Python basic to advance English version

# Regular Expressions

### Page Link:

<https://www.thepythoncode.com/article/work-with-regular-expressions-in-python>

<https://docs.python.org/3/library/re.html>

### How to Use Regular Expressions in Python

Learn how to use Python's built-in re module to use several string matching techniques using functions like match, search, finditer and sub.

A [regular expression](https://en.wikipedia.org/wiki/Regular_expression) is a special sequence of characters that forms a search pattern, it can be used to check if a string contains a specified pattern, and it can also be used to extract all occurrences of that pattern and much more.

Regex are everywhere, from validating email addresses, passwords, date formats, to being used in search engines, so it is an essential skill for any developer, and most of programming languages provide regex capabilities. In this tutorial, we will be using [re](https://docs.python.org/3/library/re.html) module in Python.

Here are the techniques we gonna cover:

* [Matching Strings](https://www.thepythoncode.com/article/work-with-regular-expressions-in-python#Matching_Strings)
* [*Search* Method](https://www.thepythoncode.com/article/work-with-regular-expressions-in-python#Search_Method)
* [Finding Multiple Matches](https://www.thepythoncode.com/article/work-with-regular-expressions-in-python#Finding_Multiple_Matches)
* [Replacing Matches](https://www.thepythoncode.com/article/work-with-regular-expressions-in-python#Replacing_Matches)

### Matching Strings

For demonstration on how to use [re.match()](https://docs.python.org/3/library/re.html#re.match) function, say you want to validate user passwords. For instance, you want to make sure the password they enter is at least 8 characters length and contain at least a single digit. The following code does that:

**import** re

lol **=** r"^**(?=**.**\***[0-9]**)**.**{8,}**$"

passwords **=** ['mahmud11']

**for** x **in** passwords:

    m **=** re.match(lol, x)

    print(bool(m))

**import** re

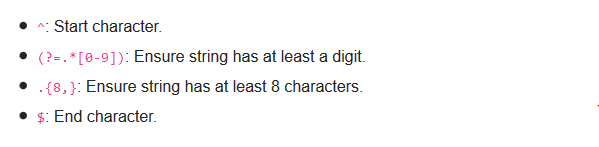
sos**=**r'^mah**(?=**.**\***[0-9]**)**.**{8,}**.s$'

cool**=**"mah123456789s"

x**=**re.match(sos,cool)

print(bool(x))

match\_regex is the regular expression responsible for validating the password criteria we mentioned earlier:



Note we wrapped re.match() method with the built-in [bool()](https://docs.python.org/3/library/functions.html#bool) method to return a boolean that indicates whether the string matches the pattern

### Search Method

A good example to demonstrate [re.search()](https://docs.python.org/3/library/re.html#re.search) method is to search for a specific pattern in a string. For this section, we'll try to extract an IPv4 address from a part of the output of [ipconfig](https://en.wikipedia.org/wiki/Ipconfig) command in Windows:

**import** re

cool**=**'292.168.1.1'

*# regex for IPv4 address*

lol **=** r"((25[0-5]**|**(2[0-4]**|**1[0-9]**|**[1-9]**|**)[0-9])(\.**(?!**$**)|**$))**{4}**"

*# use re.search() method to get the match object*

sos **=** re.search(lol, cool)

print(bool(sos))

**import** re

*# part of ipconfig output*

example\_text **=** """

Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . :

   Link-local IPv6 Address . . . . . : fe80::380e:9710:5172:caee%2

   IPv4 Address. . . . . . . . . . . : 192.168.1.100

   Subnet Mask . . . . . . . . . . . : 255.255.255.0

   Default Gateway . . . . . . . . . : 192.168.1.1

"""

*# regex for IPv4 address*

ip\_address\_regex **=** r"((25[0-5]**|**(2[0-4]**|**1[0-9]**|**[1-9]**|**)[0-9])(\.**(?!**$**)|**$))**{4}**"

*# use re.search() method to get the match object*

match **=** re.search(ip\_address\_regex, example\_text)

print(match)

Don't worry much about ip\_address\_regex expression, it basically validates an IPv4 address (making sure that each number of the total 4 doesn't exceed 255).

We used re.search() in this case to search for a valid IPv4 address, here is the output:

<\_sre.SRE\_Match object; span=(281, 292), match='192.168.1.1'>

re.search() returns a match object which has the start and end indices of the string found and the actual string, in this case, it returned '192.168.1.1' as the matched string. You can use:

* match.start() to get the index of the first character of the found pattern.
* match.end() to get the index of the last character fo the found pattern.
* match.span() to get both start and end as a tuple (start, end).
* match.group() to get the actual string found.

### Finding Multiple Matches(not under stand )

We'll be using the output of same command (ipconfig) but we will try to use regular expressions to match for MAC addresses this time:

**import** re

*# fake ipconfig output*

example\_text **=** """

Ethernet adapter Ethernet:

   Media State . . . . . . . . . . . : Media disconnected

   Physical Address. . . . . . . . . : 88-90-E6-28-35-FA

Ethernet adapter Ethernet 2:

   Physical Address. . . . . . . . . : 04-00-4C-4F-4F-60

   Autoconfiguration IPv4 Address. . : 169.254.204.56(Preferred)

Wireless LAN adapter Local Area Connection\* 2:

   Media State . . . . . . . . . . . : Media disconnected

   Physical Address. . . . . . . . . : B8-21-5E-D3-66-98

Wireless LAN adapter Wi-Fi:

   Physical Address. . . . . . . . . : A0-00-79-AA-62-74

   IPv4 Address. . . . . . . . . . . : 192.168.1.101(Preferred)

   Default Gateway . . . . . . . . . : 192.168.1.1

"""

*# regex for MAC address*

mac\_address\_regex **=** r"([0-9A-Fa-f]**{2}**[:-])**{5}**([0-9A-Fa-f]**{2}**)"

*# iterate over matches and extract MAC addresses*

extracted\_mac\_addresses **=** [ m.group(0) **for** m **in** re.finditer(mac\_address\_regex, example\_text) ]

print(extracted\_mac\_addresses)

After defining the regular expression, we used [re.finditer()](https://docs.python.org/3/library/re.html#re.finditer) function to find all occurrences of MAC addresses in the string passed.

Since finditer() returns an iterator of match objects, we used a list comprehension to extract only the found MAC addresses using group(0) (the entire match). Check out the output:

['88-90-E6-28-35-FA', '04-00-4C-4F-4F-60', 'B8-21-5E-D3-66-98', 'A0-00-79-AA-62-74']

Awesome, we have successfully extracted all MAC addresses in that string. In the next section, we'll see how to use regex to replace occurrences of the pattern in strings.

### Replacing Matches

**import** re

*# a basic regular expression for email matching*

email\_regex **=** r"[a-zA-Z0-9\_.+-]**+**@[a-zA-Z0-9-]**+**\.[a-zA-Z0-9-.]**+**"

*# example text to test with*

example\_text **=** """

Subject: This is a text email!

From: John Doe <john@doe.com>

Some text here!

===============================

Subject: This is another email!

From: Abdou Rockikz <example@domain.com>

Some other text!

"""

*# substitute any email found with [email protected]*

print(re.sub(email\_regex, "[email protected]", example\_text))

We used [re.sub()](https://docs.python.org/3/library/re.html#re.sub) method which takes 3 arguments, the first is the regular expression (the pattern), the second is the replacement of all patterns found, the third is the target string, here is the output:

Subject: This is a text email!

From: John Doe <[email protected]>

Some text here!

===============================

Subject: This is another email!

From: Abdou Rockikz <[email protected]>

Some other text!

Great, as we expected, re.sub() function return the string obtained by replacing the leftmost non-overlapping occurences of the pattern in string by the replacement specified (2nd argument).

Example:

**import** re

lol**=**r'[a-zA-Z0-9\_.+-]**+**@[a-zA-Z0-9]**+**\.[a-zA-Z0-9-.]**+**'

sos**=**'mahmud123@gmail123.com'

x**=**(re.sub(lol,'hello',sos))

print((x))

### Conclusion

Now you have the skills to use regular expressions in Python, note that we didn't cover all the methods provided by [re](https://docs.python.org/3/library/re.html) module, there are other handy functions like [split()](https://docs.python.org/3/library/re.html#re.split) and [fullmatch()](https://docs.python.org/3/library/re.html#re.fullmatch), so I highly encourage you to check the [Python's official documentation](https://docs.python.org/3/library/re.html).

If you aren't sure how to build and construct regular expressions for your needs, you can either check [the official documentation](https://docs.python.org/3/library/re.html) or [this tutorial](https://www.w3schools.com/python/python_regex.asp).

### Python RegEx by w3 school

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

RegEx can be used to check if a string contains the specified search pattern.

### RegEx Module

Python has a built-in package called re, which can be used to work with Regular Expressions.

Import the re module:

### RegEx in Python

When you have imported the re module, you can start using regular expressions:

### Example

Search the string to see if it starts with "The" and ends with "Spain"

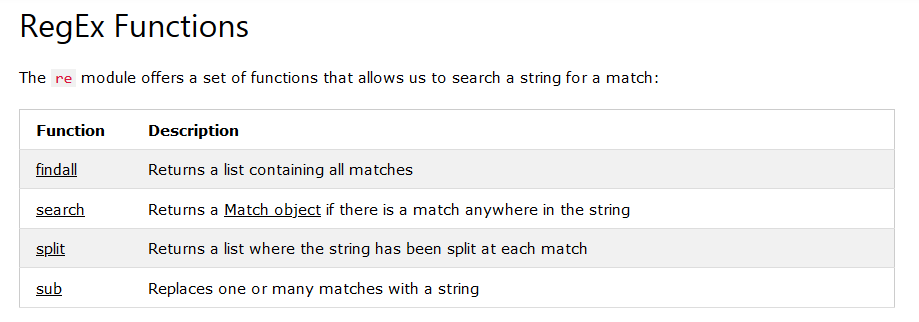
**import** re

txt **=** "The rain in Spain"

x **=** re.search("^The.\*Spain$", txt)

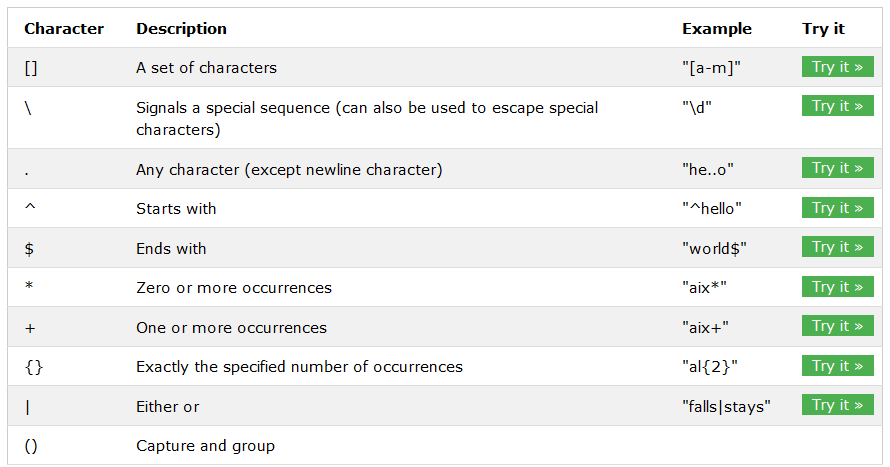
print(bool(x))

### some formula



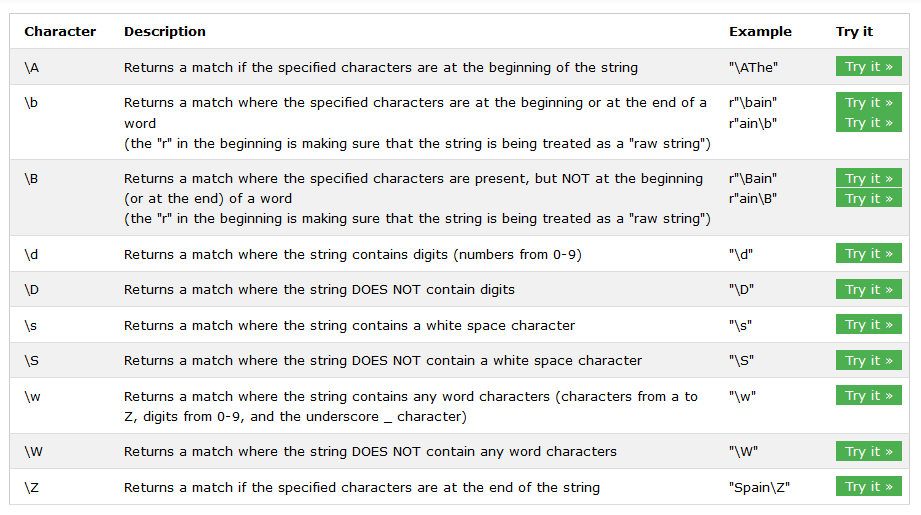
### Metacharacters

Metacharacters are characters with a special meaning:



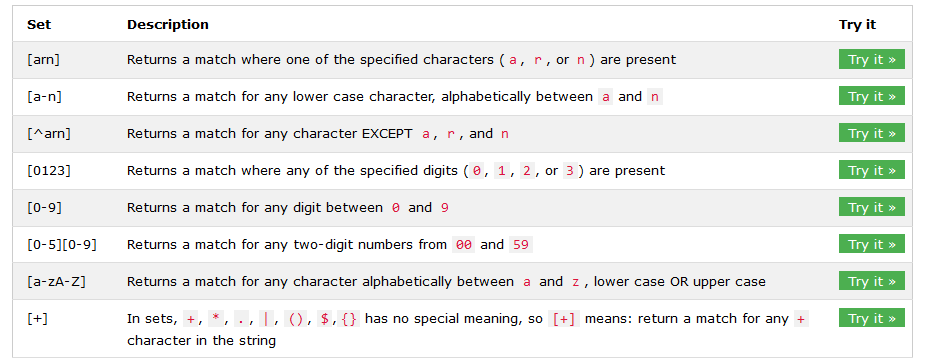
### Special Sequences

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:



### Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning:



### The findall() Function

The findall() function returns a list containing all matches.

**import** re

txt **=** "The rain in Spain"

x **=** re.findall("ai", txt)

print(x)

The list contains the matches in the order they are found.

If no matches are found, an empty list is returned:

Example

Return an empty list if no match was found:

import re  
  
txt = "The rain in Spain"  
x = re.findall("Portugal", txt)  
print(x)

### The search() Function

The search() function searches the string for a match, and returns a [Match object](https://www.w3schools.com/python/python_regex.asp#matchobject) if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

Example

Search for the first white-space character in the string:

**import** re

txt **=** "The rain in Spain"

x **=** re.search("\s", txt)

print("The first white-space character is located in position:", x.start())

If no matches are found, the value None is returned:

Example

Make a search that returns no match:

import re  
  
txt = "The rain in Spain"  
x = re.search("Portugal", txt)  
print(x)

### The split() Function

The split() function returns a list where the string has been split at each match:

Example

Split at each white-space character:

**import** re

txt **=** "The rain in Spain"

x **=** re.split("\s", txt)

print(x)

You can control the number of occurrences by specifying the maxsplit parameter:

Example

Split the string only at the first occurrence:

import re  
  
txt = "The rain in Spain"  
x = re.split("\s", txt, 1)  
print(x)

### The sub() Function

The sub() function replaces the matches with the text of your choice:

Example

Replace every white-space character with the number 9:

**import** re

txt **=** "The rain in Spain"

x **=** re.sub("\s", "9", txt)

print(x)

You can control the number of replacements by specifying the count parameter:

Example

Replace the first 2 occurrences:

import re  
  
txt = "The rain in Spain"  
x = re.sub("\s", "9", txt, 2)  
print(x)

### Match Object

A Match Object is an object containing information about the search and the result.

**Note:** If there is no match, the value None will be returned, instead of the Match Object.

**import** re

txt **=** "The rain in Spain"

x **=** re.search("ai", txt)

print(x) *#this will print an object*

The Match object has properties and methods used to retrieve information about the search, and the result:

.span() returns a tuple containing the start-, and end positions of the match.  
.string returns the string passed into the function  
.group() returns the part of the string where there was a match

Example

Print the position (start- and end-position) of the first match occurrence.

The regular expression looks for any words that starts with an upper case "S":

import re  
  
txt = "The rain in Spain"  
x = re.search(r"\bS\w+", txt)  
print(**x.span()**)

Example

Print the string passed into the function:

import re  
  
txt = "The rain in Spain"  
x = re.search(r"\bS\w+", txt)  
print(**x.string**)

Example

Print the part of the string where there was a match.

The regular expression looks for any words that starts with an upper case "S":

import re  
  
txt = "The rain in Spain"  
x = re.search(r"\bS\w+", txt)  
print(**x.group()**)

# Python JSON

## From w3school

<https://www.w3schools.com/python/python_json.asp>

JSON is a syntax for storing and exchanging data.JSON is text, written with JavaScript object notation.

### Import json

Python has a built-in package called json, which can be used to work with JSON data.

Example

Import the json module:

import json

### Parse JSON - Convert from JSON to Python

If you have a JSON string, you can parse it by using the json.loads() method.

Convert from JSON to Python:

**import** json

*# some JSON:*

x **=**  '{ "name":"John", "age":30, "city":"New York"}'

*# parse x:*

y **=** json.loads(x)

*# the result is a Python dictionary:*

print(y["age"])

### Convert from Python to JSON

If you have a Python object, you can convert it into a JSON string by using the json.dumps() method.

Example

Convert from Python to JSON:

**import** json

*# a Python object (dict):*

x **=** {

  "name": "John",

  "age": 30,

  "city": "New York"

}

*# convert into JSON:*

y **=** json.dumps(x)

*# the result is a JSON string:*

print(y)

### You can convert Python objects of the following types, into JSON strings:

* dict
* list
* tuple
* string
* int
* float
* True
* False
* None

Example

Convert Python objects into JSON strings, and print the values:

**import** json

print(json.dumps({"name": "John", "age": 30}))

print(json.dumps(["apple", "bananas"]))

print(json.dumps(("apple", "bananas")))

print(json.dumps("hello"))

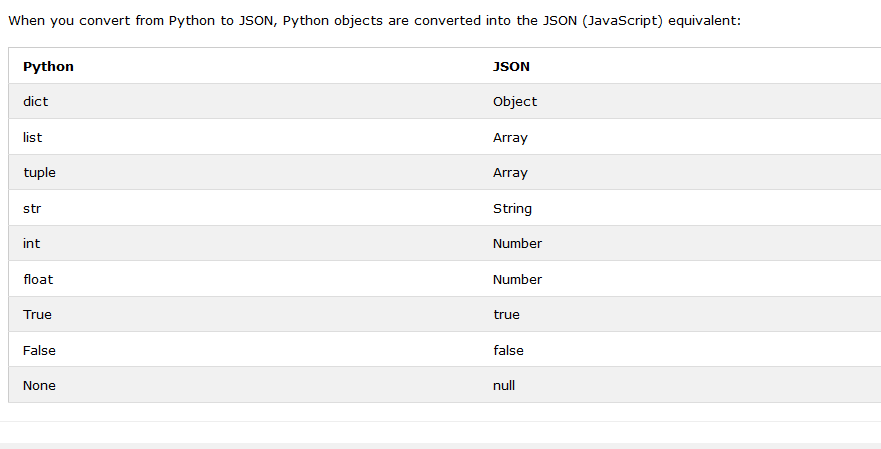
print(json.dumps(42))

print(json.dumps(31.76))

print(json.dumps(True))

print(json.dumps(False))

print(json.dumps(None))



Example

Convert a Python object containing all the legal data types:

**import** json

x **=** {

  "name": "John",

  "age": 30,

  "married": True,

  "divorced": False,

  "children": ("Ann","Billy"),

  "pets": None,

  "cars": [

    {"model": "BMW 230", "mpg": 27.5},

    {"model": "Ford Edge", "mpg": 24.1}

  ]

}

print(json.dumps(x))

### Format the Result

The example above prints a JSON string, but it is not very easy to read, with no indentations and line breaks.

The json.dumps() method has parameters to make it easier to read the result:

You can also define the separators, default value is (", ", ": "), which means using a comma and a space to separate each object, and a colon and a space to separate keys from values:

Example

Use the separators parameter to change the default separator:

json.dumps(x, indent=4, separators=(". ", " = "))

**import** json

x **=** {

  "name": "John",

  "age": 30,

  "married": True,

  "divorced": False,

  "children": ("Ann","Billy"),

  "pets": None,

  "cars": [

    {"model": "BMW 230", "mpg": 27.5},

    {"model": "Ford Edge", "mpg": 24.1}

  ]

}

json.dumps(x, **indent=**4, **separators=**(". ", " = "))

print(json.dumps(x))

### Order the Result

The json.dumps() method has parameters to order the keys in the result:

Example

Use the sort\_keys parameter to specify if the result should be sorted or not:

json.dumps(x, indent=4, sort\_keys=True)

## How to Work with JSON Files in Python

Learn how to save (serialize) and load (deserialize) JSON files in Python using the built-in json module.

<https://www.thepythoncode.com/article/working-with-json-files-in-python>

<https://docs.python.org/3/library/json.html>

[JSON](https://www.json.org/) (JavaScript Object Notation) is a lightweight open standard data-interchange file format, that uses human readable text for transmitting data.

Although you may conclude from the name that it's a Javascript data format. Well, not exactly, JSON is a text format that is completely language independent and uses conventions that are familiar of most popular programming languages such as Python.

In this tutorial, you will use Python for:

* [Saving JSON Data](https://www.thepythoncode.com/article/working-with-json-files-in-python#Saving_JSON_Data)
* [Loading JSON Data](https://www.thepythoncode.com/article/working-with-json-files-in-python#Loading_JSON_Data)

Luckily for us, Python has a built-in module [json](https://docs.python.org/3/library/json.html), that is sufficient for our work, let's get started!

### Saving JSON Data

Python dictionaries are very similar to JSON format, in fact, you can save a dictionary in very few lines of code:

#### Method1:

**import** json

*# example dictionary to save as JSON*

data **=** {

    "first\_name": "John",

    "last\_name": "Doe",

    "email": "john@doe.com",

    "salary": 1499.9, *# just to demonstrate we can use floats as well*

    "age": 17,

    "is\_real": False, *# also booleans!*

    "titles": ["The Unknown", "Anonymous"] *# also lists!*

}

*# save JSON file*

*# 1st option*

**with** open("data1.json", "w") **as** f:

    json.dump(data, f)

Once you execute the above code, you'll notice data1.json file appeared in your working directory. We've opened the file in write mode, and used json.dump() function to serialize the Python dictionary as a JSON formatted stream to the opened file.

The resulting file will look something like this:

That's one way of saving as JSON, you can use json.dumps() function as well:

#### Method2:

**import** json

*# example dictionary to save as JSON*

data **=** {

    "first\_name": "John",

    "last\_name": "Doe",

    "email": "john@doe.com",

    "salary": 1499.9, *# just to demonstrate we can use floats as well*

    "age": 17,

    "is\_real": False, *# also booleans!*

    "titles": ["The Unknown", "Anonymous"] *# also lists!*

}

*# 2nd option*

**with** open("data2.json", "w") **as** f:

    f.write(json.dumps(data, **indent=**4))

json.dumps() function returns the dictionary as a JSON parsed string, you may want this string for other use, we just saved it to a file just to make you aware that it does exist.

Notice I added indent=4 this time as a parameter to json.dumps() function, this will pretty-print JSON array elements and object members, if you use indent=0, it'll only print new lines, and if it's None (default), then it's dumped in a single line (not human readable). The indent keyword exist both in dump() and dumps() functions.

### Handling Non-ASCII Characters

If your data contains non ASCII characters, and you don't want unicode instances on your JSON file (such as \u0623), then you should pass ensure\_ascii=False to json.dump() function:

**import** json

unicode\_data **=** {

    "first\_name": "أحمد",

    "last\_name": "علي"

}

**with** open("lol.json", "w", **encoding=**"utf-8") **as** f:

    json.dump(unicode\_data, f, **ensure\_ascii=**False)

### Loading JSON Data

It is pretty straightforward to deserialize JSON files and load them into Python, the below code loads the previously saved JSON file:

**import** json

*# read a JSON file*

*# 1st option*

file\_name **=** "data1.json"

**with** open(file\_name) **as** f:

    data **=** json.load(f)

print(data)

json.load() function will automatically return a Python dictionary, which ease our work with JSON files, here is the output:

Similarly, you can also use json.loads() function to read a string instead:

**import** json

*# 2nd option*

file\_name **=** "data2.json"

**with** open(file\_name) **as** f:

    data **=** json.loads(f.read())

print(data)

So we've read the file content first using read() method, and then we pass it to json.loads() function to parse it.

### Some example:

**import** json

*# example dictionary to save as JSON*

data **=** {

    "first\_name": "John",

    "last\_name": "Doe",

    "email": "john@doe.com",

    "salary": 1499.9, *# just to demonstrate we can use floats as well*

    "age": 17,

    "is\_real": False, *# also booleans!*

    "titles": ["The Unknown", "Anonymous"] *# also lists!*

}

*# save JSON file*

*# 1st option*

**with** open("data1.json", "w") **as** f:

    json.dump(data, f)

*# 2nd option*

**with** open("data2.json", "w") **as** f:

    f.write(json.dumps(data, **indent=**4))

unicode\_data **=** {

    "first\_name": "أحمد",

    "last\_name": "علي"

}

**with** open("data\_unicode.json", "w", **encoding=**"utf-8") **as** f:

    json.dump(unicode\_data, f, **ensure\_ascii=**False)

example:

**import** requests

**import** json

*# make API request and parse JSON automatically*

data **=** requests.get("https://jsonplaceholder.typicode.com/users").json()

*# save all data in a single JSON file*

file\_name **=** "user\_data.json"

**with** open(file\_name, "w") **as** f:

    json.dump(data, f, **indent=**4)

    print(file\_name, "saved successfully!")

*# or you can save each entry into a file*

**for** user **in** data:

*# iterate over `data` list*

    file\_name **=** f"user\_{user['id']}.json"

**with** open(file\_name, "w") **as** f:

        json.dump(user, f, **indent=**4)

        print(file\_name, "saved successfully!")

*# load 2nd user for instance*

file\_name **=** "user\_2.json"

**with** open(file\_name) **as** f:

    user\_data **=** json.load(f)

print(user\_data)

print("Username:", user\_data["username"])

### Conclusion

Now you know how to use dump(), dumps(), load() and loads() functions in the json module and you have the ability to work with JSON data in Python.

As a developer, you'll likely have to interact with it frequently, you'll encounter JSON a lot of times, especially when [working with REST APIs](https://www.thepythoncode.com/topic/using-apis-in-python), or when you [scrape data from the web](https://www.thepythoncode.com/topic/web-scraping).

# Python File

## From W3 school

<https://www.w3schools.com/python/python_try_except.asp>

## File Open

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

The key function for working with files in Python is the open() function.

The open() function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

### Syntax

To open a file for reading it is enough to specify the name of the file:

f = open("demofile.txt")

The code above is the same as:

f = open("demofile.txt", "rt")

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

### Open a File on the Server

Assume we have the following file, located in the same folder as Python:

To open the file, use the built-in open() function.

The open() function returns a file object, which has a read() method for reading the content of the file:

f **=** open("lol.txt",'r')

print(f.read())

If the file is located in a different location, you will have to specify the file path, like this:

f **=** open("cool/OOP/lol.txt",'r')

print(f.read())

### Read Only Parts of the File

By default the read() method returns the whole text, but you can also specify how many characters you want to return:

#### Example

Return the 5 first characters of the file:

f **=** open("lol.txt",'r')

print(f.read(2))

### Read Lines

You can return one line by using the readline() method:

#### Example

Read one line of the file:

f **=** open("lol.txt",'r')

print(f.readline())

By calling readline() two times, you can read the two first lines:

f **=** open("lol.txt",'r')

print(f.readline())

print(f.readline())

By looping through the lines of the file, you can read the whole file, line by line:

f **=** open("lol.txt", "r")

**for** x **in** f:

  print(x)

### Close Files

f **=** open("lol.txt", "r")

**for** x **in** f:

  print(x)

f.close()

It is a good practice to always close the file when you are done with it.

## Python File Write

### Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

#### Example

##### Open the file "demofile2.txt" and append content to the file:

f **=** open("demofile2.txt", "a")

f.write("Now the file has more content!")

f.close()

*#open and read the file after the appending:*

f **=** open("demofile2.txt", "r")

print(f.read())

#### Example

##### Open the file "demofile3.txt" and overwrite the content:

f **=** open("demofile3.txt", "w")

f.write("Woops! I have deleted the content!")

f.close()

*#open and read the file after the appending:*

f **=** open("demofile3.txt", "r")

print(f.read())

### With open function

*#using with open formula ,do not need to extra close file option write*

**with** open('lol.txt','a') **as** e:

    sos**=**e.write('nowsins')

**with** open('lol.txt','r') **as** r:

    x**=**r.read()

    print(x)

## Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

#### Example

##### Create a file called "myfile.txt":

f **=** open("myfile.txt", "x")

## Python Delete File

To delete a file, you must import the OS module, and run its os.remove() function:

#### Example

##### Remove the file "lol.txt":

**import** os

os.remove("lol.txt")

## Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

#### Example

##### Check if file exists, then delete it:

**import** os

**if** os.path.exists("myfile.txt"):

  os.remove("myfile.txt")

**else**:

  print("The file does not exist")

## Create a folder

Create a folder name myfolder

**import** os

os.mkdir("myfolder")

## Delete Folder

To delete an entire folder, use the os.rmdir() method:

#### Example

##### Remove the folder "myfolder":

**import** os

os.rmdir("myfolder")

# Python File handling with (Try, Except)

The try block lets you test a block of code for errors.

The except block lets you handle the error.

The finally block lets you execute code, regardless of the result of the try- and except blocks.

### Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the try statement:

#### Example

The try block will generate an exception, because x is not defined:

**try**:

  print(x)

**except**:

  print("An exception occurred")

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error:

### Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

#### Example

Print one message if the try block raises a NameError and another for other errors:

**try**:

  print(x)

**except** NameError:

  print("Variable x is not defined")

**except**:

  print("Something else went wrong")

### Else

You can use the else keyword to define a block of code to be executed if no errors were raised:

#### Example

In this example, the try block does not generate any error:

**try**:

  print("Hello")

**except**:

  print("Something went wrong")

**else**:

  print("Nothing went wrong")

### Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

#### Example

**try**:

  print(x)

**except**:

  print("Something went wrong")

**finally**:

  print("The 'try except' is finished")

This can be useful to close objects and clean up resources:

#### Example

Try to open and write to a file that is not writable:

**try**:

  f **=** open("demofile.txt")

  f.write("Lorum Ipsum")

**except**:

  print("Something went wrong when writing to the file")

**finally**:

  f.close()

The program can continue, without leaving the file object open.

### Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the raise keyword.

#### Example

Raise an error and stop the program if x is lower than 0:

x **=** **-**1

**if** x **<** 0:

**raise** Exception("Sorry, no numbers below zero")

The raise keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

#### Example

Raise a TypeError if x is not an integer:

x **=** "hello"

**if** **not** type(x) **is** *int*:

**raise** TypeError("Only integers are allowed")

## How to Handle Files in Python

<https://www.thepythoncode.com/article/file-handling-in-python-using-os-module>

<https://docs.python.org/3/library/functions.html#open>

Learn how to work with files in Python using os and shutil modules including creating, renaming, moving, removing files and directories, listing all current files and directories and more.

In the software industry, most of the programs handle files in one way or another, such as creating files, renaming them, moving, listing, etc. As a programmer, you should definitely have this fundamental skill. As a result, in this tutorial, we will be using [os](https://docs.python.org/3/library/os.html) module in Python to make several operations on files and directories regardless of our operating system we're using.

Now, you need to know that [os](https://docs.python.org/3/library/os.html) module isn't only for handling files and directories, it has a ton of methods and tools for other operations, such as handling environment variables, command line arguments, system process management, and even Linux extended attributes that are specific for [Linux operating system](https://en.wikipedia.org/wiki/Linux).

Alright, let's get started, since [os](https://www.thepythoncode.com/article/os%20—%20Miscellaneous%20operating%20system%20interfaces) module is a built-in module, we don't have to install anything.

* [Printing the Current Working Directory](https://www.thepythoncode.com/article/file-handling-in-python-using-os-module#Printing_Current_Working_Directory)
* [Creating Directories](https://www.thepythoncode.com/article/file-handling-in-python-using-os-module#Creating_Directories)
* [Changing Directories](https://www.thepythoncode.com/article/file-handling-in-python-using-os-module#Changing_Directories)
* [Creating Nested Directories](https://www.thepythoncode.com/article/file-handling-in-python-using-os-module#Creating_Nested_Directories)
* [Creating Files](https://www.thepythoncode.com/article/file-handling-in-python-using-os-module#Creating_Files)
* [Renaming Files](https://www.thepythoncode.com/article/file-handling-in-python-using-os-module#Renaming_Files)
* [Moving Files](https://www.thepythoncode.com/article/file-handling-in-python-using-os-module#Moving_Files)
* [Listing Files and Directories](https://www.thepythoncode.com/article/file-handling-in-python-using-os-module#Listing_Files_And_Directories)
* [Deleting Files](https://www.thepythoncode.com/article/file-handling-in-python-using-os-module#Deleting_Files)
* [Deleting Directories](https://www.thepythoncode.com/article/file-handling-in-python-using-os-module#Deleting_Directories)
* [Retrieving Information About Files](https://www.thepythoncode.com/article/file-handling-in-python-using-os-module#Retrieving_Information_About_Files)

### Printing the Current Working Directory

In order to get the current working directory, you need to use os.getcwd():

**import** os

*# print the current directory*

print("The current directory:", os.getcwd())

os.getcwd() returns a unicode string representing the current working directory, here is my output:

### Creating Directories

To make a folder/directory in any operating system, all you need to do is:

**import** os

*# make an empty directory (folder)*

os.mkdir("folder")

Once you execute this, it'll immediately spawn a directory named "folder" in the current working directory.

If you run this again, it will raise a FileExistsError, that's because the folder already exists, a good way to solve this, is to run os.mkdir() only if the folder doesn't exist, we can easily do this:

**import** os

*# running mkdir again with the same name raises FileExistsError, run this instead:*

**if** **not** os.path.isdir("folder"):

     os.mkdir("folder")

os.path.isdir() function returns True if the pathname we passed refers to an existing directory.

### Changing Directories

It is fairly easy to change directories, let's change to the folder we just created:

**import** os

*# changing the current directory to 'folder'*

os.chdir("folder")

Now let's print the working directory again:

*# printing the current directory now*

print("The current directory changing the directory to folder:", os.getcwd())

### Creating Nested Directories

et's say you want to create not only one folder, but you want to create many nested folders:

**import** os

*# make several nested directories*

os.makedirs("nested1/nested2/nested3")

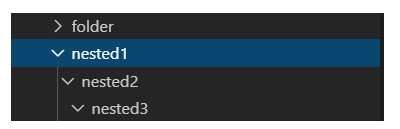
or

**import** os

*# make several nested directories*

os.makedirs("lol1/lol2/lol3")

This will create 3 folders recursively as shown in the following image:



### Creating Files

In order to create files in Python, you don't need any module, you can use the built-in function [open()](https://docs.python.org/3/library/functions.html#open) which takes the filename you want to create as the first parameter and the mode you want to open the file with as a second parameter:

*# create a new text file*

text\_file **=** open("tiktok.txt", "w")

*# write to this file some text*

text\_file.write("This is a text file")

I used "w" as file opening mode which stands for write, you can also use "a" for appending to existing files or "r" for from reading existing files. For more information about file opening modes, check this [page](https://docs.python.org/3/library/functions.html#open).

### Renaming Files

It's pretty easy to rename a file using [os](https://docs.python.org/3/library/os.html) module, let's rename the file we justed created:

**import** os

*# rename text.txt to renamed-text.txt*

os.rename("renamed-toktok.txt", "tiktik.txt")

os.rename() function takes 2 parameters, the name of file or directory you want to rename, and the destination name you want to rename to.

### Moving Files

We can use os.replace() function to move files or directories:

**import** os

*# replace (move) this file to another directory*

os.replace("renamed-text.txt", "folder/renamed-text.txt")

Note that this will overwrite the destination, so if there is another pre-existing file in "folder" that got the same name "renamed-text.txt", but different content, it will overwrite it.

# Python Modules

## What is a Module?

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

## What is a Package?

A package contains all the files you need for a module.

Modules are Python code libraries you can include in your project.

## Create a Module

To create a module just save the code you want in a file with the file extension .py:

#### Example

##### Save this code in a file named mymodule.py

**def** greeting(**name**):

  print("Hello, " **+** name)

## Use a Module

Now we can use the module we just created, by using the import statement:

#### Example

##### Import the module named mymodule, and call the greeting function:

**import** mymodule

mymodule.fun(2,2)

OR

**from** mymodule **import** fun

fun(2,2)

## Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

#### Example

##### Save this code in the file mymodule.py

person1 **=** {

  "name": "John",

  "age": 36,

  "country": "Norway"

}

Import module

**from** mymodule **import** person1

print(person1)

OR

**import** mymodule

print(mymodule.person1)

## Naming a Module

You can name the module file whatever you like, but it must have the file extension .py

## Re-naming a Module

You can create an alias when you import a module, by using the as keyword:

#### Example

##### Create an alias for mymodule called mx:

**import** mymodule **as** mx

print(mx.person1)

OR

**from** mymodule **import** person1 **as** mx

print(mx)

## Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

#### Example

##### Import and use the platform module:

**import** platform

x **=** platform.system()

print(x)

## Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module. The dir() function:

#### Example

##### List all the defined names belonging to the platform module:

**import** platform

x **=** dir(platform)

print(x)

**Note:** The dir() function can be used on all modules, also the ones you create yourself.

## Import From Module

You can choose to import only parts from a module, by using the from keyword.

#### Example

##### The module named mymodule has three functions fun1,fun2,fun3, we call only fun1,fun2:

**from** mymodule **import** fun1,fun2

print(fun1(2,2))

print(fun2(3,3))

by using \* we call all module function

## import Datetime

### Dates

A date in Python is not a data type of its own, but we can import a module named datetime to work with dates as date objects.

#### Example

##### Import the datetime module and display the current date:

**import** datetime

print(datetime.datetime.now())

OR

**from** datetime **import** datetime

print(datetime.now())

The date contains year, month, day, hour, minute, second, and microsecond.

The datetime module has many methods to return information about the date object.

Here are a few examples, you will learn more about them later in this chapter:

**import** datetime

x **=** datetime.datetime.now()

print(x.year)

print(x.month)

print(x.day)

print(x.hour)

print(x.minute)

print(x.second)

print(x.now())

print(x.strftime("%A"))

print(x.strftime("%B"))

print(x.strftime("%C"))

print(x.strftime("%D"))

print(x.strftime("%F"))

## Creating Date Objects

To create a date, we can use the datetime() class (constructor) of the datetime module.

The datetime() class requires three parameters to create a date: year, month, day.

#### Example

Create a date object:

**import** datetime

x **=** datetime.datetime(2020, 5, 17)

print(x)

The datetime() class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of 0, (None for timezone).

## The strftime() Method

The datetime object has a method for formatting date objects into readable strings.

The method is called strftime(), and takes one parameter, format, to specify the format of the returned string:

#### Example

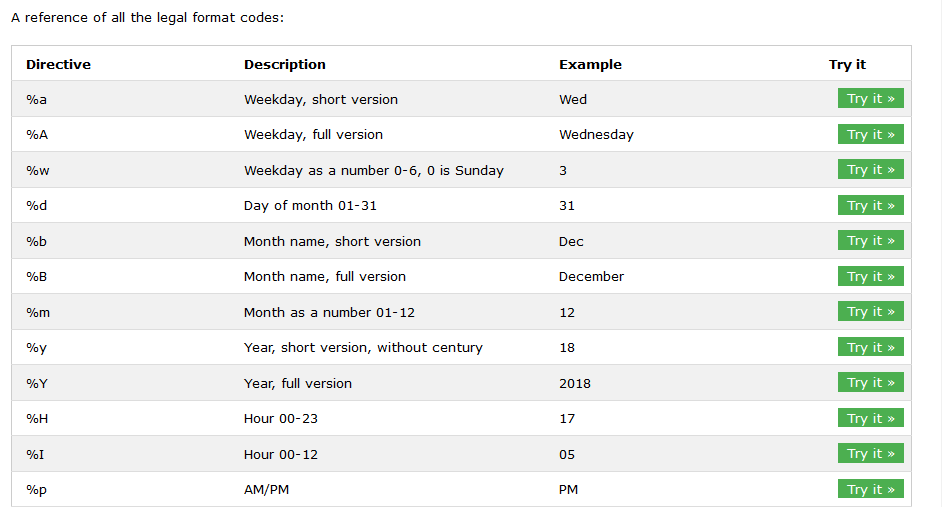
Display the name of the month:

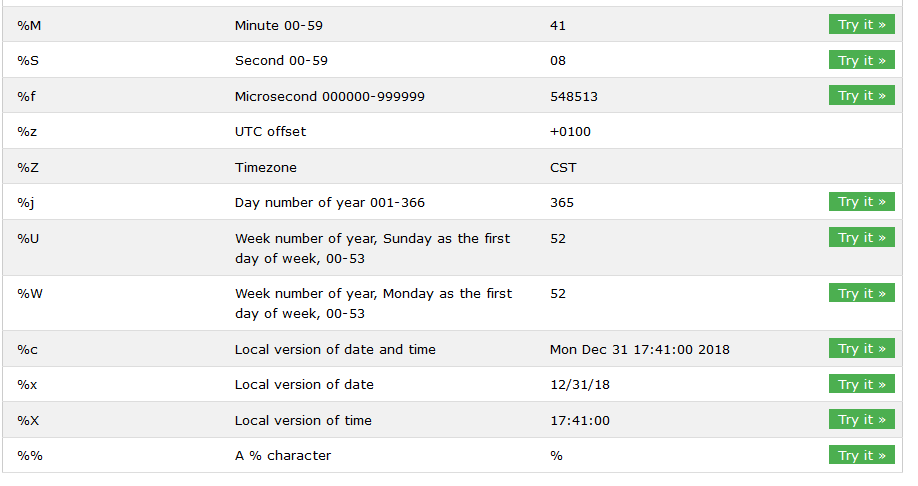
**import** datetime

x **=** datetime.datetime(2018, 6, 1)

print(x.strftime("%B"))

## code box





## Import math

Python has a set of built-in math functions, including an extensive math module, that allows you to perform mathematical tasks on numbers.

### Built-in Math module

Python has also a built-in module called math, which extends the list of mathematical functions.

To use it, you must import the math module:

Import math

The min() and max() functions can be used to find the lowest or highest value in an iterable:

#### Example

**import** math

x**=**min(2,3,4)

y**=**max(2,3,4)

print(x)

print(y)

#### Example

##### The abs() function returns the absolute (positive) value of the specified number:

**import** math

x **=** abs(**-**7.25)

print(x)

#### Example

##### Return the value of 4 to the power of 3 (same as 4 \* 4 \* 4):

**import** math

x **=** pow(2,2)

print(x)

When you have imported the math module, you can start using methods and constants of the module.

#### Example

##### The math.sqrt() method for example, returns the square root of a number:

**from** math **import** sqrt

x **=** sqrt(16)

print(x)

#### Example

The math.ceil() method rounds a number upwards to its nearest integer, and the math.floor() method rounds a number downwards to its nearest integer, and returns the result:

**import** math

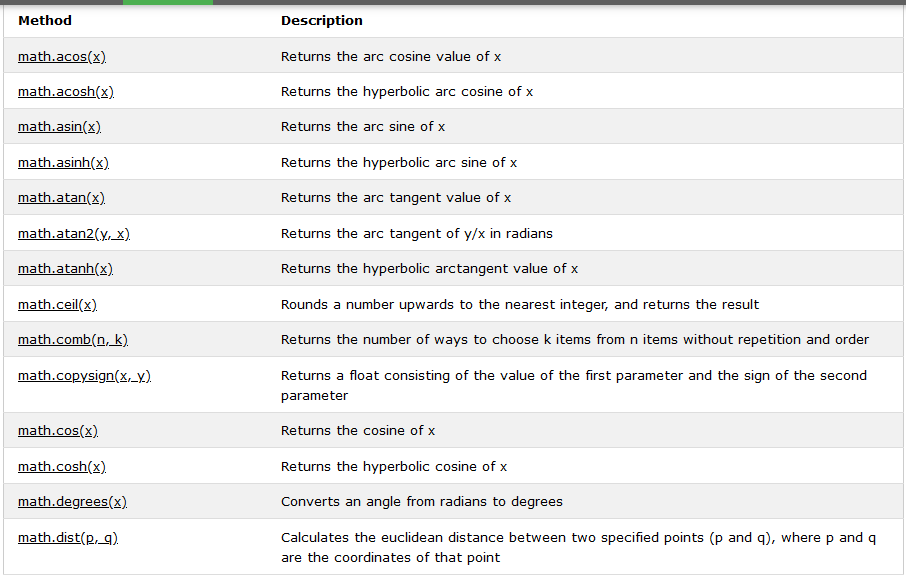
x **=** math.ceil(1.4)

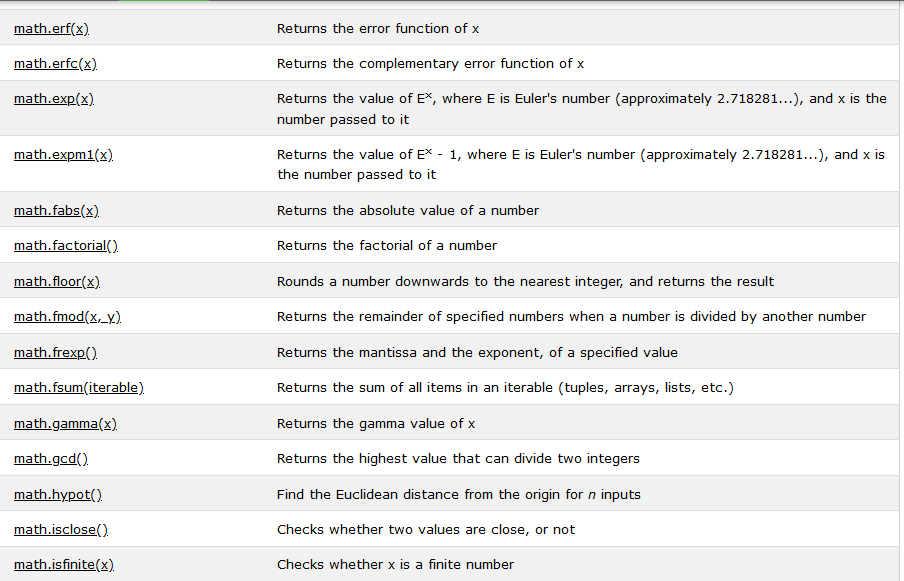
y **=** math.floor(1.4)

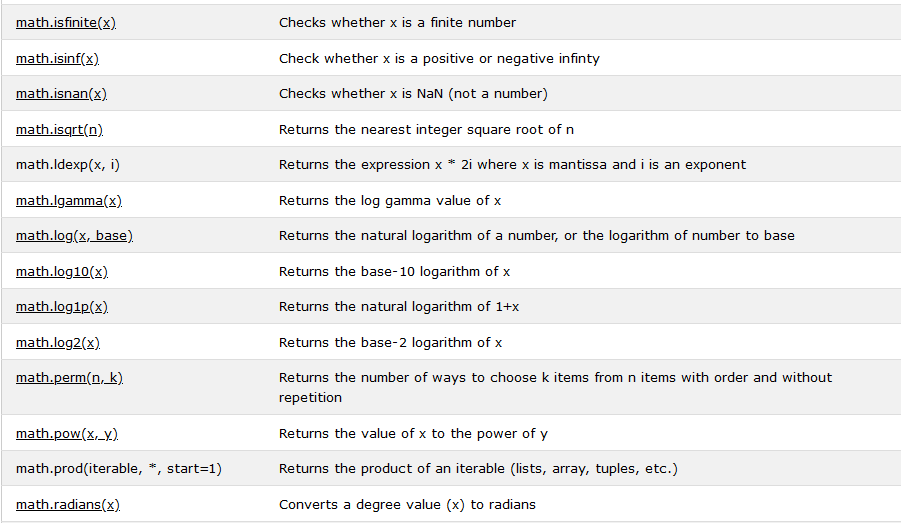
print(x) *# returns 2*

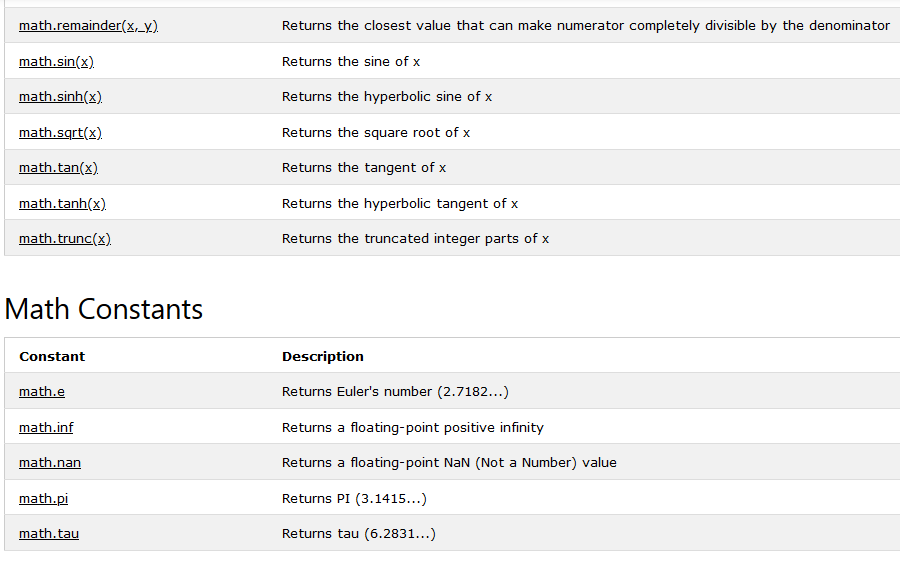
print(y) *# returns 1 )*

### all math module chart









# Virtual env or Python PIP

## What is PIP?

PIP is a package manager for Python packages, or modules if you like.

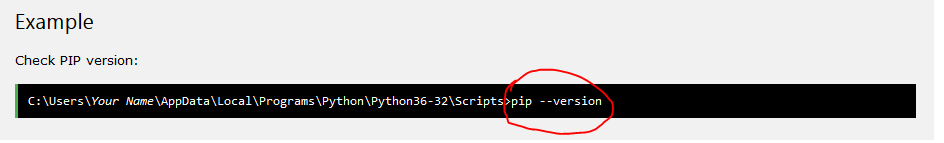
## What is a Package?

A package contains all the files you need for a module.

Modules are Python code libraries you can include in your project.

## Check if PIP is Installed

Navigate your command line to the location of Python's script directory, and type the following:



## Download a Package

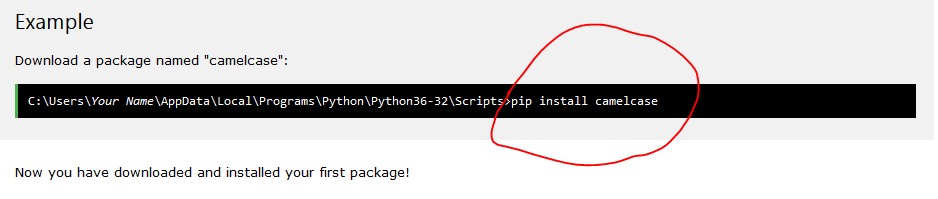
Downloading a package is very easy.

Open the command line interface and tell PIP to download the package you want.

Navigate your command line to the location of Python's script directory, and type the following:

#### Example

Download a package named "camelcase":



## Using a Package

Once the package is installed, it is ready to use.

Import the "camelcase" package into your project.

#### Example

##### Import and use "camelcase":

**import** camelcase

c **=** camelcase.CamelCase()

txt **=** "hello world"

print(c.hump(txt))

## Find Packages

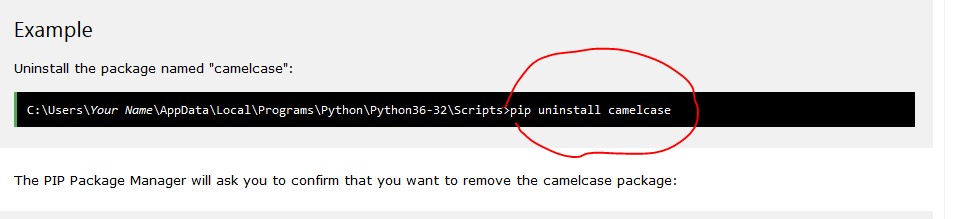
Find more packages at <https://pypi.org/>.

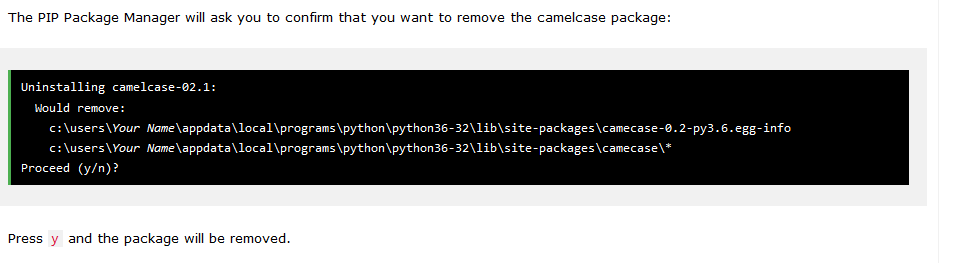
## Remove a Package

Use the uninstall command to remove a package:

#### Example

##### Uninstall the package named "camelcase":



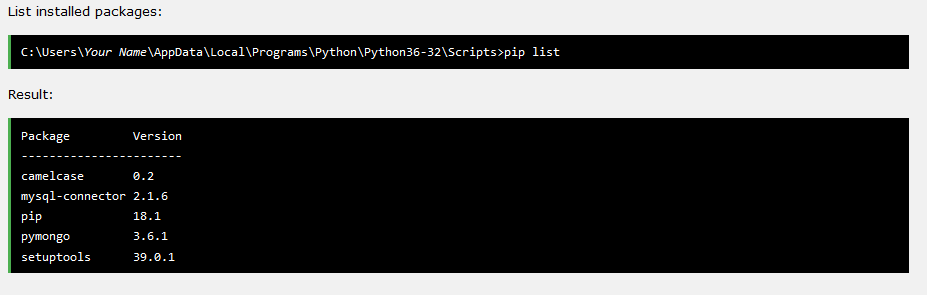


## List Packages

Use the list command to list all the packages installed on your system:

### Example

List installed packages:



# Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

## Creating a Function

In Python a function is defined using the def keyword:

**def** fun():

    print('heloo')

fun()

## Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

**def** my\_function(**fname**): *#here fname is an argument*

  print(fname **+** " Refsnes")

my\_function("Emil") *#here email,tobias etc are value or parameter*

my\_function("Tobias") *#*

my\_function("Linus")

## Parameters or Arguments?

The terms parameter and argument can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

## Arbitrary Arguments, \*args

If you do not know how many arguments that will be passed into your function, add a \* before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

#### Example

If the number of arguments is unknown, add a \* before the parameter name:

**def** fun(**\*args**):

    print(args)

fun('mahmud','hossain',12345)

## Keyword Arguments

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

#### Example

**def** fun(**x**,**y**,**z**):

    print('x is:',x ,'y is:',y ,'and z is:',z)

fun('mahmud','a',12345)

## Arbitrary Keyword Arguments, \*\*kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: \*\* before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly:

#### Example

If the number of keyword arguments is unknown, add a double \*\* before the parameter name:

**def** fun(**\*\*kwargs**):

    print(kwargs)

fun(**a=**2,**b=**3,**c=**'mahmud')

## Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

#### Example

**def** fun(**x**,**y**,**z=**123):

    print(x, 'and',y, 'and',z)

fun('mahmud','9999')

## Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

#### Example

**def** fun(**country**):

**for** x **in** country:

        print(x)

country**=**['bangladesh','india','pakistan']

fun(country)

## Return Values

To let a function return a value, use the return statement:

#### Example

**def** fun(**x**,**y**):

**return** x**+**y

print(fun(2,3))

## The pass Statement

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

#### Example

**def** fun(**x**,**y**):

**pass**

fun(2,2)

## Recursion(read again)

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, tri\_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

#### Example

Recursion Example

**def** tri\_recursion(**k**):

**if**(k **>** 0):

    result **=** k **+** tri\_recursion(k **-** 1)

    print(result)

**else**:

    result **=** 0

**return** result

print("\n\nRecursion Example Results")

tri\_recursion(6)

## Iterators(read again)

### Python Iterators

An iterator is an object that contains a countable number of values.

An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods \_\_iter\_\_() and \_\_next\_\_().

### Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.

All these objects have a iter() method which is used to get an iterator:

#### Example

Return an iterator from a tuple, and print each value:

mytuple **=** ("apple", "banana", "cherry")

myit **=** iter(mytuple)

print(next(myit))

print(next(myit))

print(next(myit))

Even strings are iterable objects, and can return an iterator:

#### Example

Strings are also iterable objects, containing a sequence of characters:

mystr **=** "banana"

myit **=** iter(mystr)

print(next(myit))

print(next(myit))

print(next(myit))

print(next(myit))

print(next(myit))

print(next(myit))

## Looping Through an Iterator

We can also use a for loop to iterate through an iterable object:

#### Example

Iterate the values of a tuple:

mytuple **=** ("apple", "banana", "cherry")

**for** x **in** mytuple:

  print(x)

#### Example

Iterate the characters of a string:

mystr = "banana"  
  
for x in mystr:  
  print(x)

The for loop actually creates an iterator object and executes the next() method for each loop.

## Create an Iterator

To create an object/class as an iterator you have to implement the methods \_\_iter\_\_() and \_\_next\_\_() to your object.

As you have learned in the [Python Classes/Objects](https://www.w3schools.com/python/python_classes.asp) chapter, all classes have a function called \_\_init\_\_(), which allows you to do some initializing when the object is being created.

The \_\_iter\_\_() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The \_\_next\_\_() method also allows you to do operations, and must return the next item in the sequence.

#### Example

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

class MyNumbers:

**def** \_\_iter\_\_(*self*):

*self*.a **=** 1

**return** *self*

**def** \_\_next\_\_(*self*):

    x **=** *self*.a

*self*.a **+=** 1

**return** x

myclass **=** MyNumbers()

myiter **=** iter(myclass)

print(next(myiter))

print(next(myiter))

print(next(myiter))

print(next(myiter))

print(next(myiter))

## StopIteration

The example above would continue forever if you had enough next() statements, or if it was used in a for loop.

To prevent the iteration to go on forever, we can use the StopIteration statement.

In the \_\_next\_\_() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

#### Example

Stop after 20 iterations:

class MyNumbers:

**def** \_\_iter\_\_(*self*):

*self*.a **=** 1

**return** *self*

**def** \_\_next\_\_(*self*):

**if** *self*.a **<=** 20:

      x **=** *self*.a

*self*.a **+=** 1

**return** x

**else**:

**raise** StopIteration

myclass **=** MyNumbers()

myiter **=** iter(myclass)

**for** x **in** myiter:

  print(x)

## Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

#### Syntax

lambda arguments : expression

The expression is executed and the result is returned:

#### Example

A lambda function that adds 10 to the number passed in as an argument, and print the result:

lol**=**((lambda **x**,**y**:x**+**y)(2,2))

print(lol)

Lambda functions can take any number of arguments:

#### Example

A lambda function that multiplies argument a with argument b and print the result:

x = lambda a, b : a \* b  
print(x(5, 6))

## Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

def myfunc(n):  
  return lambda a : a \* n  
  
mydoubler = myfunc(2)  
  
print(mydoubler(11))

#### example, lambda function into main function

**def** fun(**n**):

**return** ((lambda **x**,**y**:x**+**y**+**n)(2,2))

cool**=**fun(5)

print(cool)

Or, use the same function definition to make both functions, in the same program:

### Example

def myfunc(n):  
  return lambda a : a \* n  
  
mydoubler = myfunc(2)  
mytripler = myfunc(3)  
  
print(mydoubler(11))   
print(mytripler(11))

## Python Scope or global and local variable

A variable is only available from inside the region it is created. This is called **scope**.

### 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** fun(**x**):

    name**=**'mahmud'

**return** x**+**name

lol**=**fun('mr')

print(lol)

#### 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** fun(**x**):

    print(x)

**def** fun1(**y**):

        print(y)

**def** fun2(**z**):

            print(z)

        fun2('nowsin')

    fun1('lamyaa')

fun('mahmud')

### 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:

name**=**'mahmud'

**def** fun(**x**):

    print(x,name)

fun('mr')

### 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)

## python function from greek for greek

<https://www.geeksforgeeks.org/global-local-variables-python/?ref=lbp>

### Global and Local Variables in Python

Global variables are the one that are defined and declared outside a function and we need to use them inside a function.

**def** fun():

    print(x)

x**=**'hello'

fun()

The variable s is defined as the string “I love Geeksforgeeks” before we call the function f(). The only statement in f() is the “print s” statement. As there is no local s, the value from the global s will be used.

If two type of variable are given ,local and global, on the output global variable is show

x**=**'mahmud'

**def** fun():

    x**=**'hossain'

**return** x

print(x)

fun()

print(x)

### Global keyword in Python

*#two global variable, with sum function*

x**=**10

y**=**5

**def** fun():

    sum**=**x**+**y

    print(sum)

fun()

*#one global and one local bvariable ,with sum function*

a**=**10

**def** fun1():

    a**=**a**+**5

    print(sum)

fun1()

output error

*#type global keyword*

s**=**10

**def** fun2():

*global* s

    s**=**s**+**5

    print(s)

fun2()

Global variables across python modules :  
The best way to share global variables across different modules within the same program is to create a special module (often named config or cfg). Import the config module in all modules of your application; the module then becomes available as a global name. There is only one instance of each module and so any changes made to the module object get reflected everywhere. For Example, sharing global variables across modules  
**Code 1:** Create a config.py file to store global variables:

|  |
| --- |
| x = 0  y = 0  z ="none" |

**Code 2:** Create a modify.py file to modify global variables:

|  |
| --- |
| import config  config.x = 1  config.y = 2  config.z ="geeksforgeeks" |

Here we have modified the value of x, y, and z. These variables were defined in the module config.py, hence we have to import config module and we can use config.variable\_name to access these variables.

**Code 3:** Create a main.py file to modify global variables:

|  |
| --- |
| import config  import modify  print(config.x)  print(config.y)  print(config.z) |

**Run given two program**

**import** config

*#import modify*

print(config.x)

print(config.y)

print(config.z)

*#and*

**import** config

**import** modify

print(config.x)

print(config.y)

print(config.z)

### Global in Nested functions

In order to use global inside a nested functions, we have to declare a variable with global keyword inside a nested function

|  |
| --- |
| # Python program showing a use of  # global in nested function |

**def** add():

     x **=** 15

**def** change():

*global* x

         x **=** 20

     print("Before making changing: ", x)

     print("Making change")

     change()

     print("After making change: ", x)

add()

print("value of x",x)

In the above example Before and after making change(), the variable x takes the value of local variable i.e x = 15. Outside of the add() function, the variable x will take value defined in the change() function i.e x = 20. because we have used global keyword in x to create global variable inside the change() function (local scope).

# Classes and Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

## Create a Class

To create a class, use the keyword class:

#### Example

##### Create a class named MyClass, with a property named x:

class one:

    print('hello')

## Create Object

Now we can use the class named MyClass to create objects:

#### Example

Create an object named p1, and print the value of x:

lol=one()

### finally class and object

class one: #class define

    print('hello')

lol**=**one() #object create

## The \_\_init\_\_() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in \_\_init\_\_() function.

All classes have a function called \_\_init\_\_(), which is always executed when the class is being initiated.

Use the \_\_init\_\_() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

#### Example

Create a class named Person, use the \_\_init\_\_() function to assign values for name and age:

class one:

**def** \_\_init\_\_(*self*):

        print('i run must')

**def** fun1(*self*):

        print('i am funtion')

lol**=**one()

lol.fun1()

#### example:

class one:

**def** \_\_init\_\_(*self*,**x**,**y**):

*self*.x**=**x

*self*.y**=**y

        print(x**+**y)

lol**=**one(2,2)

**Note:** The \_\_init\_\_() function is called automatically every time the class is being used to create a new object.

## Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

#### Example

Insert a function that prints a greeting, and execute it on the p1 object:

class one:

**def** \_\_init\_\_(*self*,**name**):

*self*.name**=**name

**def** fun1(*self*):

        print('i am ',*self*.name)

lol**=**one('mahmud')

lol.fun1()

**Note:** The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

class one:

**def** \_\_init\_\_(*self*,**name**): *#init method*

*self*.name**=**name

**def** fun1(*self*):

        print('i am ',*self*.name) *#instance method*

lol**=**one('mahmud') *#object / instansiation*

lol.fun1()

## The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

class Person:

**def** \_\_init\_\_(**cool**, **name**, **age**):

   cool.name **=** name

   cool.age **=** age

**def** myfunc(**sos**):

    print("Hello my name is " **+** sos.name)

p1 **=** Person("John", 36)

p1.myfunc()

## Modify Object Properties

You can modify properties on objects like this:

#### Example

Set the name=’mahmud’

class Person:

**def** \_\_init\_\_(*self*, **name**, **age**):

*self*.name **=** name

*self*.age **=** age

**def** myfunc(*self*):

    print("Hello my name is " **+** *self*.name)

p1 **=** Person("John", 36)

p1.name**=**'mahmud'

p1.myfunc()

## Delete Object Properties

You can delete properties on objects by using the del keyword:

del p1.age

#### Example

Delete the age property from the p1 object:

class Person:

**def** \_\_init\_\_(*self*, **name**, **age**):

*self*.name **=** name

*self*.age **=** age

**def** myfunc(*self*):

    print("Hello my name is " **+** *self*.name)

p1 **=** Person("John", 36)

*#del p1.name #if delete name,show errror*

**del** p1.age

p1.myfunc()

## Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

**Parent class** is the class being inherited from, also called base class.

**Child class** is the class that inherits from another class, also called derived class.

## Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

#### Example

Create a class named Person, with firstname and lastname properties, and a printname method:

class Person:

**def** \_\_init\_\_(*self*, **fname**, **lname**):

*self*.firstname **=** fname

*self*.lastname **=** lname

**def** printname(*self*):

    print(*self*.firstname, *self*.lastname)

*#Use the Person class to create an object, and then execute the printname method:*

x **=** Person("John", "Doe")

x.printname()

## Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

#### Example

Create a class named Student, which will inherit the properties and methods from the Person class:

class one:

**def** \_\_init\_\_(*self*,**x**):

*self*.x**=**x

**def** fun1(*self*):

        print('hello ',*self*.x)

class two(*one*):

**def** fun2(*self*):

        print('i am ',*self*.x)

lol**=**two('mahmud')

lol.fun2()

## Add the \_\_init\_\_() Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the \_\_init\_\_() function to the child class (instead of the pass keyword).

**Note:** The \_\_init\_\_() function is called automatically every time the class is being used to create a new object.

#### Example

Add the \_\_init\_\_() function to the Student class:

When you add the \_\_init\_\_() function, the child class will no longer inherit the parent's \_\_init\_\_() function.

**Note:** The child's \_\_init\_\_() function **overrides** the inheritance of the parent's \_\_init\_\_() function.

To keep the inheritance of the parent's \_\_init\_\_() function, add a call to the parent's \_\_init\_\_() function:

class one:

**def** \_\_init\_\_(*self*,**x**):

*self*.x**=**x

        print(' i am first init')

**def** fun1(*self*):

        print('hello ',*self*.x)

class two(*one*):

**def** \_\_init\_\_(*self*,**x**):

        one.\_\_init\_\_(*self*,x)

**def** fun2(*self*):

        print('i am ',*self*.x)

lol**=**two('mahmud')

lol.fun2()

Now we have successfully added the \_\_init\_\_() function, and kept the inheritance of the parent class, and we are ready to add functionality in the \_\_init\_\_() function.

## Use the super() Function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

#### Example

class Student(*Person*):

**def** \_\_init\_\_(*self*, **fname**, **lname**):

    super().\_\_init\_\_(fname, lname)

By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

class one:

**def** \_\_init\_\_(*self*):

        print('i am first init')

class two(*one*):

**def** \_\_init\_\_(*self*):

        super().\_\_init\_\_()

        print('i am second init')

lol**=**two()

*#when multiple overriding funtion ,use super alter method*

class one:

**def** \_\_init\_\_(*self*):

        print('i am first init')

class two:

**def** \_\_init\_\_(*self*):

        print('i am second init')

class three(*one*,*two*):

**def** \_\_init\_\_(*self*):

*#super().\_\_init\_\_()*

*#super().\_\_init\_\_()*

        one.\_\_init\_\_(*self*)

        two.\_\_init\_\_(*self*)

        print('i am third init')

lol**=**three()

# Python Classes and Objects

<https://www.geeksforgeeks.org/python-classes-and-objects/?ref=lbp>

A class is a user-defined blueprint or prototype from which objects are created.

#### Some points on Python class:

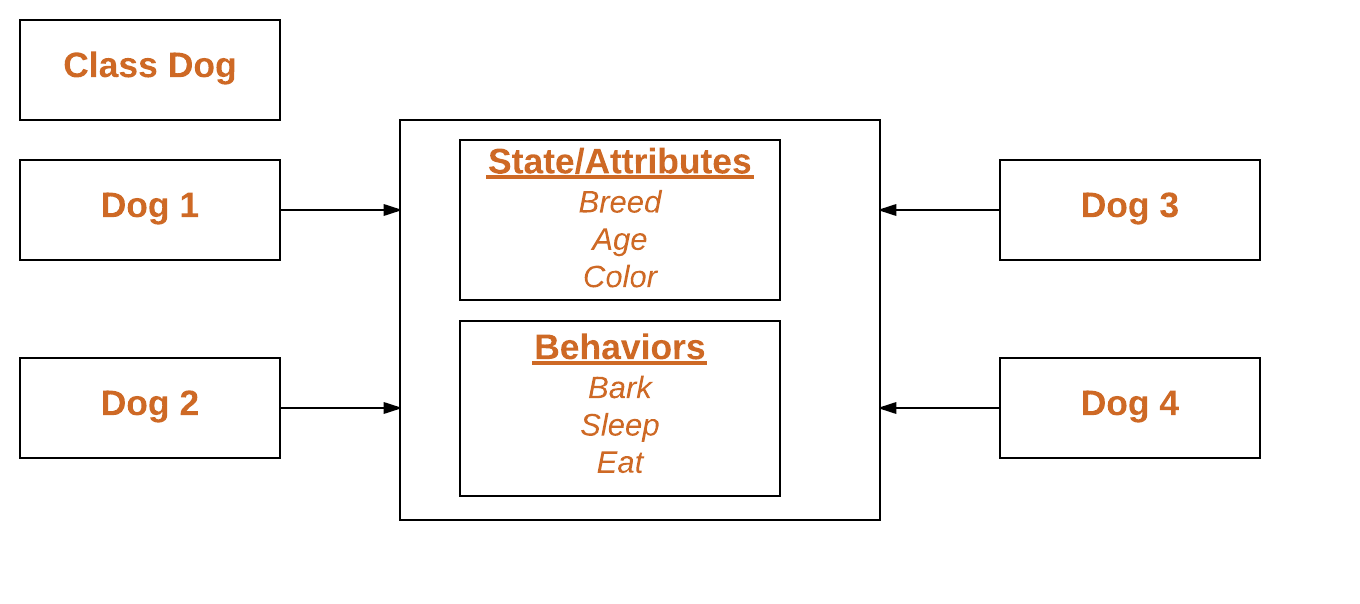
* Classes are created by keyword class.
* Attributes are the variables that belong to class.
* Attributes are always public and can be accessed using dot (.) operator. Eg.: Myclass.Myattribute

## Class Objects

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values.

#### Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.



#### The self

* Class methods must have an extra first parameter in method definition. We do not give a value for this parameter when we call the method, Python provides it.
* If we have a method which takes no arguments, then we still have to have one argument.
* This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

#### \_\_init\_\_ method

The \_\_init\_\_ method is similar to constructors in C++ and Java. Constructors are used to initialize the object’s state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

#### Class and Instance Variables

Instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.

Defining instance varibale using constructor.

## Constructors in Python

Last Updated: 19-04-2020

**Prerequisites :** [Object Oriented Programming in Python](https://www.geeksforgeeks.org/object-oriented-programming-in-python-set-1-class-and-its-members/), [Object Oriented Programming in Python | Set 2](https://www.geeksforgeeks.org/object-oriented-programming-in-python-set-2-data-hiding-and-object-printing/)  
Constructors are generally used for instantiating an object.The task of constructors is to initialize(assign values) to the data members of the class when an object of class is created.In Python the \_\_init\_\_() method is called the constructor and is always called when an object is created.

#### **Syntax of constructor declaration :**

def \_\_init\_\_(self):

# body of the constructor

### Types of constructors :

* **default constructor :**The default constructor is simple constructor which doesn’t accept any arguments.It’s definition has only one argument which is a reference to the instance being constructed.
* **parameterized constructor :**constructor with parameters is known as parameterized constructor.The parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

*#example default constructor*

class one:

**def** \_\_init\_\_(*self*):

        print('i am default constructor')

lol**=**one()

*#example parametarise contructor*

class two():

**def** \_\_init\_\_(*self*,**x**,**y**):

*self*.x**=**x

*self*.y**=**y

        print('i am parametarise constuctor',x**+**y)

lol**=**two(2,2)

## Destructors in Python

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically.  
The **\_\_**[**del**](https://www.geeksforgeeks.org/delattr-del-python/)**\_\_()** method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

#### Syntax of destructor declaration :

def \_\_del\_\_(self):

# body of destructor

class Employee:

*# Initializing*

**def** \_\_init\_\_(*self*):

        print('Employee created.')

*# Deleting (Calling destructor)*

**def** \_\_del\_\_(*self*):

        print('Destructor called, Employee deleted.')

obj **=** Employee()

**del** obj

**Note :** The destructor was called **after the program ended** or when all the references to object are deleted i.e when the reference count becomes zero, not when object went out of scope.

**Example 2 :**This example gives the explanation of above mentioned note. Here, notice that the destructor is called after the ‘Program End…’ printed.

*# Python program to illustrate destructor*

*# Python program to illustrate destructor*

class Employee:

*# Initializing*

**def** \_\_init\_\_(*self*):

        print('Employee created')

*# Calling destructor*

**def** \_\_del\_\_(*self*):

        print("Destructor called")

**def** Create\_obj():

    print('Making Object...')

    obj **=** Employee()

    print('function end...')

**return** obj

print('Calling Create\_obj() function...')

obj **=** Create\_obj()

print('Program End...')

**Example 3 :** Now, consider the following example :

*# Python program to illustrate destructor*

class A:

**def** \_\_init\_\_(*self*, **bb**):

*self*.b **=** bb

class B:

**def** \_\_init\_\_(*self*):

*self*.a **=** A(*self*)

**def** \_\_del\_\_(*self*):

        print("die")

**def** fun():

    b **=** B()

fun()

## Inheritance in Python