in exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class StudentManager {        public Student find(String studentID) throws StudentNotFoundException {          if (studentID.equals("123456")) {              return new Student();          } else {              throw new StudentNotFoundException(                  "Could not find student with ID " + studentID);          }      }  } |

And the following test program handles that exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class StudentTest {      public static void main(String[] args) {          StudentManager manager = new StudentManager();            try {                Student student = manager.find("0000001");            } catch (StudentNotFoundException ex) {              System.err.print(ex);          }      }  } |

Run this program and you will see this output:

|  |  |
| --- | --- |
| 1 | StudentNotFoundException: Could not find student with ID 0000001 |

**Re-throwing an exception which is wrapped in a custom exception**

It’s a common practice for catching a built-in exception and re-throwing it via a custom exception. To do so, let add a new constructor to our custom exception class. This constructor takes two parameters: the detail message and the cause of the exception. This constructor is implemented in the Exception class as following:

**public Exception(String message, Throwable cause)**

Besides the detail message, this constructor takes a Throwable’s subclass which is the origin (cause) of the current exception. For example, create the StudentStoreException class as following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class StudentStoreException extends Exception {        public StudentStoreException(String message, Throwable cause) {          super(message, cause);      }  } |

And the following example shows where the StudentStoreException gets thrown:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public void save(Student student) throws StudentStoreException {      try {          // execute SQL statements..      } catch (SQLException ex) {          throw new StudentStoreException("Failed to save student", ex);      }  } |

Here, suppose that the save() method stores the specified student information into a database using JDBC. The code can throw SQLException. We catch this exception and throw a new StudentStoreException which wraps the SQLException as its cause. And it’s obvious that the save() method declares to throw StudentStoreException instead of SQLException.

So what is the benefit of re-throwing exception like this?

Why not leave the original exception to be thrown?

Well, the main benefit of re-throwing exception by this manner is adding a higher abstraction layer for the exception handling, which results in more meaningful and readable API. Do you see StudentStoreException is more meaningful than SQLException, don’t you?

However, remember to include the original exception in order to preserve the cause so we won’t lose the trace when debugging the program when the exception occurred.

And the following code demonstrates handling the StudentStoreException above:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | StudentManager manager = new StudentManager();    try {        manager.save(new Student());    } catch (StudentStoreException ex) {      System.err.print(ex);  } |

Module 6 :-File handling

# File Handling modes:-

Last Updated: 13-11-2018

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but alike other concepts of Python, this concept here is also easy and short. Python treats file differently as text or binary and this is important. Each line of code includes a sequence of characters and they form text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let’s start with Reading and Writing files.

**Working of open() function**

We use **open ()** function in Python to open a file in read or write mode. As explained above, open ( ) will return a file object. To return a file object we use **open()** function along with two arguments, that accepts file name and the mode, whether to read or write. So, the syntax being: **open(filename, mode)**. There are three kinds of mode, that Python provides and how files can be opened:

* “**r**“, for reading.
* “**w**“, for writing.
* “**a**“, for appending.
* “**r+**“, for both reading and writing

One must keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be “**r**” by default. Let’s look at this program and try to analyze how the read mode works:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # a file named "geek", will be opened with the reading mode.  file = open('geek.txt', 'r')  # This will print every line one by one in the file  for each in file:      print (each) |

The open command will open the file in the read mode and the for loop will print each line present in the file.

**Working of read() mode**

There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use **file.read()**. The full code would work like this:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to illustrate read() mode  file = open("file.text", "r")  print file.read() |

Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to illustrate read() mode character wise  file = open("file.txt", "r")  print file.read(5) |

**Creating a file using write() mode**

Let’s see how to create a file and how write mode works:  
To manipulate the file, write the following in your Python environment:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to create a file  file = open('geek.txt','w')  file.write("This is the write command")  file.write("It allows us to write in a particular file")  file.close() |

The close() command terminates all the resources in use and frees the system of this particular program.

**Working of append() mode**

Let’s see how the append mode works:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to illustrate append() mode  file = open('geek.txt','a')  file.write("This will add this line")  file.close() |

There are also various other commands in file handling that is used to handle various tasks like:

rstrip(): This function strips each line of a file off spaces from the right-hand side.

lstrip(): This function strips each line of a file off spaces from the left-hand side.

It is designed to provide much cleaner syntax and exceptions handling when you are working with code. That explains why it’s good practice to use them with a statement where applicable. This is helpful because using this method any files opened will be closed automatically after one is done, so auto-cleanup.  
Example:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to illustrate with()  with open("file.txt") as file:      data = file.read()  # do something with data |

**Using write along with with() function**

We can also use write function along with with() function:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to illustrate with() alongwith write()  with open("file.txt", "w") as f:      f.write("Hello World!!!") |

**split() using file handling**

We can also split lines using file handling in Python. This splits the variable when space is encountered. You can also split using any characters as we wish. Here is the code:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to illustrate split() function  with open("file.text", "r") as file:      data = file.readlines()      for line in data:          word = line.split()          print word |

* Reading files:-

There are three ways to read data from a text file.

1. **read() :** Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.

File\_object.read([n])

1. **readline() :** Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.

File\_object.readline([n])

1. **readlines() :** Reads all the lines and return them as each line a string element in a list.

File\_object.readlines()

**Note:**‘\n’ is treated as a special character of two bytes

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Program to show various ways to read and  # write data in a file.  file1 = open("myfile.txt","w")  L = ["This is Delhi \n","This is Paris \n","This is London \n"]    # \n is placed to indicate EOL (End of Line)  file1.write("Hello \n")  file1.writelines(L)  file1.close() #to change file access modes    file1 = open("myfile.txt","r+")    print "Output of Read function is "  print file1.read()  print    # seek(n) takes the file handle to the nth  # bite from the beginning.  file1.seek(0)    print "Output of Readline function is "  print file1.readline()  print    file1.seek(0)    # To show difference between read and readline  print "Output of Read(9) function is "  print file1.read(9)  print    file1.seek(0)    print "Output of Readline(9) function is "  print file1.readline(9)    file1.seek(0)  # readlines function  print "Output of Readlines function is "  print file1.readlines()  print  file1.close() |

Output:

Output of Read function is

Hello

This is Delhi

This is Paris

This is London

Output of Readline function is

Hello

Output of Read(9) function is

Hello

Th

Output of Readline(9) function is

Hello

Output of Readlines function is

['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']