Python is a high-level, interpreted programming language that is widely used for many different purposes, from web development to scientific computing to machine learning.

There are several reasons why Python is a great choice for many different types of projects:

1. **Easy to learn:** Python has a relatively simple syntax, which makes it easier for beginners to learn compared to other programming languages.
2. **Versatile:** Python can be used for a wide range of applications, including web development, data analysis, artificial intelligence, and more.
3. **Large and active community:** There is a large and active community of Python developers, which means that there are many resources available for learning and solving problems.
4. **Plenty of libraries**: Python has a large number of libraries, including NumPy, pandas, and Matplotlib, which can be used to perform complex tasks with just a few lines of code.
5. **Cross-platform compatibility:** Python can run on multiple operating systems, including Windows, macOS, and Linux, making it a good choice for projects that need to be run on multiple platforms.

**Understanding Variables in Python**

Assignment operator (=) to assign new variables in python whether Boolean, float or integer.

Eg. Middle\_name =”josef Wambua”

Print(middle\_name)

# assigning a new value to middle\_name

Middle\_name = “Muindi”

Print(middle\_name)

Output

Josef Wambua

Muindi

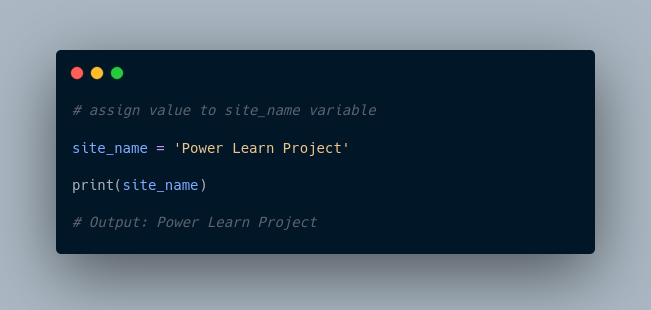
In programming, a variable is a container (storage area) to hold data. For example,

number = 10

Here, number is the variable storing the value **10**.

## Assigning values to Variables in Python

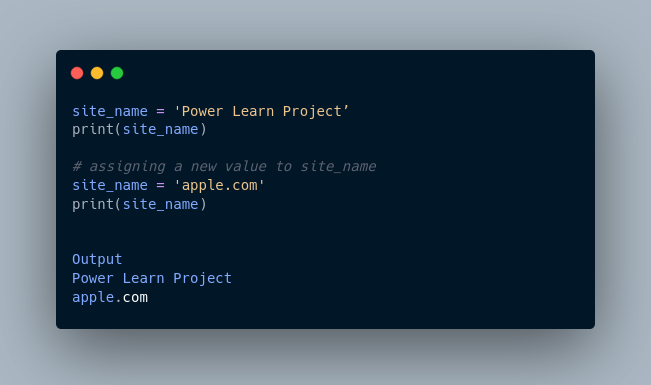
As we can see from the above example, we use the assignment operator = to assign a value to a variable.

﻿

In the above example, we assigned the value ‘Power Learn Project’ to the site\_name variable. Then, we printed out the value assigned to site\_name.

**Note**: Python is a type-inferred language, so you don't have to explicitly define the variable type. It automatically knows that Power Learn Projects is a string and declares the site\_name variable as a string.

## Changing the Value of a Variable in Python

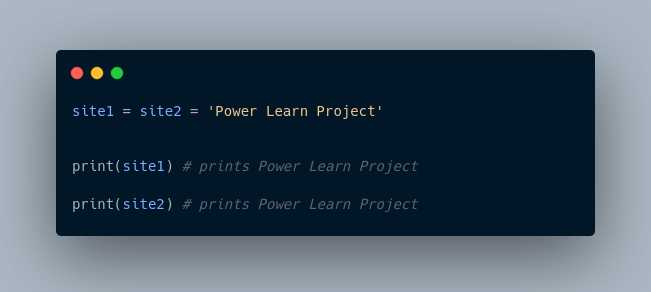


Here, the value of site\_name is changed from ‘Power Learn Project’ to 'apple.com'.

## Example: Assigning multiple values to multiple variables



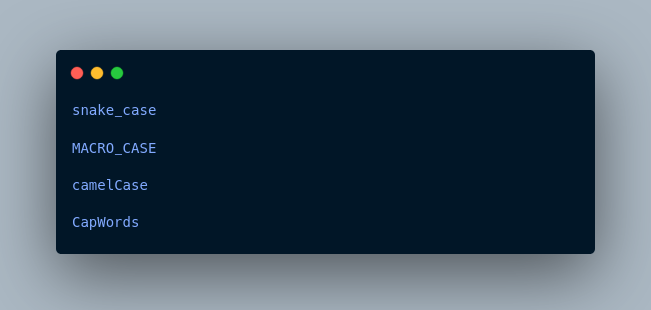
If we want to assign the same value to multiple variables at once, we can do this as:



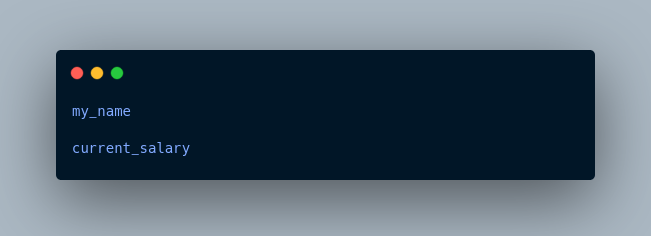
Here, we have assigned the same string value 'Power Learn Project' to both the variables site1 and site2.

## Rules for Naming Python Variables

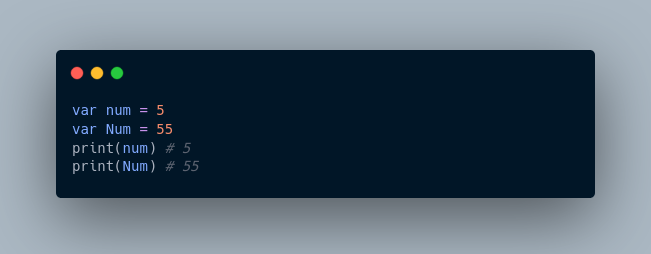
* Constant and variable names should have a combination of letters in lowercase (a to z) or uppercase (**A to Z**) or digits (**0 to 9**) or an underscore (**\_**). For example:



* Create a name that makes sense. For example, vowel makes more sense than v.
* If you want to create a variable name having two words, use underscore to separate them. For example:



* Python is case-sensitive. So num and Num are different variables. For example,



* Avoid using keywords like if, True, class, etc. as variable names.

More Resources:

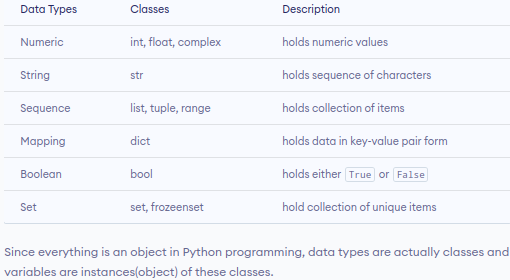
1. <https://realpython.com/python-variables/>
2. <https://www.simplilearn.com/tutorials/python-tutorial/python-variables>
3. <https://www.guru99.com/variables-in-python.html>
4. <https://www.tutorialspoint.com/python/python_variables.htm>

In computer programming, data types specify the type of data that can be stored inside a variable. For example,

num = 24

Here, **24** (an integer) is assigned to the num variable. So the data type of num is of the int class.

**Python Data Types**



Since everything is an object in Python programming, data types are actually classes and variables are instances(object) of these classes.

**Python Numeric Data type**

In Python, the numeric data type is used to hold numeric values.

Integers, floating-point numbers and complex numbers fall under Python numbers category. They are defined as int, float and complex classes in Python.

* int - holds signed integers of non-limited length.
* float - holds floating decimal points and it's accurate up to **15** decimal places.
* complex - holds complex numbers.

We can use the type() function to know which class a variable or a value belongs to.

Let's see an example,



In the above example, we have created three variables named num1, num2 and num3 with values **5**, **5.0**, and 1+2j respectively.

We have also used the type() function to know which class a certain variable belongs to.

Since,

* **5** is an integer value, type() returns int as the class of num1 i.e <class 'int'>
* **2.0** is a floating value, type() returns float as the class of num2 i.e <class 'float'>
* 1 + 2j is a complex number, type() returns complex as the class of num3 i.e <class 'complex'>

**Python List Data Type**

A list is an ordered collection of similar or different types of items separated by commas and enclosed within brackets [ ]. For example,



Here, we have created a list named languages with **3** string values inside it.

**Access List Items**

To access items from a list, we use the index number **(0, 1, 2 ...)**. For example,



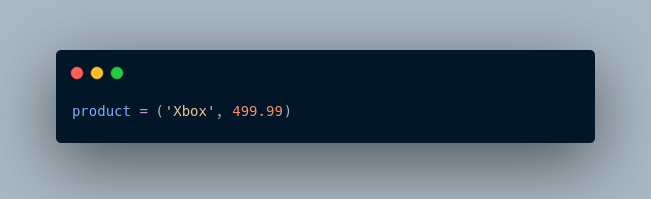
In the above example, we have used the index values to access items from the languages list.

* languages[0] - access first item from languages i.e. Swift
* languages[2] - access third item from languages i.e. Python

**Python Tuple Data Type**

A tuple is an ordered sequence of items same as a list. The only difference is that tuples are immutable. Tuples once created cannot be modified.

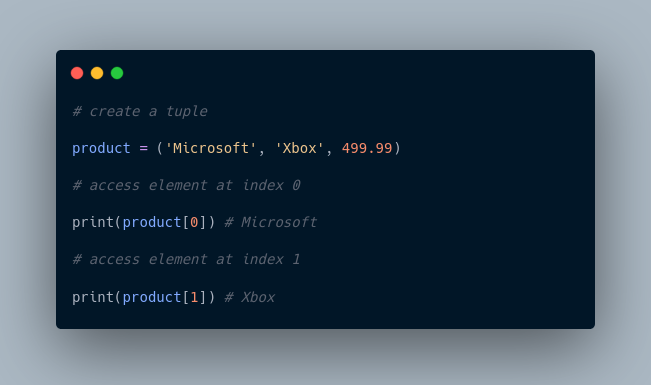
In Python, we use the parentheses () to store items of a tuple. For example,



Here, product is a tuple with a string value Xbox and integer value **499.99**.

**Access Tuple Items**

Similar to lists, we use the index number to access tuple items in Python. For example,



**Python String Data Type**

String is a sequence of characters represented by either single or double quotes. For example,



In the above example, we have created string-type variables: name and message with values 'Python' and 'Python for beginners' respectively.

**Python Set Data Type**

The Set is an unordered collection of unique items. Set is defined by values separated by commas inside braces { }. For example,



Here, we have created a set named student\_info with **5** integer values.

Since sets are unordered collections, indexing has no meaning. Hence, the slicing operator [] does not work.

**Python Dictionary Data Type**

Python dictionary is an ordered collection of items. It stores elements in key/value pairs.

Here, keys are unique identifiers that are associated with each value.

Let's see an example,



In the above example, we have created a dictionary named capital\_city. Here,

1. **Keys** are 'Nepal', 'Italy', 'England'
2. **Values** are 'Kathmandu', 'Rome', 'London'

**Access Dictionary Values Using Keys**

We use keys to retrieve the respective value. But not the other way around. For example,



Here, we have accessed values using keys from the capital\_city dictionary.

Since 'Nepal' is key, capital\_city['Nepal'] accesses its respective value i.e. Kathmandu

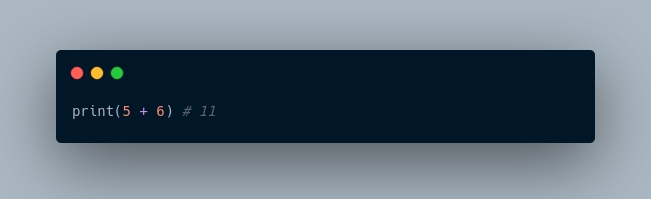
However, 'Kathmandu' is the value for the 'Nepal' key, so capital\_city['Kathmandu'] throws an error message.

More Resources:

1. <https://www.geeksforgeeks.org/python-data-types/>
2. <https://realpython.com/python-data-types/>
3. <https://www.digitalocean.com/community/tutorials/python-data-types>
4. <https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html>

In conclusion:  
Sets are enclosed using {}, tuples using (), list [],.. dictionary using parenthesis too

Operators are special symbols that perform operations on variables and values. For example,



Here, + is an operator that adds two numbers: **5** and **6**.

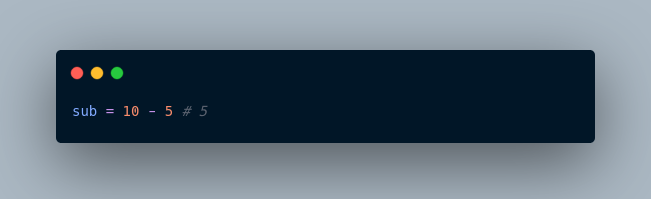
## ****Types of Python Operators****

Here's a list of different types of Python operators that we will learn in this tutorial.

1. Arithmetic operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. Bitwise Operators
6. Special Operators

### 1. **Python Arithmetic Operators**

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc. For example,



Here, - is an arithmetic operator that subtracts two values or variables.

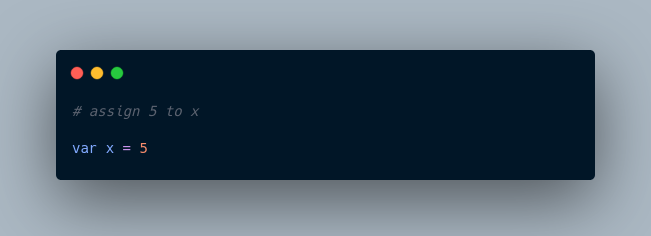
****

In the above example, we have used multiple arithmetic operators,

* + to add a and b
* - to subtract b from a
* \* to multiply a and b
* / to divide a by b
* // to floor divide a by b
* % to get the remainder
* \*\* to get a to the power b

### **2. Python Assignment Operators**

Assignment operators are used to assign values to variables. For example,



Here, = is an assignment operator that assigns 5 to x.

Here's a list of different assignment operators available in Python.

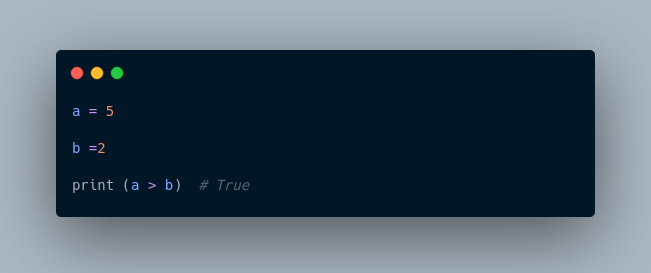
****

Here, we have used the += operator to assign the sum of a and b to a.

Similarly, we can use any other assignment operators according to the need.

### **3. Python Comparison Operators**

Comparison operators compare two values/variables and return a boolean result: True or False. For example,



Here, the > comparison operator is used to compare whether a is greater than b or not.

****

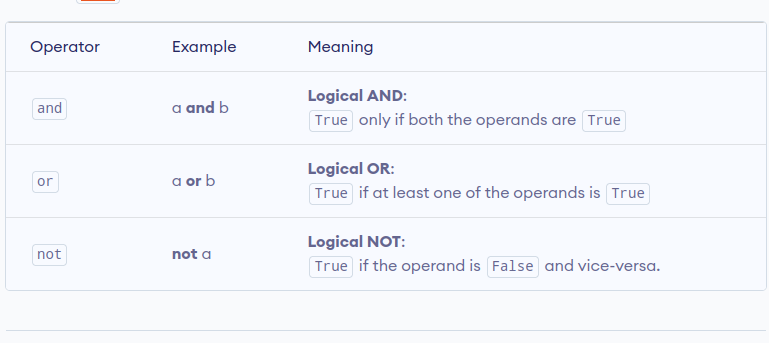
**Note:** Comparison operators are used in decision-making and loops. We'll discuss more of the comparison operator and decision-making in later tutorials.

### **4. Python Logical Operators**

Logical operators are used to check whether an expression is True or False. They are used in decision-making. For example,



Here, and is the logical operator **AND**. Since both a > 2 and b >= 6 are True, the result is True.

****

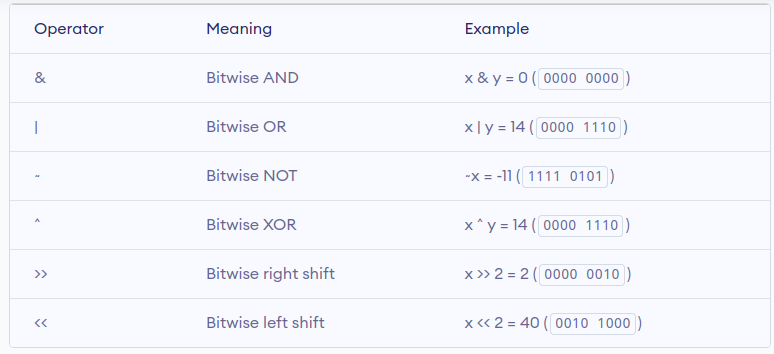
**Note**: Here is the truth table for these logical operators.

### **5. Python Bitwise operators**

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, **2** is 10 in binary and **7** is 111.

**In the table below:** Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

****

## ****6. Python Special operators****

Python language offers some special types of operators like the **identity** operator and the **membership** operator. They are described below with examples.

### **Identity operators**

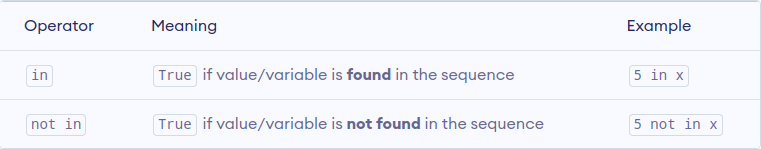
In Python, is and is not being used to check if two values are located on the same part of the memory. Two equal variables do not imply that they are identical.

****

## ****Membership operators****

In Python, in and not in are the membership operators. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set, and dictionary).

In a dictionary, we can only test for the presence of a key, not the value.

****

### **Example 5: Membership operators in Python**

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case-sensitive).

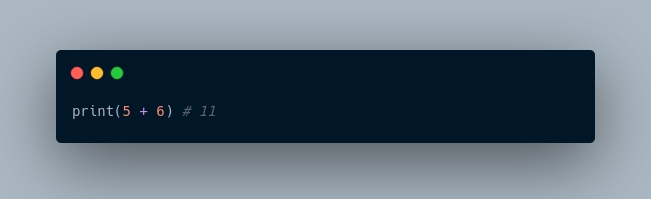
﻿

Similarly, 1 is the key, and 'a' is the value in the dictionary y. Hence, 'a' in y returns False.

More Resources:

1. <https://www.w3schools.com/python/python_operators.asp>
2. <https://www.freecodecamp.org/news/basic-operators-in-python-with-examples/>
3. <https://www.geeksforgeeks.org/python-operators/>

Operators are special symbols that perform operations on variables and values. For example,



Here, + is an operator that adds two numbers: **5** and **6**.

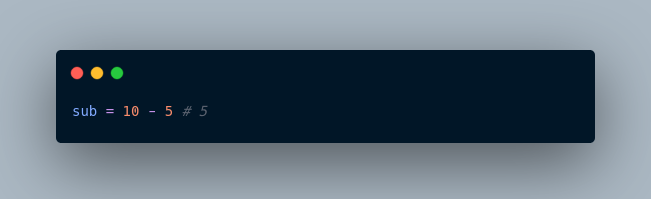
## ****Types of Python Operators****

Here's a list of different types of Python operators that we will learn in this tutorial.

1. Arithmetic operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. Bitwise Operators
6. Special Operators

### 1. **Python Arithmetic Operators**

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc. For example,



Here, - is an arithmetic operator that subtracts two values or variables.

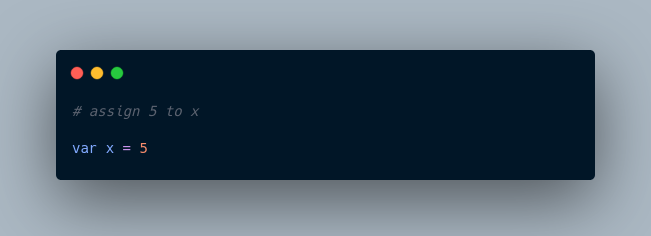
****

In the above example, we have used multiple arithmetic operators,

* + to add a and b
* - to subtract b from a
* \* to multiply a and b
* / to divide a by b
* // to floor divide a by b
* % to get the remainder
* \*\* to get a to the power b

### **2. Python Assignment Operators**

Assignment operators are used to assign values to variables. For example,



Here, = is an assignment operator that assigns 5 to x.

Here's a list of different assignment operators available in Python.

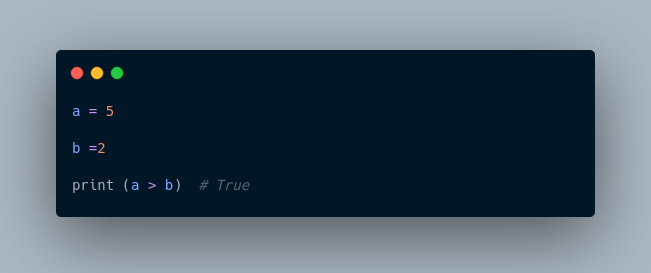
****

Here, we have used the += operator to assign the sum of a and b to a.

Similarly, we can use any other assignment operators according to the need.

### **3. Python Comparison Operators**

Comparison operators compare two values/variables and return a boolean result: True or False. For example,



Here, the > comparison operator is used to compare whether a is greater than b or not.

****

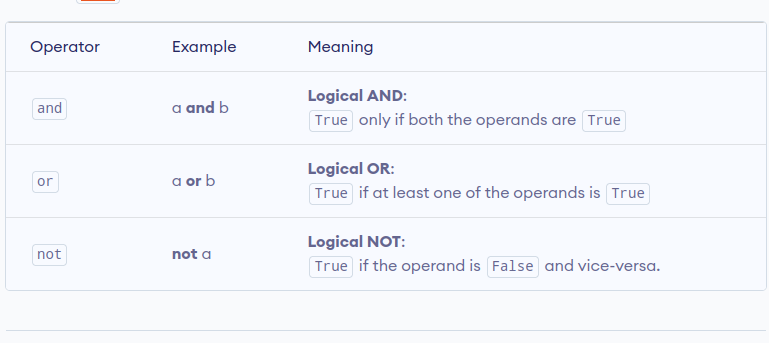
**Note:** Comparison operators are used in decision-making and loops. We'll discuss more of the comparison operator and decision-making in later tutorials.

### **4. Python Logical Operators**

Logical operators are used to check whether an expression is True or False. They are used in decision-making. For example,



Here, and is the logical operator **AND**. Since both a > 2 and b >= 6 are True, the result is True.

****

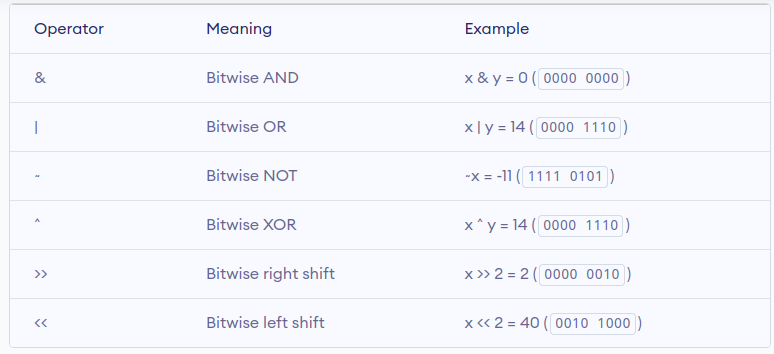
**Note**: Here is the truth table for these logical operators.

### **5. Python Bitwise operators**

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, **2** is 10 in binary and **7** is 111.

**In the table below:** Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

****

## ****6. Python Special operators****

Python language offers some special types of operators like the **identity** operator and the **membership** operator. They are described below with examples.

### **Identity operators**

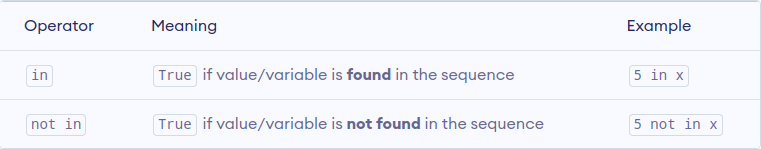
In Python, is and is not being used to check if two values are located on the same part of the memory. Two equal variables do not imply that they are identical.

****

## ****Membership operators****

In Python, in and not in are the membership operators. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set, and dictionary).

In a dictionary, we can only test for the presence of a key, not the value.

****

### **Example 5: Membership operators in Python**

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case-sensitive).

﻿

Similarly, 1 is the key, and 'a' is the value in the dictionary y. Hence, 'a' in y returns False.

More Resources:

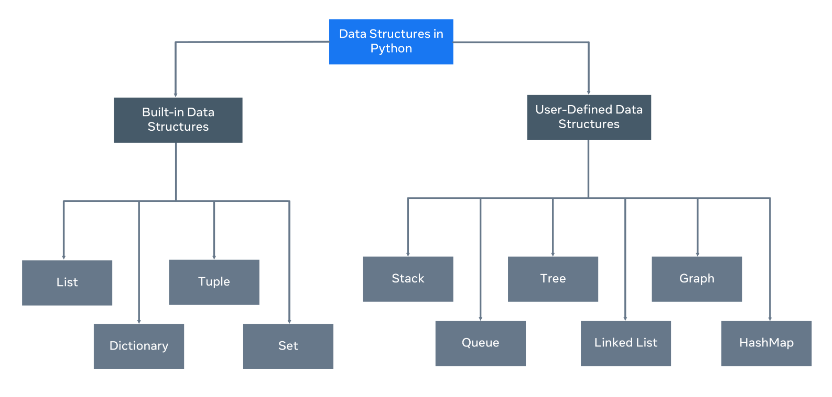
1. <https://www.w3schools.com/python/python_operators.asp>
2. <https://www.freecodecamp.org/news/basic-operators-in-python-with-examples/>
3. <https://www.geeksforgeeks.org/python-operators/>

**WEEK 2**

**Data structures**

So far, you have only stored small bits of data in a variable. This was either an integer, Boolean or a string.

But what happens if you need to work with more complex information, such as a collection of data like a list of people or a list of companies? Data structures are designed for this very purpose.

****

A data structure allows you to organize and arrange your data to perform operations on them. Python has the following built-in data structures: **List**, **dictionary**, **tuple** and **set**. These are all considered **non-primitive** data structures, meaning they are classed as objects, we will explore this more.

Along with the built-in data structures, Python allows users to create their own. Data structures such as Stacks, Queues and Trees can all be created by the user.

Each data structure can be designed to solve a particular problem or optimize a current solution to make it much more performant.

## Mutability and Immutability

Data Structures can be mutable or immutable.

**Mutability** refers to data inside the data structure that can be modified. For example, you can either change, update, or delete the data when needed. A list is an example of a mutable data structure.

An **immutable** data structure will not allow modification once the data has been set. The tuple is an example of an immutable data structure.

**Lists and Tuples**

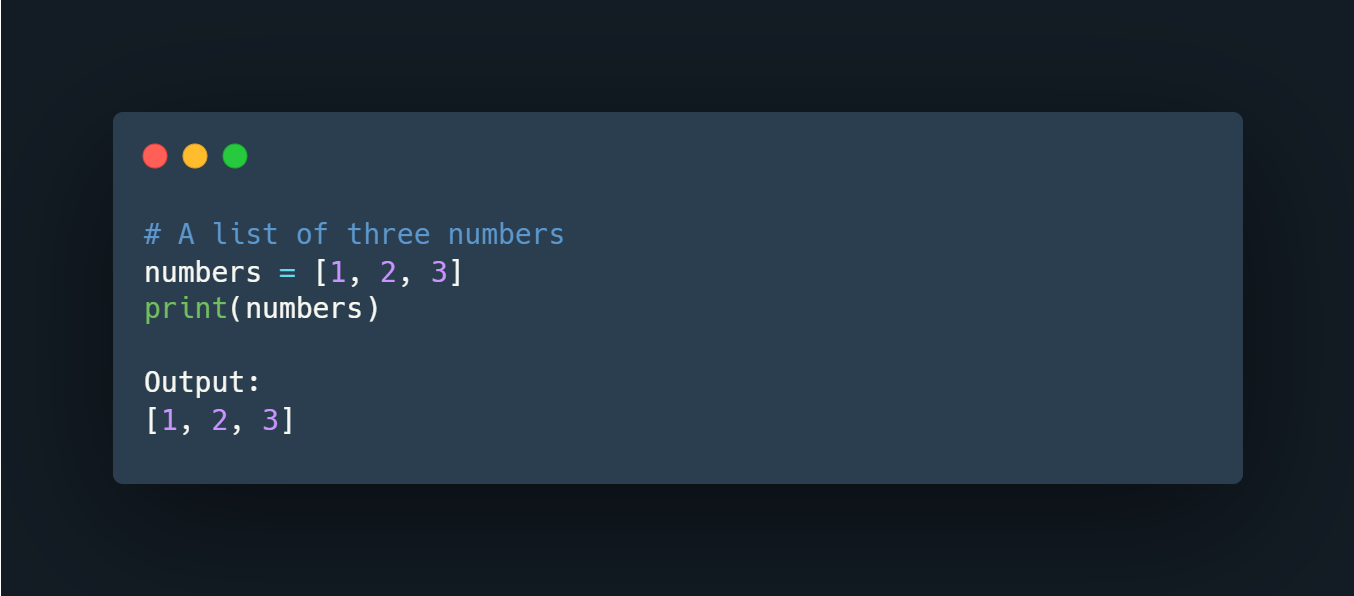
In Python, lists are used to store multiple data at once. For example,

Suppose we need to record the ages of **5** students. Instead of creating **5** separate variables, we can simply create a list:

Lists Elements

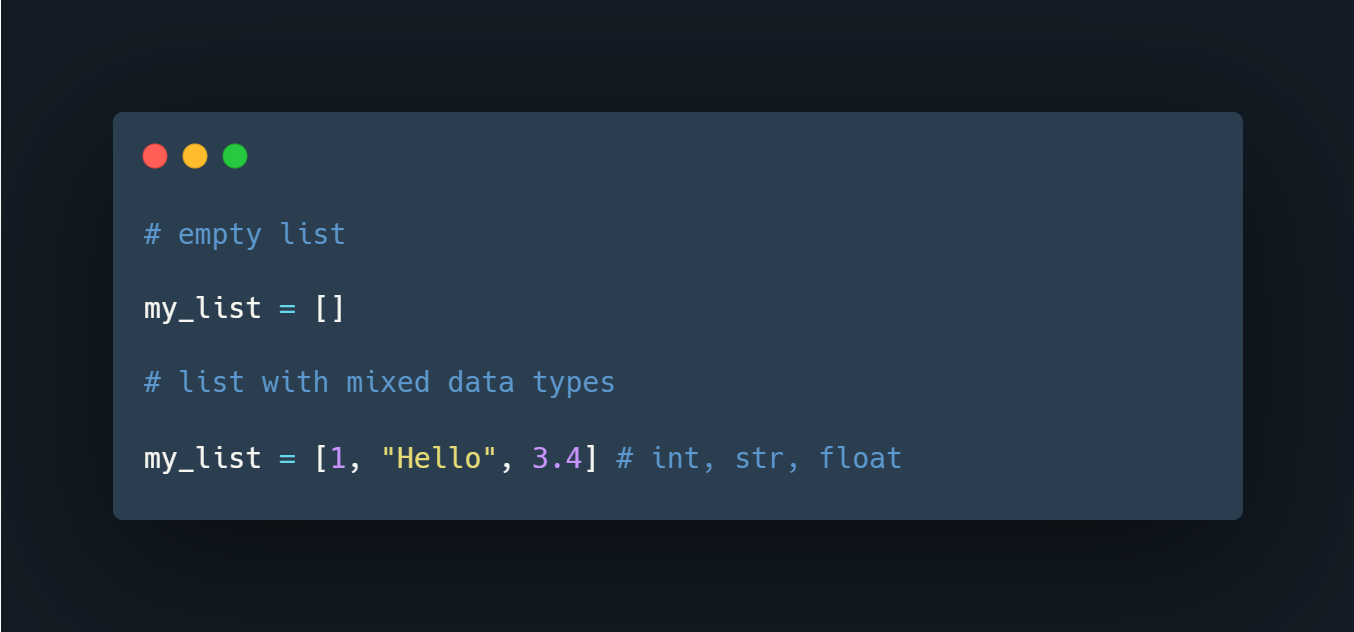
## ****Create a Python List****

A list is created in Python by placing items inside [], separated by commas. For example,



Here, we have created a list named numbers with **3** integer items.

A list can have any number of items and they may be of different types (integer, float, string, etc.). For example,



## ****Access Python List Elements****

In Python, each item in a list is associated with a number. The number is known as a list index.

We can access elements of an array using the index number **(0, 1, 2 …)**. For example,



In the above example, we have created a list named languages.



List Indexing in Python

Here, we can see each list item is associated with the index number. And, we have used the index number to access the items.

**Note:** The list index always starts with **0**. Hence, the first element of a list is present at index **0**, not **1**.

## ****Slicing of a Python List****

In Python, it is possible to access a section of items from the list using the slicing operator :, not just a single item. For example,



Here,

* my\_list[2:5] returns a list with items from index **2** to index **4**.
* my\_list[5:] returns a list with items from index **1** to the end.
* my\_list[:] returns all list items

**Note**: When we slice lists, the start index is inclusive but the end index is exclusive.

## ****Add Elements to a Python List****

Python List provides different methods to add items to a list.

**1. Using append()**

The append() method adds an item at the end of the list.

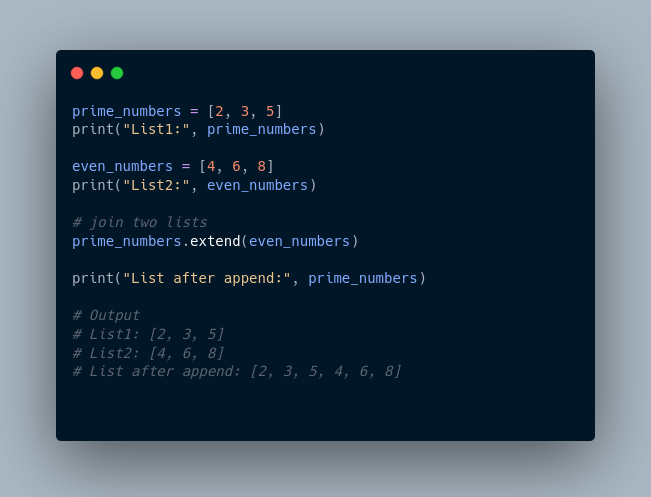
For example,



Here, append() adds **32** at the end of the array.

**2. Using extend()**

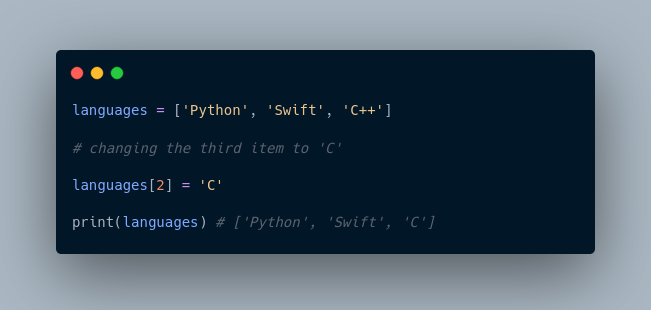
We use the extend() method to add all items of one list to another. For example,



In the above example, we have two lists named prime\_numbers and even\_numbers. Notice the statement,

## ****Change List Items****

Python lists are mutable. Meaning lists are changeable. And, we can change items of a list by assigning new values using = operator. For example,



## ****Remove an Item From a List****

**1. Using del()**

In Python, we can use the del statement to remove one or more items from a list. For example,



**2. Using remove()**

We can also use the remove() method to delete a list item.

For example:



Here, languages.remove('Python') removes 'Python' from the languages list.

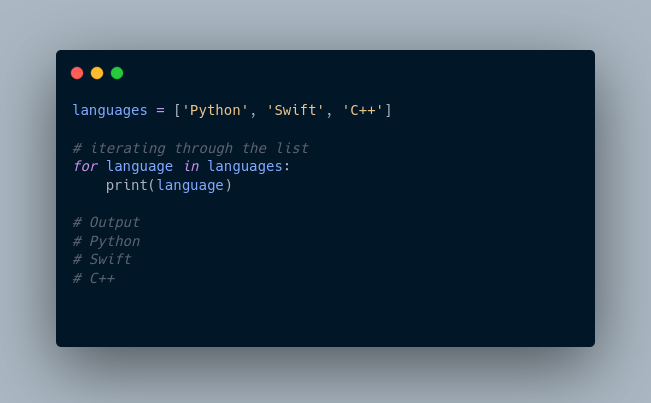
## ****Python List Methods****

Python has many useful list methods that make it really easy to work with lists.



## ****Iterating through a List****

We can use the for loop to iterate over the elements of a list. For example,

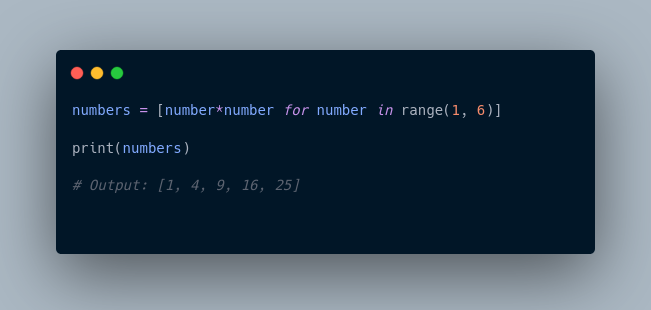


## ****Python List Comprehension****

List comprehension is a concise and elegant way to create lists.

A list comprehension consists of an expression followed by the for statement inside square brackets.

Here is an example to make a list with each item being increasing by power of **2.**

****

In the above example, we have used the list comprehension to make a list with each item being increased by power of **2.** Notice the code,



# **Tuples**

A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.

## ****Creating a Tuple****

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.

A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

## ****Create a Python Tuple With one Element****

In Python, creating a tuple with one element is a bit tricky. Having one element within parentheses is not enough.

We can use the type() function to know which class a variable or a value belongs to.



Here,

* ("hello") is a string so type() returns str as class of var1 i.e. <class 'str'>
* ("hello",) and "hello", both are tuples so type() returns tuple as class of var1 i.e. <class 'tuple'>

## ****Access Python Tuple Elements****

Like a list, each element of a tuple is represented by index numbers **(0, 1, ...)** where the first element is at index **0**.

We use the index number to access tuple elements. For example,

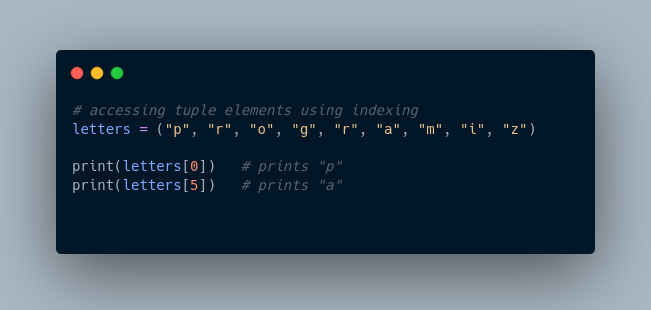
### **1. Indexing**

We can use the index operator [] to access an item in a tuple, where the index starts from 0.

So, a tuple having **6** elements will have indices from **0** to **5**. Trying to access an index outside of the tuple index range( **6,7,...** in this example) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result in TypeError.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.



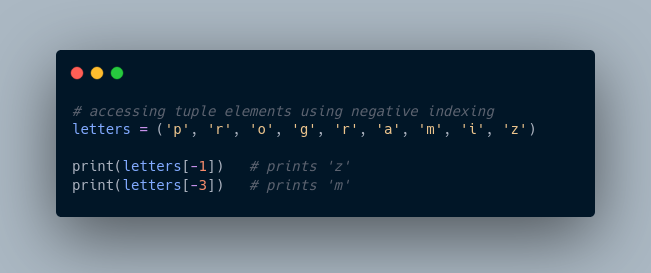
In the above example,

* letters[0] - accesses the first element
* letters[5] - accesses the sixth element

### **2. Negative Indexing**

Python allows negative indexing for its sequences.

The index of **-1** refers to the last item, **-2** to the second last item and so on. For example,



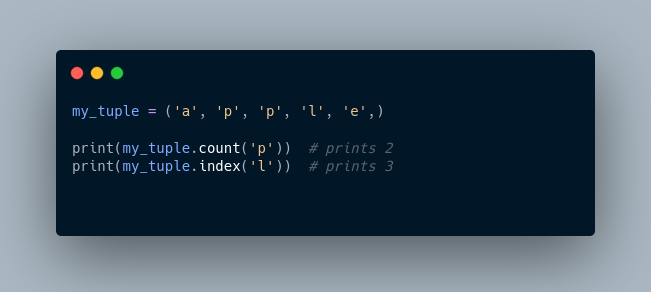
In the above example,

* letters[-1] - access last element
* letters[-3] - access third last element

## ****Python Tuple Methods****

In Python, methods that add items or remove items are not available with tuples. Only the following two methods are available.

Some examples of Python tuple methods:



Here,

* my\_tuple.count('p') - counts total number of 'p' in my\_tuple
* my\_tuple.index('l') - returns the first occurrence of 'l' in my\_tuple

**More Resources:**

1. <https://www.knowledgehut.com/tutorials/python-tutorial/python-lists-tuples>
2. <https://www.freecodecamp.org/news/python-tuple-vs-list-what-is-the-difference/>
3. <https://builtin.com/software-engineering-perspectives/python-tuples-vs-lists>
4. <https://realpython.com/python-lists-tuples/>

**>>> # A list of three numbers**

**>>> numbers = [1,2,3]**

**>>> print(numbers0**

**... ^X**

**KeyboardInterrupt**

**>>> print(numbers)**

**[1, 2, 3]**

**>>> # creating an empty list**

**>>> my\_list = []**

**>>> # list with mixed data types**

**>>> my\_list = [1, "Hello", 3.4]**

**>>> print(my\_list)**

**[1, 'Hello', 3.4]**

**>>> #list indexing**

**>>> languages = ["swift","ruby","java","python"]**

**>>> print(languages[1])**

**ruby**

**>>> #slicing - accessing a section of items from the list using the slicing operator**

**>>> languages = ["swift","ruby","java","python"]**

**>>> #access item at index 0**

**>>> print(languages[-1]) # python**

**python**

**>>> my\_list = ['p','r','o','g','r','a','m','i']**

**>>> # items from index 2 to 4**

**>>> print(my\_list[2:5])**

**['o', 'g', 'r']**

**>>> print(my\_list[5:])**

**['a', 'm', 'i']**

**>>> print(my\_list[:])**

**['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i']**

**>>> print(languages[-3])**

**ruby**

**>>>**

**Append ()- adding new values to a list**

**>>> #Add elements to a python list**

**>>> #Append()**

**>>> numbers = [2,53,35,34]**

**>>> print("Before Append:", numbers)**

**Before Append: [2, 53, 35, 34]**

**>>> #using append method**

**>>> numbers.append(304)**

**>>> print("After Append:", numbers)**

**After Append: [2, 53, 35, 34, 304]**

**>>>**

**Remove() – delete item from a list**

#using remove - to delete a list item

>>> languages = ["python","r","java","rust","c++","|"]

>>> languages.remove("|")

>>> print(languages)

['python', 'r', 'java', 'rust', 'c++']

**Iterating through a List**

languages =["java","cpp","swift"]

>>> for language in languages:

... print(language)

...

java

cpp

swift

>>>

**Python list comprehension**

#python list comprehension

>>> #example to make a list with each item being increasing by power of 2.

>>>

>>> numbers= [number\*number for number in range(1,6)]

>>> print(numbers)

[1, 4, 9, 16, 25]

>>>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Tuples**

>>> #Tuples

>>> #Tuples are immutable meaning that once theyre are created cannot be modified

≫>The index of **-1** refers to the last item, **-2** to the second last item and so on

>>> # accessing tuple elements using indexing

>>> letters =("p","r","o","g","r","a","m","i","z")

>>> print(letters[0])

p

>>> # negative indexing

>>> print(letters[-1])

z

>>>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Python tuple methods**

>>> #python tuple methods

>>> my\_tuple = ('a','p','p','l','e',)

>>> print(my\_tuple.count('p'))

2

>>> print(my\_tuple.count('l'))

1

>>> print(my\_tuple.index('l'))

3

>>>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Dictionaries**

Let's see an example,

If we want to store information about countries and their capitals, we can create a dictionary with country names as **keys** and capitals as **values**.

## Create a dictionary in Python

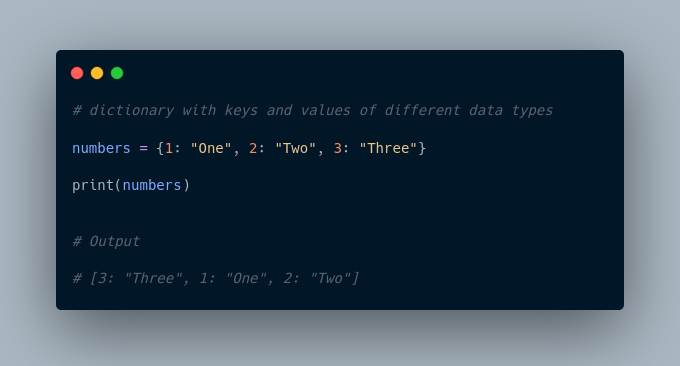
Here's how we can create a dictionary in Python.

In the above example, we have created a dictionary named capital\_city. Here,

1. **Keys** are "Nepal", "Italy", "England"
2. **Values** are "Kathmandu", "Rome", "London"

**Note**: Here, **keys** and **values** both are of string type. We can also have **keys** and **values** of different data types.

## Example 1: Python Dictionary



In the above example, we have created a dictionary named numbers. Here, **keys** are of integer type and **values** are of string type.

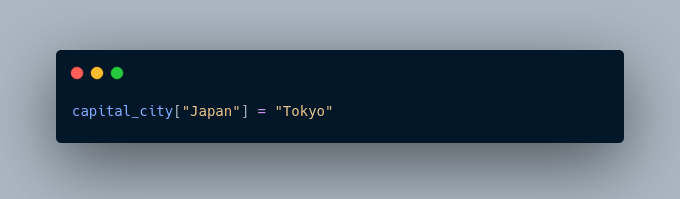
## Add Elements to a Python Dictionary

We can add elements to a dictionary using the name of the dictionary with [].

For example,



In the above example, we have created a dictionary named capital\_city. Notice the line,



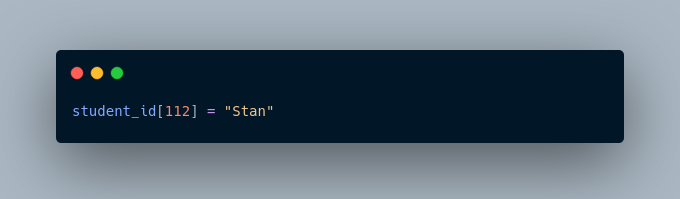
Here, we have added a new element to capital\_city with **key**: Japan and **value**: Tokyo.

## Change Value of Dictionary

We can also use [] to change the value associated with a particular key. For example,



In the above example, we have created a dictionary named student\_id. Initially, the value associated with the key 112 is "Kyle". Now, notice the line,



Here, we have changed the value associated with the key 112 to "Stan".

## Accessing Elements from Dictionary

In Python, we use the keys to access their corresponding values. For example,



Here, we have used the keys to access their corresponding values.

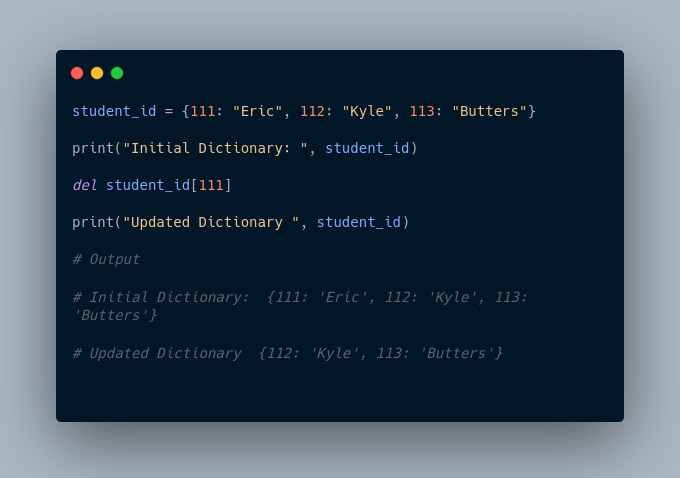
If we try to access the value of a key that doesn't exist, we'll get an error.

For example,

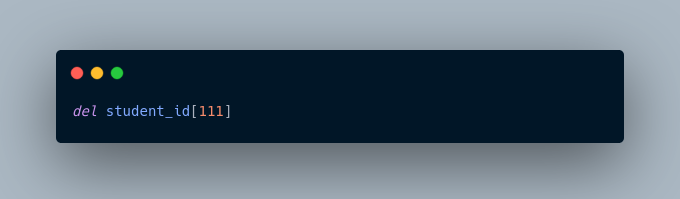


## Removing elements from Dictionary

We use the del statement to remove an element from the dictionary. For example,

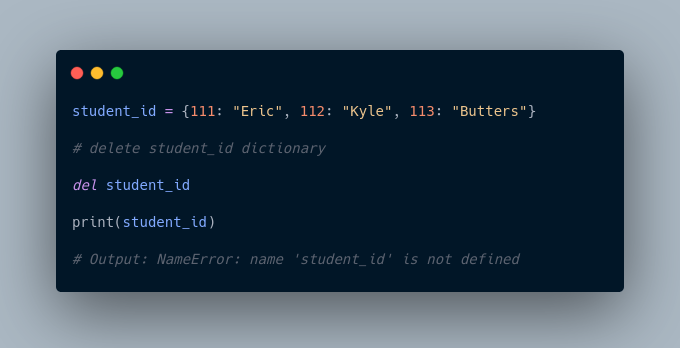


Here, we have created a dictionary named student\_id. Notice the code,



The del statement removes the element associated with the key 111.

We can also delete the whole dictionary using the del statement,



We are getting an error message because we have deleted the student\_id dictionary and student\_id doesn't exist anymore.

## Python Dictionary Methods

Methods that are available with a dictionary are tabulated below. Some of them have already been used in the above examples.

## Dictionary Membership Test

We can test if a key is in a dictionary or not using the keyword in. Notice that the membership test is only for the keys and not for the values.



## Iterating Through a Dictionary

We can iterate through each key in a dictionary using a loop.



﻿

Here, we have iterated through each **key** in the squares dictionary using the for loop.

More Resources:

1. <https://www.w3schools.com/python/python_dictionaries.asp>
2. <https://realpython.com/python-dicts/>

Python dictionary is an ordered collection (starting from **Python 3.7**) of items. It stores elements in **key/value** pairs. Here, **keys** are unique identifiers that are associated with each **value**.

**Used for storing key value pairs**

>>> #Dictionaries are to used to store key pair values

>>> capital\_city = {"Nepal": "Kathmandu", "Italy": "rome","Kenya": "Nairobi"}

>>> print(capital\_city)

{'Nepal': 'Kathmandu', 'Italy': 'rome', 'Kenya': 'Nairobi'}

>>>

1. \_\_\_\_\_\_\_\_\_\_\_\_**Keys** are "Nepal", "Italy", "England"
2. **Values** are "Kathmandu", "Rome", "London"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**#Add elemenents to a dictionary**

>>> capital\_city = {"Nepal": "Kathmandu", "England": "London"}

>>> print("Initial Dictionary: ", capital\_city)

Initial Dictionary: {'Nepal': 'Kathmandu', 'England': 'London'}

>>> capital\_city["Kenya"] = "Nairobi"

>>> print("Updated Dictionary: ", capital\_city)

Updated Dictionary: {'Nepal': 'Kathmandu', 'England': 'London', 'Kenya': 'Nairobi'}

>>>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Changing key value pairs / changing a dictionary**

# Dictionaries in python

# Changing value of dictionary

student\_id = {111: "Eric", 112:"Kyle",113:"joe"}

print("Initial dictionary: ", student\_id)

del student\_id[111]

print("updated Dictionary", student\_id)

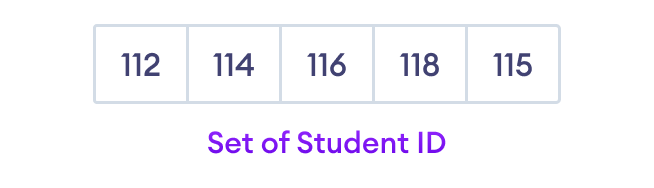
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Python sets**

A set is a collection of unique data. That is, elements of a set cannot be duplicated.

For example,

Suppose we want to store information about **student IDs**. Since **student IDs** cannot be duplicated, we can use a set.



Python Set Elements

## ****Create a Set in Python****

In Python, we create sets by placing all the elements inside curly braces {}, separated by comma.

A set can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

Let's see an example,



In the above example, we have created different types of sets by placing all the elements inside the curly braces {}.

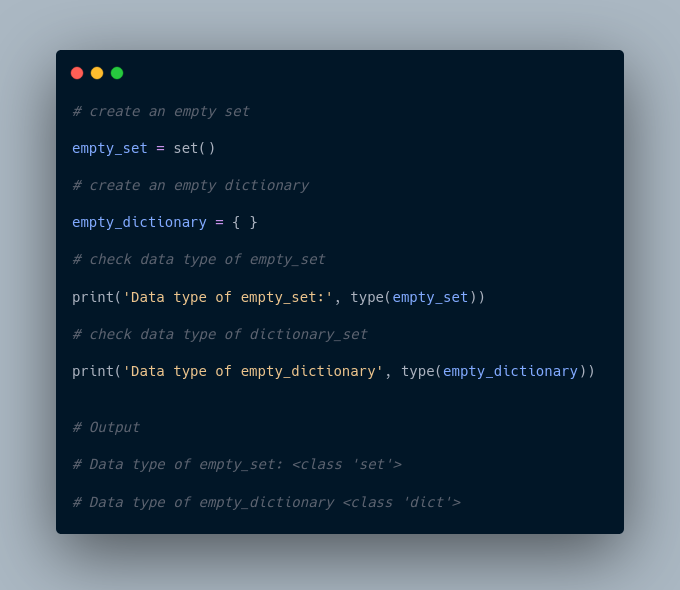
**Note:** When you run this code, you might get output in a different order. This is because the set has no particular order.

## ****Create an Empty Set in Python****

Creating an empty set is a bit tricky. Empty curly braces {} will make an empty dictionary in Python.

To make a set without any elements, we use the set() function without any argument.

For example,



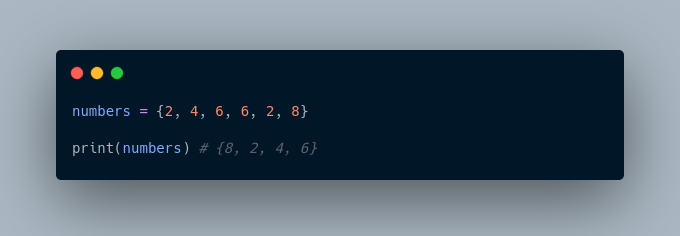
Here,

* empty\_set - an empty set created using set()
* empty\_dictionary - an empty dictionary created using {}

Finally we have used the type() function to know which class empty\_set and empty\_dictionary belong to.

## ****Duplicate Items in a Set****

Let's see what will happen if we try to include duplicate items in a set.



Here, we can see there are no duplicate items in the set as a set cannot contain duplicates.

## ****Add and Update Set Items in Python****

Sets are mutable. However, since they are unordered, indexing has no meaning.

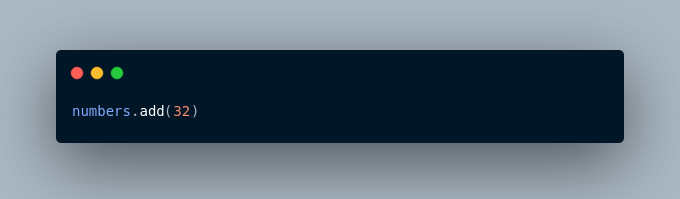
We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

### **Add Items to a Set in Python**

In Python, we use the add() method to add an item to a set. For example,



In the above example, we have created a set named numbers. Notice the line,

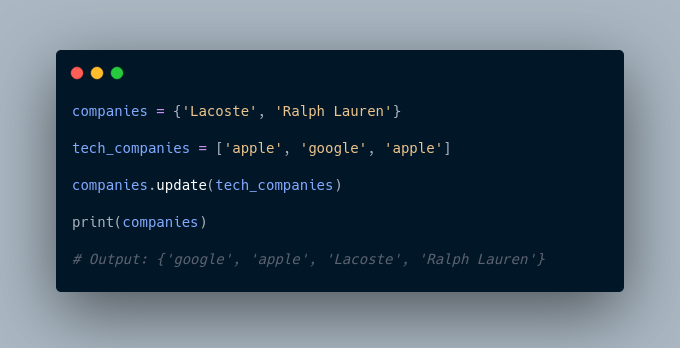


Here, add() adds **32** to our set.

### **Update Python Set**

The update() method is used to update the set with items other collection types (lists, tuples, sets, etc).

For example,

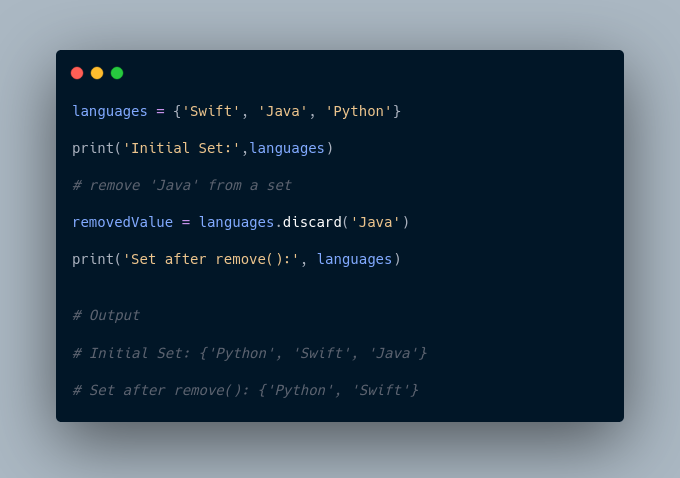


Here, all the unique elements of tech\_companies are added to the company's set.

## ****Remove an Element from a Set****

We use the discard() method to remove the specified element from a set.

For example,



Here, we have used the discard() method to remove 'Java' from the languages set.

## ****Built-in Functions with Set****

Built-in functions like all(), any(), enumerate(), len(), max(), min(), sorted(), sum() etc. are commonly used with sets to perform different tasks.



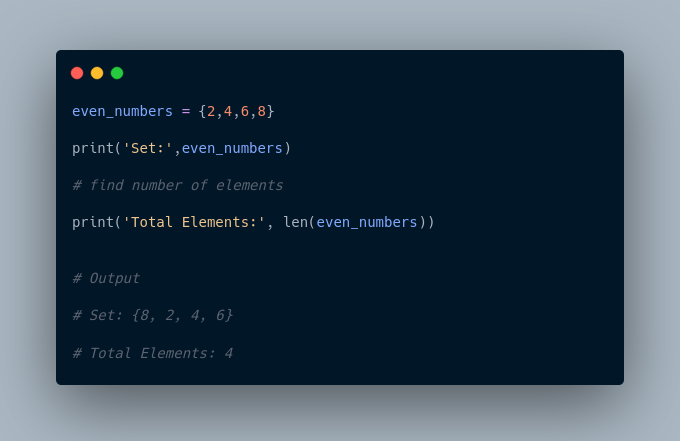
## ****Iterate Over a Set in Python****

****

## ****Find Number of Set Elements****

We can use the len() method to find the number of elements present in a Set.

For example,



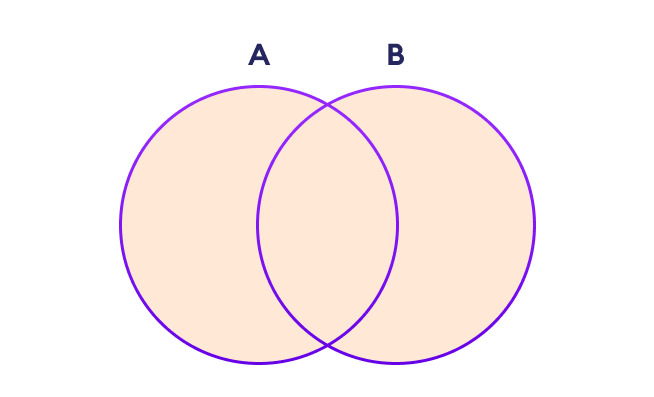
Here, we have used the len() method to find the number of elements present in a Set.

## ****Python Set Operations****

Python Set provides different built-in methods to perform mathematical set operations like union, intersection, subtraction, and symmetric difference.

## ****Union of Two Sets****

The union of two sets **A** and **B** include all the elements of set **A** and **B**.



Set Union in Python

We use the | operator or the union() method to perform the set union operation.

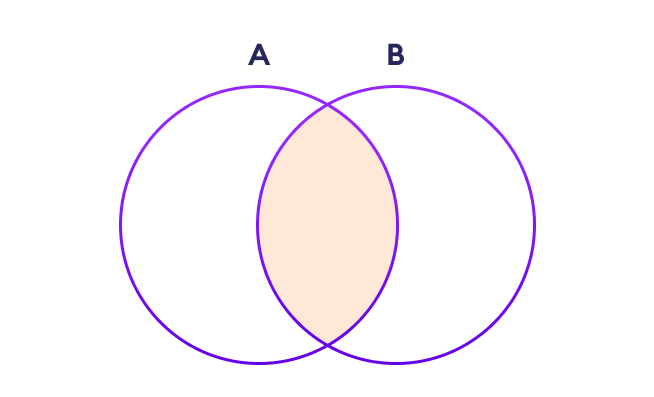
For example,



**Note**: A|B and union() is equivalent to A ⋃ B set operation.

## ****Set Intersection****

The intersection of two sets **A** and **B** include the common elements between set **A** and **B**.



Set Intersection in Python

In Python, we use the & operator or the intersection() method to perform the set intersection operation.

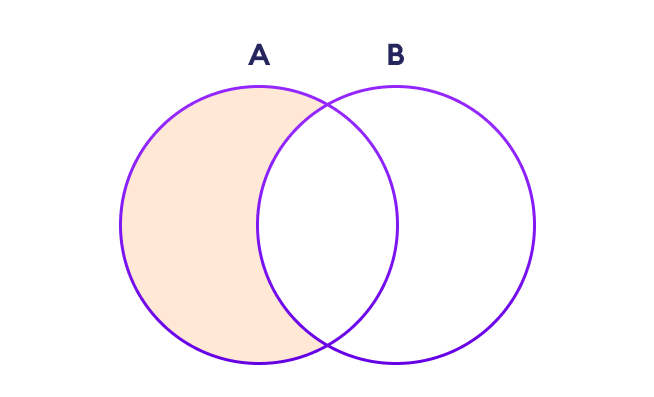
For example,



**Note**: A&B and intersection() is equivalent to A ⋂ B set operation.

## ****Difference between Two Sets****

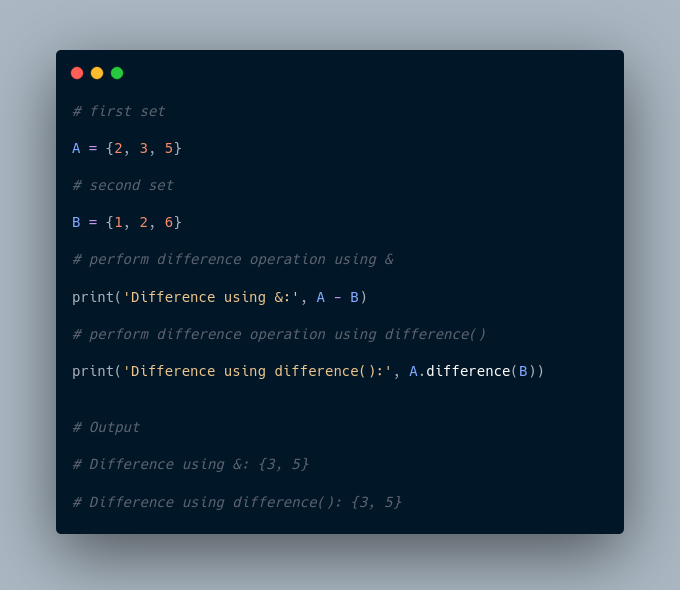
The difference between two sets **A** and **B** include elements of set **A** that are not present on set **B**.



Set Difference in Python

We use the - operator or the difference() method to perform the difference between two sets.

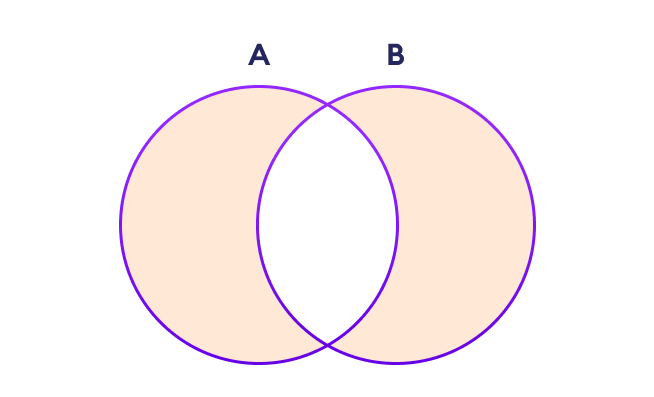
For example,



**Note**: A - B and A.difference(B) is equivalent to A - B set operation.

## ****Set Symmetric Difference****

The symmetric difference between two sets **A** and **B** includes all elements of **A** and **B** without the common elements.



Set Symmetric Difference in Python

In Python, we use the ^ operator or the symmetric\_difference() method to perform symmetric difference between two sets.

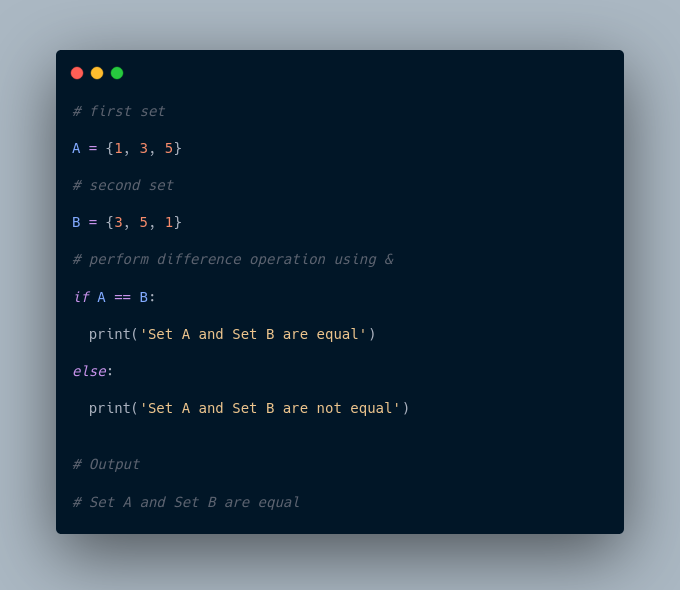
For example,



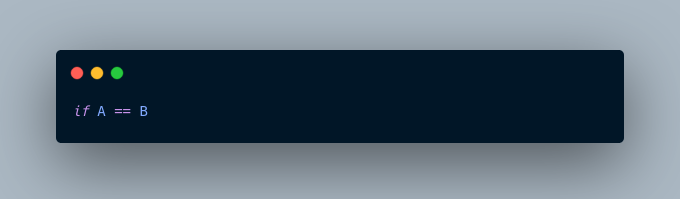
## ****Check if two sets are equal****

We can use the == operator to check whether two sets are equal or not.

For example,



In the above example, A and B have the same elements, so the condition

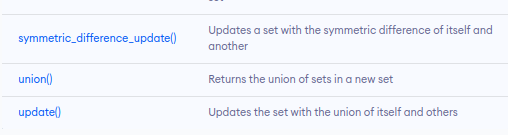


evaluates to True. Hence, the statement print('Set A and Set B are equal') inside the if is executed.

## ****Other Python Set Methods****

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with the set objects:





More Resources:

1. <https://www.geeksforgeeks.org/sets-in-python/>
2. <https://realpython.com/python-sets/>

**Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32**

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

**>>> #set**

**>>> # A set is a collection of unique date. The elements of a set cannot be duplicated**

**>>> #Eg student id cannot be duplicated**

**>>> # A set can have any number of items and they may be of different types(integer, float, tuple, string) but it cannot have a mutable elements like lists, sets or ductionaries**

**>>> #eg**

**>>> #sets are created by placing all the elements inside curly braces {}, separated by a comma**

**>>> #create a set of integer type**

**>>> std\_id = {11,12,13,14,15,16}**

**>>> print('Student ID:', std\_id)**

**Student ID: {16, 11, 12, 13, 14, 15}**

**>>> #create a set of string type**

**>>> vowel\_letters ={'a','e','i','o','u'}**

**>>> print('Vowel Letters:', vowel\_letters)**

**Vowel Letters: {'u', 'o', 'i', 'e', 'a'}**

**>>> #create a set of mixed data types**

**>>> mixed\_set ={"Hi", "I love coding wooow",1,2,3,"Byeeee"}**

**>>> print('Set of mixed data types: ', mixed\_set)**

**Set of mixed data types: {1, 'I love coding wooow', 3, 2, 'Byeeee', 'Hi'}**

**>>> #The output displayed has no order since sets do not have an order**

**>>> #To create an empty set we use set() function without any argument**

**>>> #create empty set**

**>>> empty\_set = set ()**

**>>> #create empty dictionary**

**>>> empty\_dictionary ={ }**

**>>> #check data type of empty set**

**>>> print('Data type of empty\_set:', type(empty\_set))**

**Data type of empty\_set: <class 'set'>**

**>>> #check data type of dictionary set**

**>>> print('Data type of empty\_dictionary:', type(empty\_dictionary))**

**Data type of empty\_dictionary: <class 'dict'>**

**>>> # create duplicate items. nb sets dont have duplicates**

**>>> number =[1,3,4,2,2,1}**

**File "<stdin>", line 1**

**number =[1,3,4,2,2,1}**

**^**

**SyntaxError: closing parenthesis '}' does not match opening parenthesis '['**

**>>> numbers ={1,2,3,2,3,2,32,2}**

**>>> print(numbers)**

**{32, 1, 2, 3}**

**>>> #Add items to a set in python**

**>>> numbers={1,2,43,2,3,4}**

**>>> numbers.add(100)**

**>>> print('Updated set:', numbers)**

**Updated set: {1, 2, 3, 4, 100, 43}**

**>>> #Update python set**

**>>> # we use update() method**

**>>> companies = {'lacoste', 'Ralph Lauren'}**

**>>> tech\_companies = ['apple','google','apple']**

**>>> companies.update(tech\_companies)**

**>>> print(companies)**

**{'Ralph Lauren', 'apple', 'google', 'lacoste'}**

**>>> #Remove an element from a set**

**>>> languages ={'java', 'swift','python','r','scala'}**

**>>> print('Initial set:',languages)**

**Initial set: {'r', 'java', 'swift', 'python', 'scala'}**

**>>> #we use discard() method to remove an element from a set**

**>>> remove\_element = languages.discard('java')**

**>>> print('Set after remove():', languages)**

**Set after remove(): {'r', 'swift', 'python', 'scala'}**

**>>> #Iterate over a set**

**>>> fruits = {"apple","mangoes","peach","dates"}**

**>>> #for loop to access each fruit**

**>>> for fruit in fruits:**

**... print(fruit)**

**...**

**apple**

**mangoes**

**peach**

**dates**

**>>> #Find number of set elements**

**>>> even\_numbers = {2,4,6,8}**

**>>> print('set:', even\_numbers)**

**set: {8, 2, 4, 6}**

**>>> #find number of elements**

**>>> print('Total element:', len(even\_numbers))**

**Total element: 4**

**>>> #Python set operations**

**>>> #Python set provides different built-in methods to perform mathematical set operations like union, intersection, subtraction and symmetric difference**

**>>> #set union.. we use | operator or the union() method to perform the set union operation**

**>>> #first ser**

**>>> A = {1,3,5}**

**>>> #second set**

**>>> B={0,2,4}**

**>>> #perform union operation using |**

**>>> print('union using |:', A | B)**

**union using |: {0, 1, 2, 3, 4, 5}**

**>>> #perform union using union()**

**>>> print('Union method using Union():',A.union(B))**

**Union method using Union(): {0, 1, 2, 3, 4, 5}**

**>>> #Set intersection - the intersection of two sets A and B include the common elements between set A and B**

**>>> #First set**

**>>> A={1,3,5}**

**>>> #second set**

**>>> B ={1,2,3}**

**>>> #perform set intersection using ambassand operant (&)**

**>>> print('intersection using &:', A & B)**

**intersection using &: {1, 3}**

**>>> # using intersection method**

**>>> print('intersection using interesection():',A.intersection(B))**

**intersection using interesection(): {1, 3}**

**>>> #Set symmetric Difference in python**

**>>> #We use ^ operator or the the symmetric\_difference() method**

**>>> #first set**

**>>> A={2,3,5}**

**>>> #second set**

**>>> B={1,2,6}**

**>>> #perform the difference using the ^**

**>>> print('Difference using^:', A ^ B)**

**Difference using^: {1, 3, 5, 6}**

**>>> # Using difference function**

**>>> print('Using symmetric\_difference():', A.symmetric\_difference(B))**

**Using symmetric\_difference(): {1, 3, 5, 6}**

**>>> #check if two symmetric sets are equal**

**>>> #first set**

**>>> A={1,3,5}**

**>>> #second set**

**>>> B={3,5,1}**

**>>> #perform difference operation using ==**

**>>> if A == B:**

**... print('Set A and B are equal')**

**... else:**

**... print('Set A and B are not equal')**

**...**

**Set A and B are equal**

**>>>**

**Python String and String Methods**

In computer programming, a string is a sequence of characters.

For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

## ****Object Oriented Programming****

Object-Oriented Programming (OOP) is a programming paradigm in computer science that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects.

## ****Building blocks of OOP****

* Classes - a blueprint of an object
* Objects - an instance of a class
* Methods - methods represent behaviors
* Attributes - information to be stored in a class about an object

## ****Four Principles of OOP****

## The four pillars of object-oriented programming are:

## ****Inheritance****: child classes inherit data and behaviors from the parent class

## ****Encapsulation****: containing information in an object, exposing only selected information

## ****Abstraction****: only exposing high-level public methods for accessing an object

## ****Polymorphism****: many methods can do the same task

## ****What are the benefits of OOP?****

## Modularity.

* Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.

## Reusability.

* Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.

## Productivity.

* Programmers can construct new programs quickly through the use of multiple libraries and reusable code.

## Easily upgradable and scalable.

* Programmers can implement system functionalities independently.

## Interface descriptions.

* Descriptions of external systems are simple, due to message-passing techniques that are used for object communication.

## Security.

* Using encapsulation and abstraction, complex code is hidden, software maintenance is easier and [internet protocols](https://www.techtarget.com/searchunifiedcommunications/definition/Internet-Protocol) are protected.

## ****Flexibility****.

* Polymorphism enables a single function to adapt to the class it is placed in. Different objects can also pass through the same interface.

**Class Attributes**

Class attributes are the variables defined directly in the class that are shared by all objects of the class. Class attributes can be accessed using the class name as well as using the objects.

class Person:

    name = 'Skinny’ #Class attribute

Above, the name is a class attribute defined inside a class Person. The value of the name will remain the same for all the objects unless modified explicitly.

**Accessing class attributes**

>>> Person.name

'Skinny'

>>> details = Person()

>>> details.name

'Skinny'

**Constructor**

In Python, the constructor method is invoked automatically whenever a new object of a class is instantiated, same as constructors in C++ or Java.

The constructor must have a special name \_\_init\_\_() and a special parameter called self.

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

The constructor in Python is used to define the attributes of an instance and assign values to them.

NB: The first parameter of each method in a class must be the self, which refers to the calling object. However, you can give any name to the first parameter, not necessarily self.

Example of how to define a constructor:

class Person:

    def \_\_init\_\_(self): # constructor method

        print('Constructor invoked')

Now, whenever you create an object of the Person class, the \_\_init\_\_() constructor method will be called, as shown below.

>>>details1 = Person()

Constructor invoked

>>>details2 = Person()

Constructor invoked

**Instance Attributes**

Instance attributes are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.

The following example defines instance attributes name and age in the constructor.

class Person:

    nationality = 'Ethiopian' # class attribute

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

        self.name = '' # instance attribute

        self.age = 0 # instance attribute

To access an instance variable, we use dot notation:

* [instance name].[attribute name], as shown below

>>> p1 = Person()

>>> p1.name

''

>>> p1.age

0

You can set the value of attributes using the dot notation, as shown below.

>>> p1 = Person()

>>> p1.name = "Mutemi" # assign value to instance attribute

>>> p1.age = 65    # assign value to instance attribute

>>> p1.name     # access instance attribute value

Mutemi

>>> std.age     # access value to instance attribute

65

The best practice is to always specify the values of instance attributes through the constructor.

***Setting Attribute Values***

The following constructor includes the name and age parameters, other than the self parameter.

class Person:

# name & age parameters passed in constructor

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

        self.name = name

        self.age = age

***Passing Instance Attribute Values in Constructor***

Now, you can specify the values while creating an instance, as shown below.

>>> p1 = Person('Mutemi', 65)

>>> p1.name

'Mutemi'

>>> p1.age

65

***Setting Default Values of Instance Attributes***

Also, you can set default values to instance attributes. By doing this, if the values are not provided when creating an object, the default values will be assigned.

Lets assign name=”mkuu” and age=101

class Person:

    def \_\_init\_\_(self, name="mkuu", age=101)

        self.name=name

        self.age=age

***Instance Attribute Default Value***

Now, you can create an object with default values, as shown below

>>> p1 = Person()

>>> p1.name

'Guest'

>>> p1.age

65

**Class Methods**

A python function in a class is called class method. Methods are defined using the def keyword.

Each method must have a self as the first parameter, which refers to calling the instance.

* Self is just a conventional name for the first argument of a method in the class.
* A method defined as mymethod(self, a, b) should be called as x.mymethod(a, b) for the object x of the class.

Example: here we have a method named displayInfo

class Person:

    def displayInfo(self): # class method

        print('Personal Information')

The above class method can be called as a normal function, as shown below.

>>> p1 = Person()

>>> p1.displayInfo()

'Personal Information'

Let's combine our knowledge so far for class constructors and methods to access instance attributes using self parameter.

class Person:

    def \_\_init\_\_(self, name, age): # class constructor

        self.name = name

        self.age = age

    def displayInfo(self): # class method

        print('Person Name: ', self.name,', Age: ', self.age)

*Calling a Method*

Let's call/Invoke the displayInfo method as shown below:

>>> p1 = Person('Mutemi', 65)

>>> p1.displayInfo()

Person Name: Mutemi , Age: 65

**Django**

**MVC architecture**

**Model –** represents the data- encapsulates data structure and data val, manipulation and storage

**View –** UI/presentation- rendering data to be displayed to the user

**Controller –** acts as the servers- intermediate between view and model

Scalable framework to use.

Building blocks for efficient programming

How to know which dsa to use:

Dsa helps in memory management – memory allocation.

Importance of algorithms

Algorithms – step by step procedures to solve problems

They determine efficiency of how a program can work.

Help with scalability – real time processing

Help with correctness.

Help with code reusability

Building Blocks for Efficient Programming

Topics

**Linked list implementation**

**Sorting Algorithms visualization**

**Graph Traversal and shortest path**

**Binary search tree (BST) operations**

**Linked list implementation**

A **Linked List** is, as the word implies, a list where the nodes are linked together. Each node contains data and a pointer. The way they are linked together is that each node points to where in the memory the next node is placed.

## The Importance of Sorting Algorithms in Python

[Sorting](https://docs.python.org/3/howto/sorting.html) is one of the most thoroughly studied algorithms in computer science. There are dozens of different sorting implementations and applications that you can use to make your code more efficient and effective.

You can use sorting to solve a wide range of problems:

* **Searching:** Searching for an item on a [list](https://realpython.com/python-lists-tuples/) works much faster if the list is sorted.
* **Selection:** Selecting items from a list based on their relationship to the rest of the items is easier with sorted data. For example, finding the kth-largest or smallest value, or finding the median value of the list, is much easier when the values are in ascending or descending order.
* **Duplicates:** Finding duplicate values on a list can be done very quickly when the list is sorted.
* **Distribution:** Analyzing the frequency distribution of items on a list is very fast if the list is sorted. For example, finding the element that appears most or least often is relatively straightforward with a sorted list.

From commercial applications to academic research and everywhere in between, there are countless ways you can use sorting to save yourself time and effort.

## Python’s Built-In Sorting Algorithm

The Python language, like many other high-level programming languages, offers the ability to sort data out of the box using sorted(). Here’s an example of sorting an integer array:

Python

>>> array = [8, 2, 6, 4, 5]

>>> sorted(array)

[2, 4, 5, 6, 8]

You can use sorted() to sort any list as long as the values inside are comparable.

## The Significance of Time Complexity

This tutorial covers two different ways to measure the **runtime** of sorting algorithms:

1. For a practical point of view, you’ll measure the runtime of the implementations using the timeit module.
2. For a more theoretical perspective, you’ll measure the **runtime complexity** of the algorithms using [**Big O notation**](https://en.wikipedia.org/wiki/Big_O_notation).

### **Timing Your Code**

When comparing two sorting algorithms in Python, it’s always informative to look at how long each one takes to run. The specific time each algorithm takes will be partly determined by your hardware, but you can still use the proportional time between executions to help you decide which implementation is more time efficient.

In this section, you’ll focus on a practical way to measure the actual time it takes to run to your sorting algorithms using the timeit module. For more information on the different ways you can time the execution of code in Python, check out [Python Timer Functions: Three Ways to Monitor Your Code](https://realpython.com/python-timer/).

Here’s a function you can use to time your algorithms:

Python

1from random import randint

2from timeit import repeat

3

4def run\_sorting\_algorithm(algorithm, array):

5 # Set up the context and prepare the call to the specified

6 # algorithm using the supplied array. Only import the

7 # algorithm function if it's not the built-in `sorted()`.

8 setup\_code = f"from \_\_main\_\_ import {algorithm}" \

9 if algorithm != "sorted" else ""

10

11 stmt = f"{algorithm}({array})"

12

13 # Execute the code ten different times and return the time

14 # in seconds that each execution took

15 times = repeat(setup=setup\_code, stmt=stmt, repeat=3, number=10)

16

17 # Finally, display the name of the algorithm and the

18 # minimum time it took to run

19 print(f"Algorithm: {algorithm}. Minimum execution time: {min(times)}")

In this example, run\_sorting\_algorithm() receives the name of the algorithm and the input array that needs to be sorted. Here’s a line-by-line explanation of how it works:

* **Line 8** imports the name of the algorithm using the magic of [Python’s f-strings](https://realpython.com/python-f-strings/). This is so that timeit.repeat() knows where to call the algorithm from. Note that this is only necessary for the custom implementations used in this tutorial. If the algorithm specified is the built-in sorted(), then nothing will be imported.
* **Line 11** prepares the call to the algorithm with the supplied array. This is the statement that will be executed and timed.
* **Line 15** calls timeit.repeat() with the setup code and the statement. This will call the specified sorting algorithm ten times, returning the number of seconds each one of these executions took.
* **Line 19** identifies the shortest time returned and prints it along with the name of the algorithm.

**Note:** A common misconception is that you should find the average time of each run of the algorithm instead of selecting the single shortest time. Time measurements are [noisy](https://en.wikipedia.org/wiki/Noisy_data) because the system runs other processes concurrently. The shortest time is always the least noisy, making it the best representation of the algorithm’s true runtime.

Here’s an example of how to use run\_sorting\_algorithm() to determine the time it takes to sort an array of ten thousand integer values using sorted():

Python

21ARRAY\_LENGTH = 10000

22

23if \_\_name\_\_ == "\_\_main\_\_":

24 # Generate an array of `ARRAY\_LENGTH` items consisting

25 # of random integer values between 0 and 999

26 array = [randint(0, 1000) for i in range(ARRAY\_LENGTH)]

27

28 # Call the function using the name of the sorting algorithm

29 # and the array you just created

30 run\_sorting\_algorithm(algorithm="sorted", array=array)

If you save the above code in a sorting.py file, then you can run it from the [terminal](https://realpython.com/terminal-commands/) and see its output:

Shell

$ python sorting.py

Algorithm: sorted. Minimum execution time: 0.010945824000000007

Remember that the time in seconds of every experiment depends in part on the hardware you use, so you’ll likely see slightly different results when running the code.

**Note:** You can learn more about the timeit module in the [official Python documentation](https://docs.python.org/2/library/timeit.html).

## The Bubble Sort Algorithm in Python

**Bubble Sort** is one of the most straightforward sorting algorithms. Its name comes from the way the algorithm works: With every new pass, the largest element in the list “bubbles up” toward its correct position.

Bubble sort consists of making multiple passes through a list, comparing elements one by one, and swapping adjacent items that are out of order.

### **Implementing Bubble Sort in Python**

Here’s an implementation of a bubble sort algorithm in Python:

Python

1def bubble\_sort(array):

2 n = len(array)

3

4 for i in range(n):

5 # Create a flag that will allow the function to

6 # terminate early if there's nothing left to sort

7 already\_sorted = True

8

9 # Start looking at each item of the list one by one,

10 # comparing it with its adjacent value. With each

11 # iteration, the portion of the array that you look at

12 # shrinks because the remaining items have already been

13 # sorted.

14 for j in range(n - i - 1):

15 if array[j] > array[j + 1]:

16 # If the item you're looking at is greater than its

17 # adjacent value, then swap them

18 array[j], array[j + 1] = array[j + 1], array[j]

19

20 # Since you had to swap two elements,

21 # set the `already\_sorted` flag to `False` so the

22 # algorithm doesn't finish prematurely

23 already\_sorted = False

24

25 # If there were no swaps during the last iteration,

26 # the array is already sorted, and you can terminate

27 if already\_sorted:

28 break

29

30 return array

Since this implementation sorts the array in ascending order, each step “bubbles” the largest element to the end of the array. This means that each iteration takes fewer steps than the previous iteration because a continuously larger portion of the array is sorted.

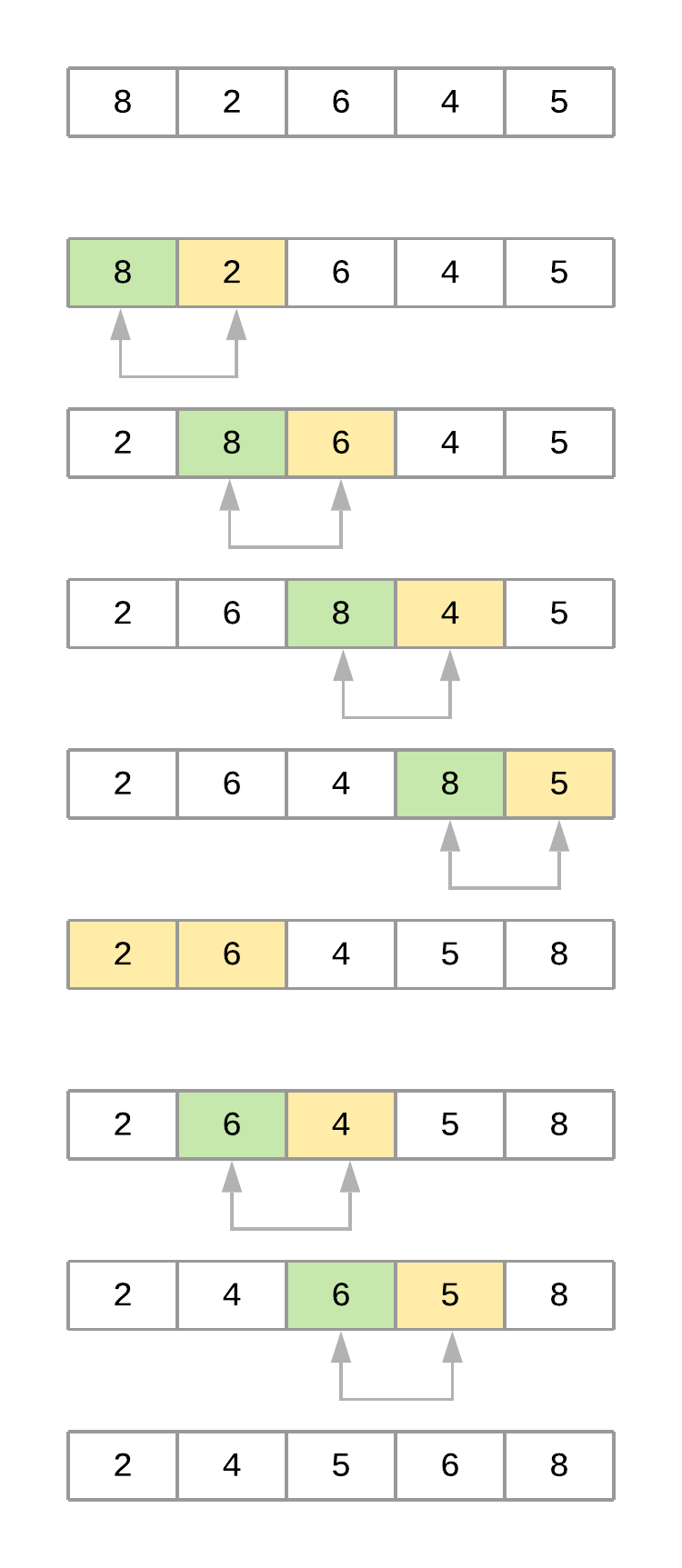
The loops in **lines 4 and 10** determine the way the algorithm runs through the list. Notice how j initially goes from the first element in the list to the element immediately before the last. During the second iteration, j runs until two items from the last, then three items from the last, and so on. At the end of each iteration, the end portion of the list will be sorted.

As the loops progress, **line 15** compares each element with its adjacent value, and **line 18** swaps them if they are in the incorrect order. This ensures a sorted list at the end of the function.

**Note:** The already\_sorted flag in **lines 13, 23, and 27** of the code above is an optimization to the algorithm, and it’s not required in a fully functional bubble sort implementation. However, it allows the function to skip unnecessary steps if the list ends up wholly sorted before the loops have finished.

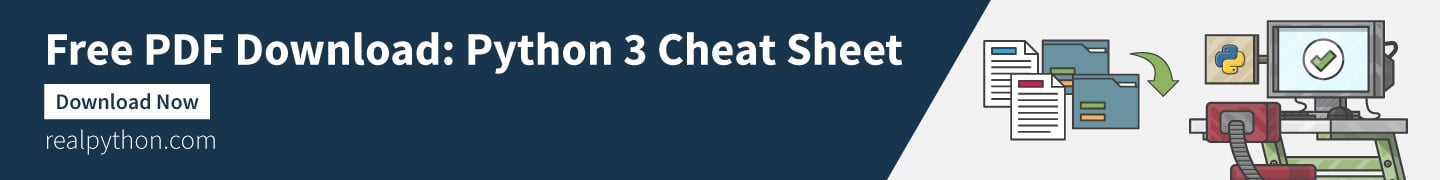
As an exercise, you can remove the use of this flag and compare the runtimes of both implementations.

To properly analyze how the algorithm works, consider a list with values [8, 2, 6, 4, 5]. Assume you’re using bubble\_sort() from above. Here’s a figure illustrating what the array looks like at each iteration of the algorithm:

[](https://files.realpython.com/media/python-sorting-algorithms-bubble-sort.216ab9a52018.png)The Bubble Sort Process

Now take a step-by-step look at what’s happening with the array as the algorithm progresses:

1. The code starts by comparing the first element, 8, with its adjacent element, 2. Since 8 > 2, the values are swapped, resulting in the following order: [2, 8, 6, 4, 5].
2. The algorithm then compares the second element, 8, with its adjacent element, 6. Since 8 > 6, the values are swapped, resulting in the following order: [2, 6, 8, 4, 5].
3. Next, the algorithm compares the third element, 8, with its adjacent element, 4. Since 8 > 4, it swaps the values as well, resulting in the following order: [2, 6, 4, 8, 5].
4. Finally, the algorithm compares the fourth element, 8, with its adjacent element, 5, and swaps them as well, resulting in [2, 6, 4, 5, 8]. At this point, the algorithm completed the first pass through the list (i = 0). Notice how the value 8 bubbled up from its initial location to its correct position at the end of the list.
5. The second pass (i = 1) takes into account that the last element of the list is already positioned and focuses on the remaining four elements, [2, 6, 4, 5]. At the end of this pass, the value 6 finds its correct position. The third pass through the list positions the value 5, and so on until the list is sorted.

[](https://srv.realpython.net/click/53616533979/?c=62098619220&p=58946116052&r=58780)

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

### **Measuring Bubble Sort’s Big O Runtime Complexity**

Your implementation of bubble sort consists of two nested [for loops](https://realpython.com/python-for-loop/) in which the algorithm performs n - 1 comparisons, then n - 2 comparisons, and so on until the final comparison is done. This comes at a total of (n - 1) + (n - 2) + (n - 3) + … + 2 + 1 = n(n-1)/2 comparisons, which can also be written as ½n2 - ½n.

You learned earlier that Big O focuses on how the runtime grows in comparison to the size of the input. That means that, in order to turn the above equation into the Big O complexity of the algorithm, you need to remove the constants because they don’t change with the input size.

Doing so simplifies the notation to n2 - n. Since n2 grows much faster than n, this last term can be dropped as well, leaving bubble sort with an average- and worst-case complexity of **O(n2)**.

In cases where the algorithm receives an array that’s already sorted—and assuming the implementation includes the already\_sorted flag optimization explained before—the runtime complexity will come down to a much better O(n) because the algorithm will not need to visit any element more than once.

O(n), then, is the best-case runtime complexity of bubble sort. But keep in mind that best cases are an exception, and you should focus on the average case when comparing different algorithms.

### **Timing Your Bubble Sort Implementation**

Using your run\_sorting\_algorithm() from earlier in this tutorial, here’s the time it takes for bubble sort to process an array with ten thousand items. **Line 8** replaces the name of the algorithm and everything else stays the same:

Python

1if \_\_name\_\_ == "\_\_main\_\_":

2 # Generate an array of `ARRAY\_LENGTH` items consisting

3 # of random integer values between 0 and 999

4 array = [randint(0, 1000) for i in range(ARRAY\_LENGTH)]

5

6 # Call the function using the name of the sorting algorithm

7 # and the array you just created

8 run\_sorting\_algorithm(algorithm="bubble\_sort", array=array)

You can now run the script to get the execution time of bubble\_sort:

Shell

$ python sorting.py

Algorithm: bubble\_sort. Minimum execution time: 73.21720498399998

It took 73 seconds to sort the array with ten thousand elements. This represents the fastest execution out of the ten repetitions that run\_sorting\_algorithm() runs. Executing this script multiple times will produce similar results.

**Note:** A single execution of bubble sort took 73 seconds, but the algorithm ran ten times using timeit.repeat(). This means that you should expect your code to take around 73 \* 10 = 730 seconds to run, assuming you have similar hardware characteristics. Slower machines may take much longer to finish.

### **Analyzing the Strengths and Weaknesses of Bubble Sort**

The main advantage of the bubble sort algorithm is its **simplicity**. It is straightforward to both implement and understand. This is probably the main reason why most computer science courses introduce the topic of sorting using bubble sort.

As you saw before, the disadvantage of bubble sort is that it is **slow**, with a runtime complexity of O(n2). Unfortunately, this rules it out as a practical candidate for sorting large arrays.

## The Insertion Sort Algorithm in Python

Like bubble sort, the **insertion sort** algorithm is straightforward to implement and understand. But unlike bubble sort, it builds the sorted list one element at a time by comparing each item with the rest of the list and inserting it into its correct position. This “insertion” procedure gives the algorithm its name.

An excellent analogy to explain insertion sort is the way you would sort a deck of cards. Imagine that you’re holding a group of cards in your hands, and you want to arrange them in order. You’d start by comparing a single card step by step with the rest of the cards until you find its correct position. At that point, you’d insert the card in the correct location and start over with a new card, repeating until all the cards in your hand were sorted.

### **Implementing Insertion Sort in Python**

The insertion sort algorithm works exactly like the example with the deck of cards. Here’s the implementation in Python:

Python

1def insertion\_sort(array):

2 # Loop from the second element of the array until

3 # the last element

4 for i in range(1, len(array)):

5 # This is the element we want to position in its

6 # correct place

7 key\_item = array[i]

8

9 # Initialize the variable that will be used to

10 # find the correct position of the element referenced

11 # by `key\_item`

12 j = i - 1

13

14 # Run through the list of items (the left

15 # portion of the array) and find the correct position

16 # of the element referenced by `key\_item`. Do this only

17 # if `key\_item` is smaller than its adjacent values.

18 while j >= 0 and array[j] > key\_item:

19 # Shift the value one position to the left

20 # and reposition j to point to the next element

21 # (from right to left)

22 array[j + 1] = array[j]

23 j -= 1

24

25 # When you finish shifting the elements, you can position

26 # `key\_item` in its correct location

27 array[j + 1] = key\_item

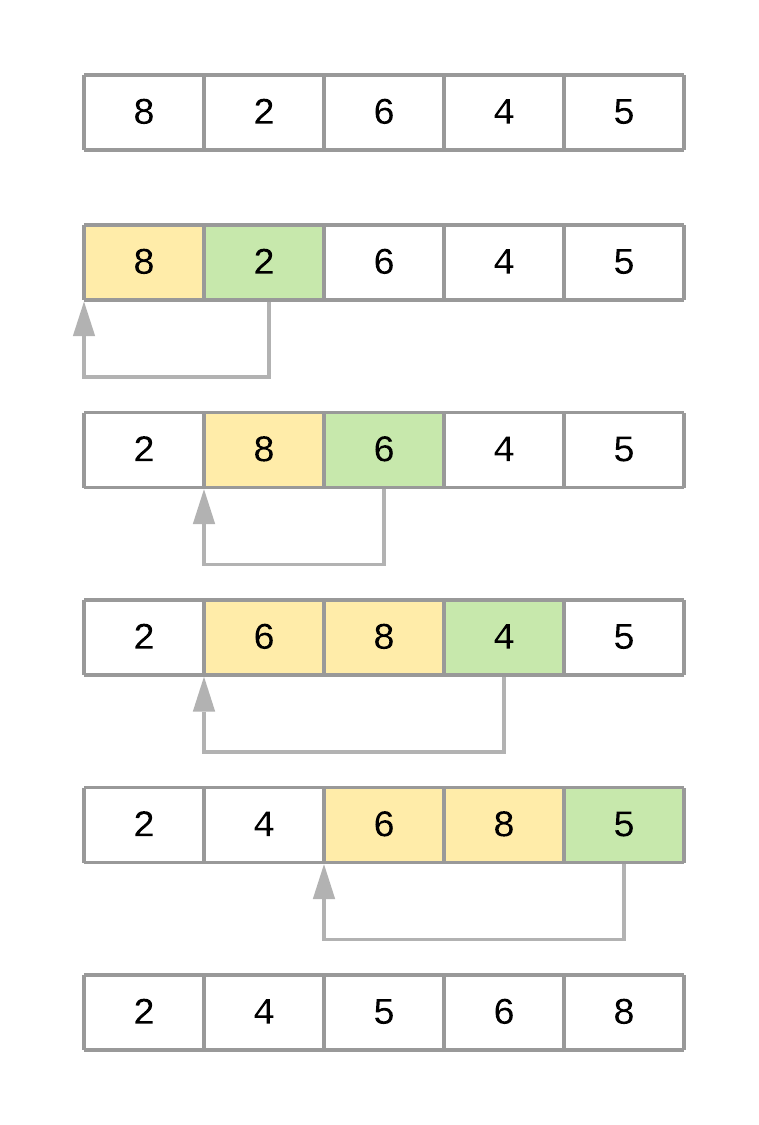
28

29 return array

Unlike bubble sort, this implementation of insertion sort constructs the sorted list by pushing smaller items to the left. Let’s break down insertion\_sort() line by line:

* **Line 4** sets up the loop that determines the key\_item that the function will position during each iteration. Notice that the loop starts with the second item on the list and goes all the way to the last item.
* **Line 7** initializes key\_item with the item that the function is trying to place.
* **Line 12** initializes a variable that will consecutively point to each element to the left of key item. These are the elements that will be consecutively compared with key\_item.
* **Line 18** compares key\_item with each value to its left using a while loop, shifting the elements to make room to place key\_item.
* **Line 27** positions key\_item in its correct place after the algorithm shifts all the larger values to the right.

Here’s a figure illustrating the different iterations of the algorithm when sorting the array [8, 2, 6, 4, 5]:

[](https://files.realpython.com/media/python-sorting-algorithms-insertion-sort.a102f819b3d7.png)The Insertion Sort Process

Now here’s a summary of the steps of the algorithm when sorting the array:

1. The algorithm starts with key\_item = 2 and goes through the subarray to its left to find the correct position for it. In this case, the subarray is [8].
2. Since 2 < 8, the algorithm shifts element 8 one position to its right. The resultant array at this point is [8, 8, 6, 4, 5].
3. Since there are no more elements in the subarray, the key\_item is now placed in its new position, and the final array is [2, 8, 6, 4, 5].
4. The second pass starts with key\_item = 6 and goes through the subarray located to its left, in this case [2, 8].
5. Since 6 < 8, the algorithm shifts 8 to its right. The resultant array at this point is [2, 8, 8, 4, 5].
6. Since 6 > 2, the algorithm doesn’t need to keep going through the subarray, so it positions key\_item and finishes the second pass. At this time, the resultant array is [2, 6, 8, 4, 5].
7. The third pass through the list puts the element 4 in its correct position, and the fourth pass places element 5 in the correct spot, leaving the array sorted.

[](https://srv.realpython.net/click/14590204085/?c=62098619220&p=58946116052&r=72805)

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

### **Measuring Insertion Sort’s Big O Runtime Complexity**

Similar to your bubble sort implementation, the insertion sort algorithm has a couple of nested loops that go over the list. The inner loop is pretty efficient because it only goes through the list until it finds the correct position of an element. That said, the algorithm still has an **O(n2)** runtime complexity on the average case.

The worst case happens when the supplied array is sorted in reverse order. In this case, the inner loop has to execute every comparison to put every element in its correct position. This still gives you an O(n2) runtime complexity.

The best case happens when the supplied array is already sorted. Here, the inner loop is never executed, resulting in an O(n) runtime complexity, just like the best case of bubble sort.

Although bubble sort and insertion sort have the same Big O runtime complexity, in practice, insertion sort is considerably more efficient than bubble sort. If you look at the implementation of both algorithms, then you can see how insertion sort has to make fewer comparisons to sort the list.

### **Timing Your Insertion Sort Implementation**

To prove the assertion that insertion sort is more efficient than bubble sort, you can time the insertion sort algorithm and compare it with the results of bubble sort. To do this, you just need to replace the call to run\_sorting\_algorithm() with the name of your insertion sort implementation:

Python

1if \_\_name\_\_ == "\_\_main\_\_":

2 # Generate an array of `ARRAY\_LENGTH` items consisting

3 # of random integer values between 0 and 999

4 array = [randint(0, 1000) for i in range(ARRAY\_LENGTH)]

5

6 # Call the function using the name of the sorting algorithm

7 # and the array we just created

8 run\_sorting\_algorithm(algorithm="insertion\_sort", array=array)

You can execute the script as before:

Shell

$ python sorting.py

Algorithm: insertion\_sort. Minimum execution time: 56.71029764299999

Notice how the insertion sort implementation took around 17 fewer seconds than the bubble sort implementation to sort the same array. Even though they’re both O(n2) algorithms, insertion sort is more efficient.

### **Analyzing the Strengths and Weaknesses of Insertion Sort**

Just like bubble sort, the insertion sort algorithm is very uncomplicated to implement. Even though insertion sort is an O(n2) algorithm, it’s also much more efficient in practice than other quadratic implementations such as bubble sort.

There are more powerful algorithms, including merge sort and Quicksort, but these implementations are recursive and usually fail to beat insertion sort when working on small lists. Some Quicksort implementations even use insertion sort internally if the list is small enough to provide a faster overall implementation. [**Timsort**](https://en.wikipedia.org/wiki/Timsort) also uses insertion sort internally to sort small portions of the input array.

That said, insertion sort is not practical for large arrays, opening the door to algorithms that can scale in more efficient ways.

## The Merge Sort Algorithm in Python

**Merge sort** is a very efficient sorting algorithm. It’s based on the [divide-and-conquer](https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm) approach, a powerful algorithmic technique used to solve complex problems.

To properly understand divide and conquer, you should first understand the concept of **recursion**. Recursion involves breaking a problem down into smaller subproblems until they’re small enough to manage. In programming, recursion is usually expressed by a function calling itself.

**Note**: This tutorial doesn’t explore recursion in depth. To better understand how recursion works and see it in action using Python, check out [Thinking Recursively in Python](https://realpython.com/python-thinking-recursively/) and [Recursion in Python: An Introduction](https://realpython.com/python-recursion/).

Divide-and-conquer algorithms typically follow the same structure:

1. The original input is broken into several parts, each one representing a subproblem that’s similar to the original but simpler.
2. Each subproblem is solved recursively.
3. The solutions to all the subproblems are combined into a single overall solution.

In the case of merge sort, the divide-and-conquer approach divides the set of input values into two equal-sized parts, sorts each half recursively, and finally merges these two sorted parts into a single sorted list.

[](https://srv.realpython.net/click/30513864559/?c=62098619220&p=58946116052&r=72335)

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

### **Implementing Merge Sort in Python**

The implementation of the merge sort algorithm needs two different pieces:

1. A function that recursively splits the input in half
2. A function that merges both halves, producing a sorted array

Here’s the code to merge two different arrays:

Python

1def merge(left, right):

2 # If the first array is empty, then nothing needs

3 # to be merged, and you can return the second array as the result

4 if len(left) == 0:

5 return right

6

7 # If the second array is empty, then nothing needs

8 # to be merged, and you can return the first array as the result

9 if len(right) == 0:

10 return left

11

12 result = []

13 index\_left = index\_right = 0

14

15 # Now go through both arrays until all the elements

16 # make it into the resultant array

17 while len(result) < len(left) + len(right):

18 # The elements need to be sorted to add them to the

19 # resultant array, so you need to decide whether to get

20 # the next element from the first or the second array

21 if left[index\_left] <= right[index\_right]:

22 result.append(left[index\_left])

23 index\_left += 1

24 else:

25 result.append(right[index\_right])

26 index\_right += 1

27

28 # If you reach the end of either array, then you can

29 # add the remaining elements from the other array to

30 # the result and break the loop

31 if index\_right == len(right):

32 result += left[index\_left:]

33 break

34

35 if index\_left == len(left):

36 result += right[index\_right:]

37 break

38

39 return result

merge() receives two different sorted arrays that need to be merged together. The process to accomplish this is straightforward:

* **Lines 4 and 9** check whether either of the arrays is empty. If one of them is, then there’s nothing to merge, so the function returns the other array.
* **Line 17** starts a [while loop](https://realpython.com/courses/mastering-while-loops/) that ends whenever the result contains all the elements from both of the supplied arrays. The goal is to look into both arrays and combine their items to produce a sorted list.
* **Line 21** compares the elements at the head of both arrays, selects the smaller value, and [appends](https://realpython.com/python-append/) it to the end of the resultant array.
* **Lines 31 and 35** append any remaining items to the result if all the elements from either of the arrays were already used.

With the above function in place, the only missing piece is a function that recursively splits the input array in half and uses merge() to produce the final result:

Python

41def merge\_sort(array):

42 # If the input array contains fewer than two elements,

43 # then return it as the result of the function

44 if len(array) < 2:

45 return array

46

47 midpoint = len(array) // 2

48

49 # Sort the array by recursively splitting the input

50 # into two equal halves, sorting each half and merging them

51 # together into the final result

52 return merge(

53 left=merge\_sort(array[:midpoint]),

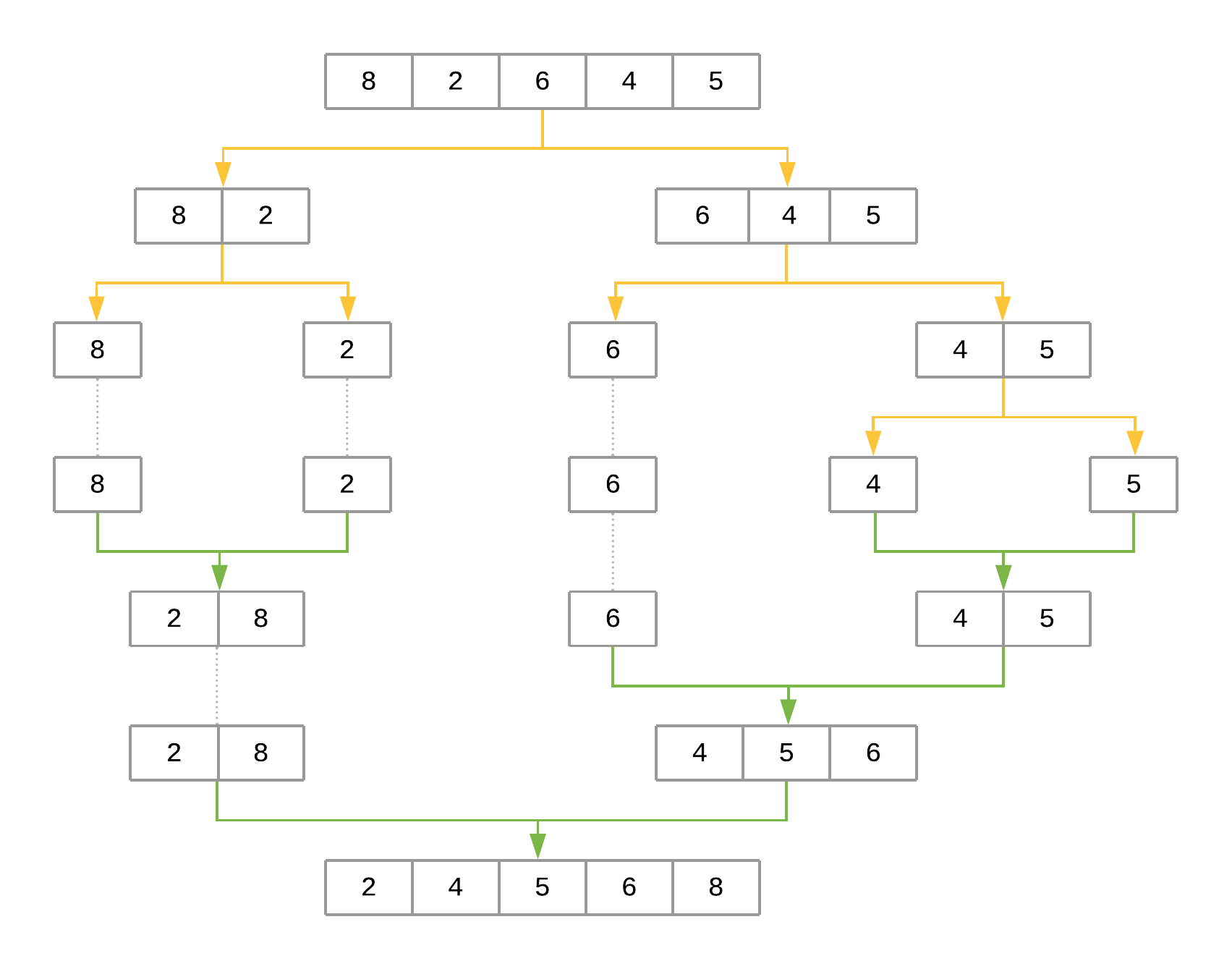
54 right=merge\_sort(array[midpoint:]))

Here’s a quick summary of the code:

* **Line 44** acts as the stopping condition for the recursion. If the input array contains fewer than two elements, then the function returns the array. Notice that this condition could be triggered by receiving either a single item or an empty array. In both cases, there’s nothing left to sort, so the function should return.
* **Line 47** computes the middle point of the array.
* **Line 52** calls merge(), passing both sorted halves as the arrays.

Notice how this function calls itself **recursively**, halving the array each time. Each iteration deals with an ever-shrinking array until fewer than two elements remain, meaning there’s nothing left to sort. At this point, merge() takes over, merging the two halves and producing a sorted list.

Take a look at a representation of the steps that merge sort will take to sort the array [8, 2, 6, 4, 5]:

[](https://files.realpython.com/media/python-sorting-algorithms-merge-sort.d6b5c7dec9ef.png)The Merge Sort Process

The figure uses yellow arrows to represent halving the array at each recursion level. The green arrows represent merging each subarray back together. The steps can be summarized as follows:

1. The first call to merge\_sort() with [8, 2, 6, 4, 5] defines midpoint as 2. The midpoint is used to halve the input array into array[:2] and array[2:], producing [8, 2] and [6, 4, 5], respectively. merge\_sort() is then recursively called for each half to sort them separately.
2. The call to merge\_sort() with [8, 2] produces [8] and [2]. The process repeats for each of these halves.
3. The call to merge\_sort() with [8] returns [8] since that’s the only element. The same happens with the call to merge\_sort() with [2].
4. At this point, the function starts merging the subarrays back together using merge(), starting with [8] and [2] as input arrays, producing [2, 8] as the result.
5. On the other side, [6, 4, 5] is recursively broken down and merged using the same procedure, producing [4, 5, 6] as the result.
6. In the final step, [2, 8] and [4, 5, 6] are merged back together with merge(), producing the final result: [2, 4, 5, 6, 8].

### **Measuring Merge Sort’s Big O Complexity**

To analyze the complexity of merge sort, you can look at its two steps separately:

1. merge() has a linear runtime. It receives two arrays whose combined length is at most n (the length of the original input array), and it combines both arrays by looking at each element at most once. This leads to a runtime complexity of O(n).
2. The second step splits the input array recursively and calls merge() for each half. Since the array is halved until a single element remains, the total number of halving operations performed by this function is log2n. Since merge() is called for each half, we get a total runtime of **O(n log2n)**.

Interestingly, O(n log2n) is the best possible worst-case runtime that can be achieved by a sorting algorithm.

[](https://srv.realpython.net/click/33172000876/?c=43772654581&p=58946116052&r=84682)

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

### **Timing Your Merge Sort Implementation**

To compare the speed of merge sort with the previous two implementations, you can use the same mechanism as before and replace the name of the algorithm in **line 8**:

Python

1if \_\_name\_\_ == "\_\_main\_\_":

2 # Generate an array of `ARRAY\_LENGTH` items consisting

3 # of random integer values between 0 and 999

4 array = [randint(0, 1000) for i in range(ARRAY\_LENGTH)]

5

6 # Call the function using the name of the sorting algorithm

7 # and the array you just created

8 run\_sorting\_algorithm(algorithm="merge\_sort", array=array)

You can execute the script to get the execution time of merge\_sort:

Shell

$ python sorting.py

Algorithm: merge\_sort. Minimum execution time: 0.6195857160000173

Compared to bubble sort and insertion sort, the merge sort implementation is extremely fast, sorting the ten-thousand-element array in less than a second!

### **Analyzing the Strengths and Weaknesses of Merge Sort**

Thanks to its runtime complexity of O(n log2n), merge sort is a very efficient algorithm that scales well as the size of the input array grows. It’s also straightforward to [**parallelize**](https://en.wikipedia.org/wiki/Parallel_algorithm) because it breaks the input array into chunks that can be distributed and processed in parallel if necessary.

That said, for small lists, the time cost of the recursion allows algorithms such as bubble sort and insertion sort to be faster. For example, running an experiment with a list of ten elements results in the following times:

Shell

Algorithm: bubble\_sort. Minimum execution time: 0.000018774999999998654

Algorithm: insertion\_sort. Minimum execution time: 0.000029786000000000395

Algorithm: merge\_sort. Minimum execution time: 0.00016983000000000276

Both bubble sort and insertion sort beat merge sort when sorting a ten-element list.

Another drawback of merge sort is that it creates copies of the array when calling itself recursively. It also creates a new list inside merge() to sort and return both input halves. This makes merge sort use much more memory than bubble sort and insertion sort, which are both able to sort the list in place.

Due to this limitation, you may not want to use merge sort to sort large lists in memory-constrained hardware.

## The Quicksort Algorithm in Python

Just like merge sort, the **Quicksort** algorithm applies the divide-and-conquer principle to divide the input array into two lists, the first with small items and the second with large items. The algorithm then sorts both lists recursively until the resultant list is completely sorted.

Dividing the input list is referred to as **partitioning** the list. Quicksort first selects a pivot element and partitions the list around the pivot, putting every smaller element into a low array and every larger element into a high array.

Putting every element from the low list to the left of the pivot and every element from the high list to the right positions the pivot precisely where it needs to be in the final sorted list. This means that the function can now recursively apply the same procedure to low and then high until the entire list is sorted.

### **Implementing Quicksort in Python**

Here’s a fairly compact implementation of Quicksort:

Python

1from random import randint

2

3def quicksort(array):

4 # If the input array contains fewer than two elements,

5 # then return it as the result of the function

6 if len(array) < 2:

7 return array

8

9 low, same, high = [], [], []

10

11 # Select your `pivot` element randomly

12 pivot = array[randint(0, len(array) - 1)]

13

14 for item in array:

15 # Elements that are smaller than the `pivot` go to

16 # the `low` list. Elements that are larger than

17 # `pivot` go to the `high` list. Elements that are

18 # equal to `pivot` go to the `same` list.

19 if item < pivot:

20 low.append(item)

21 elif item == pivot:

22 same.append(item)

23 elif item > pivot:

24 high.append(item)

25

26 # The final result combines the sorted `low` list

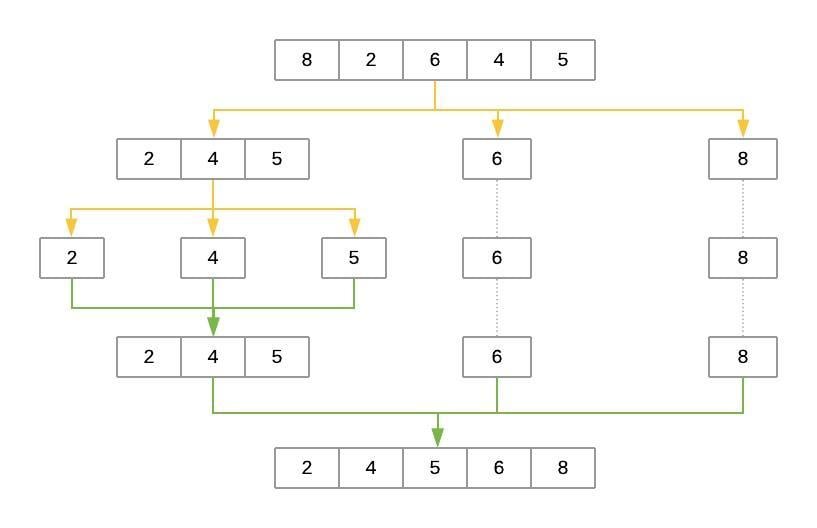
27 # with the `same` list and the sorted `high` list

28 return quicksort(low) + same + quicksort(high)

Here’s a summary of the code:

* **Line 6** stops the recursive function if the array contains fewer than two elements.
* **Line 12** selects the pivot element randomly from the list and proceeds to partition the list.
* **Lines 19 and 20** put every element that’s smaller than pivot into the list called low.
* **Lines 21 and 22** put every element that’s equal to pivot into the list called same.
* **Lines 23 and 24** put every element that’s larger than pivot into the list called high.
* **Line 28** recursively sorts the low and high lists and combines them along with the contents of the same list.

Here’s an illustration of the steps that Quicksort takes to sort the array [8, 2, 6, 4, 5]:

[](https://files.realpython.com/media/Python_Sorting_Algorithms_-_Quick_Sort.ee6dbe24f0d3.jpeg)The Quicksort Process

The yellow lines represent the partitioning of the array into three lists: low, same, and high. The green lines represent sorting and putting these lists back together. Here’s a brief explanation of the steps:

1. The pivot element is selected randomly. In this case, pivot is 6.
2. The first pass partitions the input array so that low contains [2, 4, 5], same contains [6], and high contains [8].
3. quicksort() is then called recursively with low as its input. This selects a random pivot and breaks the array into [2] as low, [4] as same, and [5] as high.
4. The process continues, but at this point, both low and high have fewer than two items each. This ends the recursion, and the function puts the array back together. Adding the sorted low and high to either side of the same list produces [2, 4, 5].
5. On the other side, the high list containing [8] has fewer than two elements, so the algorithm returns the sorted low array, which is now [2, 4, 5]. Merging it with same ([6]) and high ([8]) produces the final sorted list.

[](https://srv.realpython.net/click/36075076065/?c=30337084727&p=58946116052&r=96487)

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

### **Selecting the pivot Element**

Why does the implementation above select the pivot element randomly? Wouldn’t it be the same to consistently select the first or last element of the input list?

Because of how the Quicksort algorithm works, the number of recursion levels depends on where pivot ends up in each partition. In the best-case scenario, the algorithm consistently picks the **median** element as the pivot. That would make each generated subproblem exactly half the size of the previous problem, leading to at most log2n levels.

On the other hand, if the algorithm consistently picks either the smallest or largest element of the array as the pivot, then the generated partitions will be as unequal as possible, leading to n-1 recursion levels. That would be the worst-case scenario for Quicksort.

As you can see, Quicksort’s efficiency often depends on the pivot selection. If the input array is unsorted, then using the first or last element as the pivot will work the same as a random element. But if the input array is sorted or almost sorted, using the first or last element as the pivot could lead to a worst-case scenario. Selecting the pivot at random makes it more likely Quicksort will select a value closer to the median and finish faster.

Another option for selecting the pivot is to [find the median value of the array](https://brilliant.org/wiki/median-finding-algorithm/) and force the algorithm to use it as the pivot. This can be done in O(n) time. Although the process is little bit more involved, using the median value as the pivot for Quicksort guarantees you will have the best-case Big O scenario.

### **Measuring Quicksort’s Big O Complexity**

With Quicksort, the input list is partitioned in linear time, O(n), and this process repeats recursively an average of log2n times. This leads to a final complexity of **O(n log2n)**.

That said, remember the discussion about how the selection of the pivot affects the runtime of the algorithm. The O(n) best-case scenario happens when the selected pivot is close to the median of the array, and an O(n2) scenario happens when the pivot is the smallest or largest value of the array.

Theoretically, if the algorithm focuses first on finding the median value and then uses it as the pivot element, then the worst-case complexity will come down to O(n log2n). The median of an array can be found in linear time, and using it as the pivot guarantees the Quicksort portion of the code will perform in O(n log2n).

By using the median value as the pivot, you end up with a final runtime of O(n) + O(n log2n). You can simplify this down to O(n log2n) because the logarithmic portion grows much faster than the linear portion.

**Note**: Although achieving O(n log2n) is possible in Quicksort’s worst-case scenario, this approach is seldom used in practice. Lists have to be quite large for the implementation to be faster than a simple randomized selection of the pivot.

Randomly selecting the pivot makes the worst case very unlikely. That makes random pivot selection good enough for most implementations of the algorithm.

### **Timing Your Quicksort Implementation**

By now, you’re familiar with the process for timing the runtime of the algorithm. Just change the name of the algorithm in **line 8**:

Python

1if \_\_name\_\_ == "\_\_main\_\_":

2 # Generate an array of `ARRAY\_LENGTH` items consisting

3 # of random integer values between 0 and 999

4 array = [randint(0, 1000) for i in range(ARRAY\_LENGTH)]

5

6 # Call the function using the name of the sorting algorithm

7 # and the array you just created

8 run\_sorting\_algorithm(algorithm="quicksort", array=array)

You can execute the script as you have before:

Shell

$ python sorting.py

Algorithm: quicksort. Minimum execution time: 0.11675417600002902

Not only does Quicksort finish in less than one second, but it’s also much faster than merge sort (0.11 seconds versus 0.61 seconds). Increasing the number of elements specified by ARRAY\_LENGTH from 10,000 to 1,000,000 and running the script again ends up with merge sort finishing in 97 seconds, whereas Quicksort sorts the list in a mere 10 seconds.

### **Analyzing the Strengths and Weaknesses of Quicksort**

True to its name, Quicksort is very fast. Although its worst-case scenario is theoretically O(n2), in practice, a good implementation of Quicksort beats most other sorting implementations. Also, just like merge sort, Quicksort is straightforward to **parallelize**.

One of Quicksort’s main disadvantages is the lack of a guarantee that it will achieve the average runtime complexity. Although worst-case scenarios are rare, certain applications can’t afford to risk poor performance, so they opt for algorithms that stay within O(n log2n) regardless of the input.

Just like merge sort, Quicksort also trades off memory space for speed. This may become a limitation for sorting larger lists.

A quick experiment sorting a list of ten elements leads to the following results:

Shell

Algorithm: bubble\_sort. Minimum execution time: 0.0000909000000000014

Algorithm: insertion\_sort. Minimum execution time: 0.00006681900000000268

Algorithm: quicksort. Minimum execution time: 0.0001319930000000004

The results show that Quicksort also pays the price of recursion when the list is sufficiently small, taking longer to complete than both insertion sort and bubble sort.

[](https://srv.realpython.net/click/54948072182/?c=30337084727&p=58946116052&r=18055)

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

## The Timsort Algorithm in Python

The **Timsort** algorithm is considered a **hybrid** sorting algorithm because it employs a best-of-both-worlds combination of insertion sort and merge sort. Timsort is near and dear to the Python community because it was created by Tim Peters in 2002 to be used as the [standard sorting algorithm of the Python language](https://en.wikipedia.org/wiki/Timsort).

The main characteristic of Timsort is that it takes advantage of already-sorted elements that exist in most real-world datasets. These are called **natural runs**. The algorithm then iterates over the list, collecting the elements into runs and merging them into a single sorted list.

### **Implementing Timsort in Python**

In this section, you’ll create a barebones Python implementation that illustrates all the pieces of the Timsort algorithm. If you’re interested, you can also check out the [original C implementation of Timsort](https://github.com/python/cpython/blob/master/Objects/listobject.c).

The first step in implementing Timsort is modifying the implementation of insertion\_sort() from before:

Python

1def insertion\_sort(array, left=0, right=None):

2 if right is None:

3 right = len(array) - 1

4

5 # Loop from the element indicated by

6 # `left` until the element indicated by `right`

7 for i in range(left + 1, right + 1):

8 # This is the element we want to position in its

9 # correct place

10 key\_item = array[i]

11

12 # Initialize the variable that will be used to

13 # find the correct position of the element referenced

14 # by `key\_item`

15 j = i - 1

16

17 # Run through the list of items (the left

18 # portion of the array) and find the correct position

19 # of the element referenced by `key\_item`. Do this only

20 # if the `key\_item` is smaller than its adjacent values.

21 while j >= left and array[j] > key\_item:

22 # Shift the value one position to the left

23 # and reposition `j` to point to the next element

24 # (from right to left)

25 array[j + 1] = array[j]

26 j -= 1

27

28 # When you finish shifting the elements, position

29 # the `key\_item` in its correct location

30 array[j + 1] = key\_item

31

32 return array

This modified implementation adds a couple of parameters, left and right, that indicate which portion of the array should be sorted. This allows the Timsort algorithm to sort a portion of the array in place. Modifying the function instead of creating a new one means that it can be reused for both insertion sort and Timsort.

Now take a look at the implementation of Timsort:

Python

1def timsort(array):

2 min\_run = 32

3 n = len(array)

4

5 # Start by slicing and sorting small portions of the

6 # input array. The size of these slices is defined by

7 # your `min\_run` size.

8 for i in range(0, n, min\_run):

9 insertion\_sort(array, i, min((i + min\_run - 1), n - 1))

10

11 # Now you can start merging the sorted slices.

12 # Start from `min\_run`, doubling the size on

13 # each iteration until you surpass the length of

14 # the array.

15 size = min\_run

16 while size < n:

17 # Determine the arrays that will

18 # be merged together

19 for start in range(0, n, size \* 2):

20 # Compute the `midpoint` (where the first array ends

21 # and the second starts) and the `endpoint` (where

22 # the second array ends)

23 midpoint = start + size - 1

24 end = min((start + size \* 2 - 1), (n-1))

25

26 # Merge the two subarrays.

27 # The `left` array should go from `start` to

28 # `midpoint + 1`, while the `right` array should

29 # go from `midpoint + 1` to `end + 1`.

30 merged\_array = merge(

31 left=array[start:midpoint + 1],

32 right=array[midpoint + 1:end + 1])

33

34 # Finally, put the merged array back into

35 # your array

36 array[start:start + len(merged\_array)] = merged\_array

37

38 # Each iteration should double the size of your arrays

39 size \*= 2

40

41 return array

Although the implementation is a bit more complex than the previous algorithms, we can summarize it quickly in the following way:

* **Lines 8 and 9** create small slices, or runs, of the array and sort them using insertion sort. You learned previously that insertion sort is speedy on small lists, and Timsort takes advantage of this. Timsort uses the newly introduced left and right parameters in insertion\_sort() to sort the list in place without having to create new arrays like merge sort and Quicksort do.
* **Line 16** merges these smaller runs, with each run being of size 32 initially. With each iteration, the size of the runs is doubled, and the algorithm continues merging these larger runs until a single sorted run remains.

Notice how, unlike merge sort, Timsort merges subarrays that were previously sorted. Doing so decreases the total number of comparisons required to produce a sorted list. This advantage over merge sort will become apparent when running experiments using different arrays.

Finally, **line 2** defines min\_run = 32. There are two reasons for using 32 as the value here:

1. Sorting small arrays using insertion sort is very fast, and min\_run has a small value to take advantage of this characteristic. Initializing min\_run with a value that’s too large will defeat the purpose of using insertion sort and will make the algorithm slower.
2. Merging two balanced lists is much more efficient than merging lists of disproportionate size. Picking a min\_run value that’s a power of two ensures better performance when merging all the different runs that the algorithm creates.

Combining both conditions above offers several options for min\_run. The implementation in this tutorial uses min\_run = 32 as one of the possibilities.

**Note:** In practice, Timsort does something a little more complicated to compute min\_run. It picks a value between 32 and 64 inclusive, such that the length of the list divided by min\_run is exactly a power of 2. If that’s not possible, it chooses a value that’s close to, but strictly less than, a power of 2.

If you’re curious, you can read the [complete analysis on how to pick min\_run](https://hg.python.org/cpython/file/tip/Objects/listsort.txt) under the Computing minrun section.

### **Measuring Timsort’s Big O Complexity**

On average, the complexity of Timsort is **O(n log2n)**, just like merge sort and Quicksort. The logarithmic part comes from doubling the size of the run to perform each linear merge operation.

However, Timsort performs exceptionally well on already-sorted or close-to-sorted lists, leading to a best-case scenario of O(n). In this case, Timsort clearly beats merge sort and matches the best-case scenario for Quicksort. But the worst case for Timsort is also O(n log2n), which surpasses Quicksort’s O(n2).

[](https://srv.realpython.net/click/28266526380/?c=30337084727&p=58946116052&r=40575)

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

### **Timing Your Timsort Implementation**

You can use run\_sorting\_algorithm() to see how Timsort performs sorting the ten-thousand-element array:

Python

1if \_\_name\_\_ == "\_\_main\_\_":

2 # Generate an array of `ARRAY\_LENGTH` items consisting

3 # of random integer values between 0 and 999

4 array = [randint(0, 1000) for i in range(ARRAY\_LENGTH)]

5

6 # Call the function using the name of the sorting algorithm

7 # and the array you just created

8 run\_sorting\_algorithm(algorithm="timsort", array=array)

Now execute the script to get the execution time of timsort:

Shell

$ python sorting.py

At 0.51 seconds, this Timsort implementation is a full 0.1 seconds, or 17 percent, faster than merge sort, though it doesn’t match the 0.11 of Quicksort. It’s also a ridiculous 11,000 percent faster than insertion sort!

Now try to sort an already-sorted list using these four algorithms and see what happens. You can modify your \_\_main\_\_ section as follows:

Python

1if \_\_name\_\_ == "\_\_main\_\_":

2 # Generate a sorted array of ARRAY\_LENGTH items

3 array = [i for i in range(ARRAY\_LENGTH)]

4

5 # Call each of the functions

6 run\_sorting\_algorithm(algorithm="insertion\_sort", array=array)

7 run\_sorting\_algorithm(algorithm="merge\_sort", array=array)

8 run\_sorting\_algorithm(algorithm="quicksort", array=array)

9 run\_sorting\_algorithm(algorithm="timsort", array=array)

If you execute the script now, then all the algorithms will run and output their corresponding execution time:

Shell

Algorithm: insertion\_sort. Minimum execution time: 53.5485634999991

Algorithm: merge\_sort. Minimum execution time: 0.372304601

Algorithm: quicksort. Minimum execution time: 0.24626494199999982

Algorithm: timsort. Minimum execution time: 0.23350277099999994

This time, Timsort comes in at a whopping thirty-seven percent faster than merge sort and five percent faster than Quicksort, flexing its ability to take advantage of the already-sorted runs.

Notice how Timsort benefits from two algorithms that are much slower when used by themselves. The genius of Timsort is in combining these algorithms and playing to their strengths to achieve impressive results.

### **Analyzing the Strengths and Weaknesses of Timsort**

The main disadvantage of Timsort is its complexity. Despite implementing a very simplified version of the original algorithm, it still requires much more code because it relies on both insertion\_sort() and merge().

One of Timsort’s advantages is its ability to predictably perform in O(n log2n) regardless of the structure of the input array. Contrast that with Quicksort, which can degrade down to O(n2). Timsort is also very fast for small arrays because the algorithm turns into a single insertion sort.

For real-world usage, in which it’s common to sort arrays that already have some preexisting order, Timsort is a great option. Its adaptability makes it an excellent choice for sorting arrays of any length.

## Conclusion

Sorting is an essential tool in any Pythonista’s toolkit. With knowledge of the different sorting algorithms in Python and how to maximize their potential, you’re ready to implement faster, more efficient apps and programs!

In this tutorial, you learned:

* How Python’s built-in **sort()** works behind the scenes
* What **Big O notation** is and how to use it to compare the efficiency of different algorithms
* How to measure the **actual time spent** running your code
* How to implement five different **sorting algorithms** in Python
* What the **pros and cons** are of using different algorithms

You also learned about different techniques such as **recursion**, **divide and conquer**, and **randomization**. These are fundamental building blocks for solving a long list of different algorithms, and they’ll come up again and again as you keep researching.

Take the code presented in this tutorial, create new experiments, and explore these algorithms further. Better yet, try implementing other sorting algorithms in Python. The list is vast, but [**selection sort**](https://en.wikipedia.org/wiki/Selection_sort), [**heapsort**](https://en.wikipedia.org/wiki/Heapsort), and [**tree sort**](https://en.wikipedia.org/wiki/Tree_sort) are three excellent options to s

https://github.com/Evans-mutuku/Data-StructuresPython.git <https://github.com/DaveOuma/DATA-STRUCTURES-ALGORITHMS-in-python.git>

A node is a basic unit of storing information>any data type/ collection of data.

Pointers – references to the next point

**Linked List Implementation**

Linked lists are fundamental data structures in cs programing. Unlike Arrays, which store elements in contiguous memory locations, linked lists consists of nodes connected by pointers, allowing for dynamic memory allocation and flexible data storage. Each node in a linked list contains data and a reference to the next node in the sequence forming a chain-like data structure.

**Node class with data and next attributes**

We define a node class to represent each element in a linked list. Each node has two attributes: data, which stores value of the element, and next, which points to the next node in the sequence.

**Methods for Insertion, deletion and traversal**

To interact with linked lists, we need to implement methods for inserting new nodes, deleting existing nodes and traversing the list to access or modify its elements. Insertion methods allow us to add new nodes at the beginning , end or middle of the list.

Deletion methods enable us to remove nodes based on their position or value. Traversal methods facilitate the iteration through the list to access each nodes data sequentially.

**Example**

Suppose we are managing a database of students records and need to store information such as studentId, name, age, and GPA. How can we use a linked list to efficiently organize and manage this data?

Soln

**File handling in python**

**Why do we use Input and Output in Python:**

Input and Output are used in Python to allow the user to interact with the program. The use of Input and Output enables the program to receive data from the user and provide results to the user.

Here are some common use cases for Input and Output in Python:

1. Taking input from the user: The input function is used to take input from the user, such as a name, age, or any other type of data. The input received from the user can be stored in a variable for further processing.
2. Displaying output to the user: The print function is used to display output to the user. This can be used to display the result of a calculation, the contents of a list or dictionary, or any other type of data.
3. Reading data from a file: Input can also be taken from a file, such as a text file, CSV file, or any other type of file. This can be done using the open function in Python or using third-party libraries like Pandas.
4. Writing data to a file: Output can also be stored in a file, such as a text file, CSV file, or any other type of file. This can be done using the open function in Python or using third-party libraries like Pandas.

Input and Output are essential in making the program more interactive and user-friendly. They also allow the program to receive and provide data, which is crucial for many applications.

## Python (Files I/O)

**1. Understanding how to read and write files:**

In Python, reading from and writing to files can be accomplished using the open function. The open function takes two arguments: the name of the file and the mode in which the file should be opened.

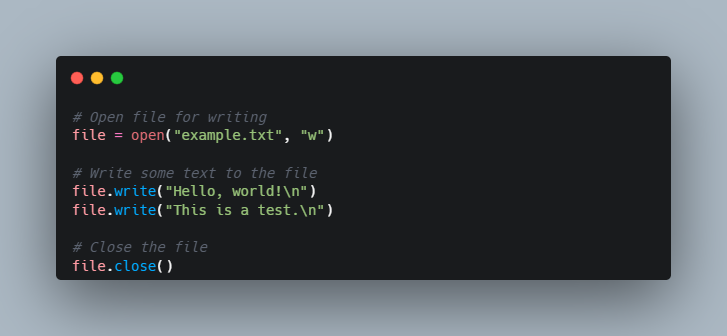
There are four modes in which a file can be opened in Python:

1. "r" (Read Only): This mode is used to only read the contents of a file. If the file does not exist, a FileNotFoundError will be raised.
2. "w" (Write Only): This mode is used to write to a file. If the file already exists, its contents will be truncated (i.e., deleted), and a new file with the same name will be created. If the file does not exist, a new file with the specified name will be created.
3. "a" (Append Only): This mode is used to append to an existing file. If the file does not exist, a new file with the specified name will be created.
4. "x" (Write Only, Exclusive Creation): This mode is used to write to a file only if the file does not already exist. If the file already exists, a FileExistsError will be raised.

**Here's an example of how to read from a file in Python:**

****

And here's an example of how to write to a file in Python:

****

It's recommended to use the with the statement when opening files in Python, as it automatically takes care of closing the file when the code block is exited, even if an exception is raised

**Reading Keyboard Input**

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are −

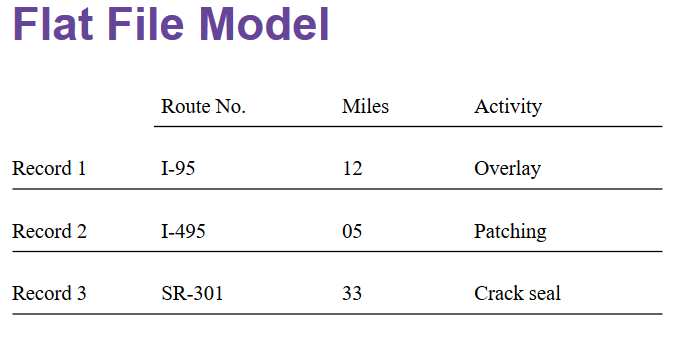
* raw\_input
* input

**The raw\_input Function**

The raw\_input([prompt]) function reads one line from standard input and returns it as a string (removing the trailing newline).

## ****Flat Files vs. Non-Flat Files****

Flat files are data files that contain records with no structured relationships between the records, and there's also no structure for indexing like you typically find it in relational databases. These files can contain only basic formatting, have a small fixed number of fields, and can or can not have a file format.



## ****Python File Objects****

Python has in-built functions to create, read, write, and manipulate accessible files. The io module is the default module for accessing files that can be used off the shelf without even importing it. Before you read, write, or manipulate the file, you need to make use of the module open(filename, access\_mode) that returns a file object called "handle". After which you can simply use this handle to read from or write to a file. Like everything else, files in Python are also treated as an object, which has its own attributes and methods.

An IOError exception is raised if, while opening the file, the operation fails. It could be due to various reasons like trying to read a file that is opened in write mode or accessing a file that is already closed.

As you already read before, there are two types of flat files, text and binary files:

* As you might have expected from reading the previous section, text files have an End-Of-Line (EOL) character to indicate each line's termination. In Python, the new line character (\n) is the default EOL terminator.
* Since binary files store data after converting it into a binary language (0s and 1s), there is no EOL character. This file type returns bytes. This file is used when dealing with non-text files such as images, .exe, or .pyc.

Let's now understand the Python file objects in detail, along with necessary examples.

**Open()**

The built-in Python function open() has the following arguments: open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None) The open() function has almost 8 parameters along with their default values for each argument as shown above.

You would be focusing on the first and second parameters for now, which are essential for reading and writing files. And go through other parameters one by one as the tutorial progresses.

Let's understand the first argument, i.e., file.

**File**

file is a mandatory argument that you have to provide to the open function while the rest of the arguments are optional and use their default values.

To put it simply, the file argument represents the path where your file resides in your system.

If the path is in the current working directory, you can just provide the filename. If not then you have to provide the absolute path of the file, just like in the following examples: my\_file\_handle=open("mynewtextfile.txt") If the file resides in a directory other than the current directory, you have to provide the absolute path with the file name:

****

Let's understand the second argument of the open function, i.e., access modes.

**Access Modes**

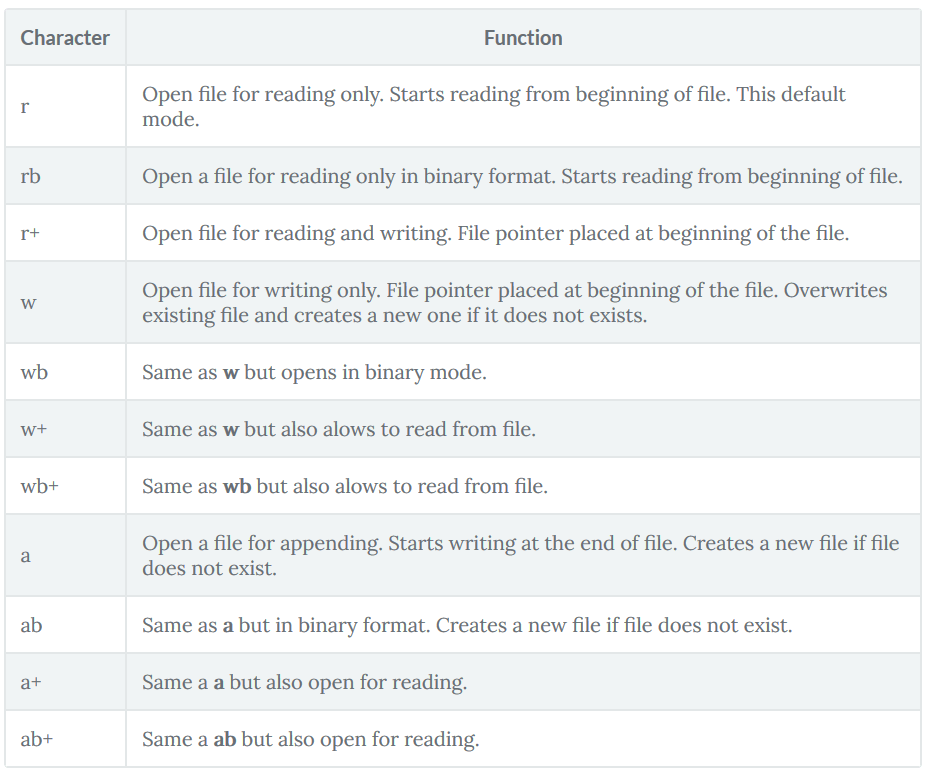
Access modes define in which way you want to open a file, whether you want to open a file in:

* read-only mode
* write-only mode
* append mode
* both read and write mode

Though a lot of access modes exist as shown in the below table, the most commonly used ones are read and write modes. It specifies where you want to start reading or writing in the file.

You use 'r', the default mode, to read the file. In other cases where you want to write or append, you use 'w' or 'a', respectively.

There are, of course, more access modes! Take a look at the following table:



As you have seen in the first section, there are two types of flat files. This is also why there's an option to specify which format you want to open, such as text or binary. Of course, the former is the default. When you add 'b' to the access modes, you can read the file in binary format rather than the default text format. It is used when the file to be accessed is not in text format.

Understanding the file methods (read, write, append, seek(), tell())

Content

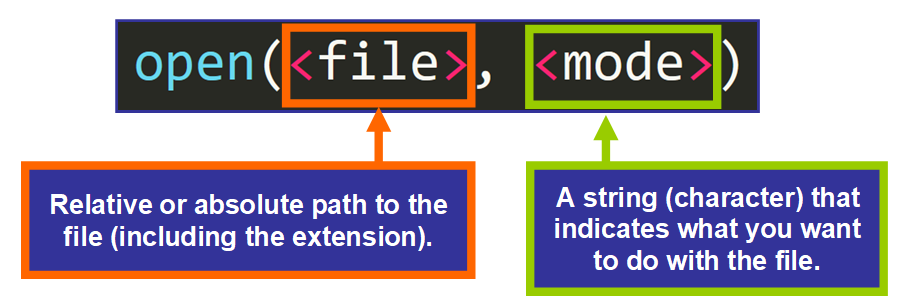
Discussion

The open function returns a file object, which has several methods that can be used to manipulate the file. Some of the most commonly used methods are:

1. read: This method is used to read the contents of a file. By default, the read method reads the entire contents of the file and returns it as a string. If a number is passed as an argument to the read method, it reads that many characters from the file and returns them as a string.
2. write: This method is used to write to a file. The argument passed to the write method is written to the file as a string. If the file was opened in write mode ("w"), the contents of the file will be truncated and the new data will be written to the file. If the file was opened in append mode ("a"), the new data will be appended to the end of the file.
3. seek: This method is used to change the file pointer's position. The argument passed to the seek method is the number of bytes to move the file pointer. The seek method takes two optional arguments: the first argument is the number of bytes to move the file pointer, and the second argument is the starting point from where the file pointer should be moved. The starting point can be one of the following: 0 (the beginning of the file), 1 (the current position of the file pointer), or 2 (the end of the file).
4. tell: This method is used to return the current position of

**Working with Files: Basic Syntax**

One of the most important functions that you will need to use as you work with files in Python is **open(),** a built-in function that opens a file and allows your program to use it and work with it.

**First Parameter: File**

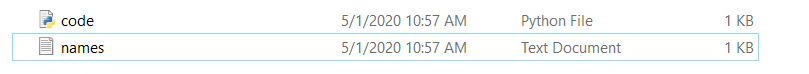
The first parameter of the open() function is **file**, the absolute or relative path to the file that you are trying to work with.

We usually use a relative path, which indicates where the file is located relative to the location of the script (Python file) that is calling the open() function.

For example, the path in this function call:

code here

Only contains the name of the file. This can be used when the file that you are trying to open is in the same directory or folder as the Python script, like this:



But if the file is within a nested folder, like this:



Then we need to use a specific path to tell the function that the file is within another folder.

In this example, this would be the path:

code here

Notice that we are writing data/ first (the name of the folder followed by a /) and then names.txt (the name of the file with the extension).

💡 **Tip:**The three letters .txt that follow the dot in names.txt is the "extension" of the file, or its type. In this case, .txt indicates that it's a text file.

**Second Parameter: Mode**

The second parameter of the open() function is the **mode**, a string with one character. That single character basically tells Python what you are planning to do with the file in your program.

Modes available are:

* Read ("r").
* Append ("a")
* Write ("w")
* Create ("x")

You can also choose to open the file in:

* Text mode ("t")
* Binary mode ("b")

To use text or binary mode, you would need to add these characters to the main mode. For example: "wb" means writing in binary mode.

💡 **Tip:** The default modes are read ("r") and text ("t"), which means "open for reading text" ("rt"), so you don't need to specify them in **open()** if you want to use them because they are assigned by default. You can simply write open(<file>).

**Why Modes?**

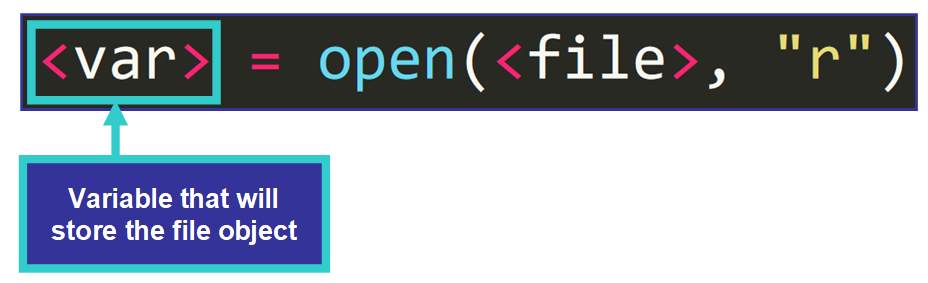
It really makes sense for Python to grant only certain permissions based what you are planning to do with the file, right? Why should Python allow your program to do more than necessary? This is basically why modes exist.

Think about it — allowing a program to do more than necessary can problematic. For example, if you only need to read the content of a file, it can be dangerous to allow your program to modify it unexpectedly, which could potentially introduce bugs.

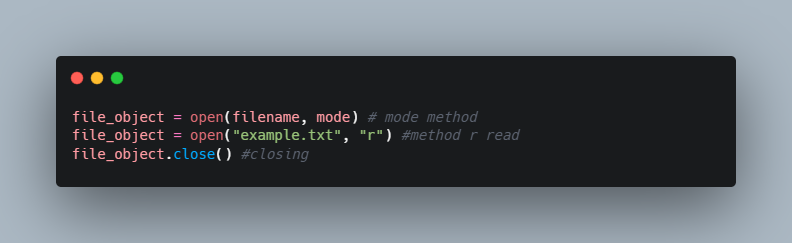
## ****🔸 How to Read a File****

Now that you know more about the arguments that the **open()** function takes, let's see how you can open a file and store it in a variable to use it in your program.

This is the basic syntax:



We are simply assigning the value returned to a variable. For example:



I know you might be asking: what type of value is returned by **open()**?

Well, **a** **file object**.

Let's talk a little bit about them.

**File Objects**

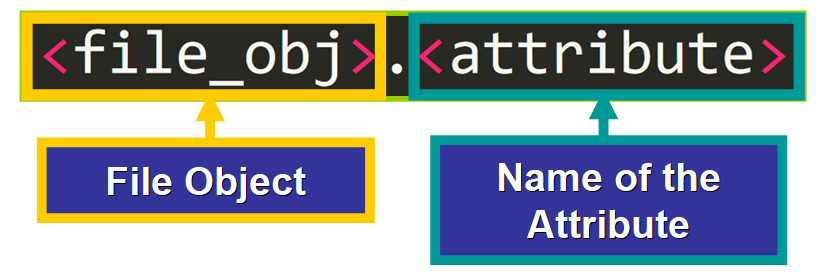
According to the [Python Documentation](https://docs.python.org/3/glossary.html#term-file-object), a **file object** is:

An object exposing a file-oriented API (with methods such as read() or write()) to an underlying resource.

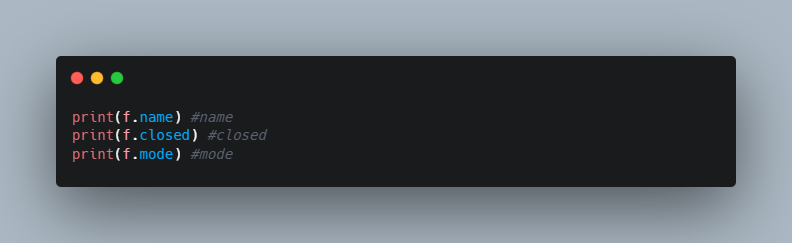
This is basically telling us that a file object is an object that lets us work and interact with existing files in our Python program.

File objects have attributes, such as:

* **name**: the name of the file.
* **closed**: True if the file is closed. False otherwise.
* **mode**: the mode used to open the file.



For example:

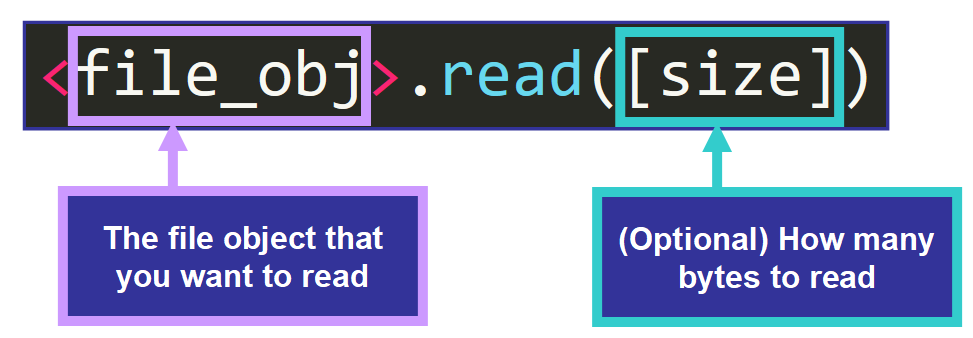


**Methods to Read a File**

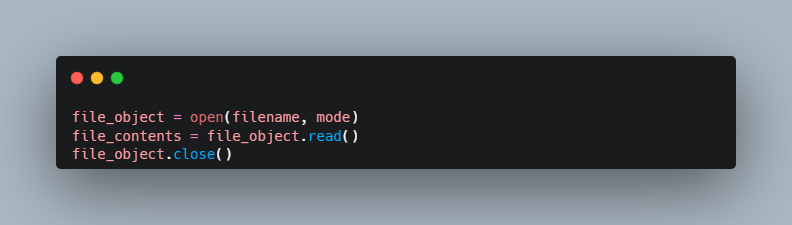
For us to be able to work with file objects, we need to have a way to "interact" with them in our program and that is exactly what methods do. Let's see some of them.

**Read()**

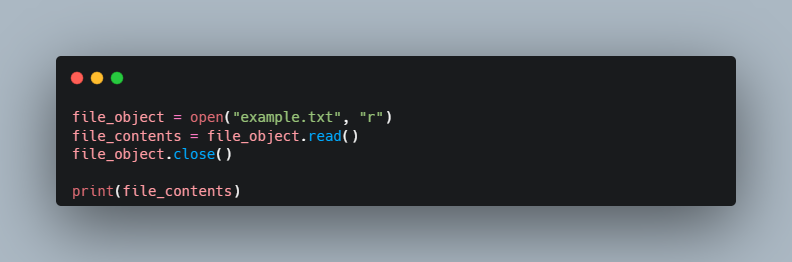
The first method that you need to learn about is **read()**,which **returns the entire content of the file as a string.**



Here we have an example:



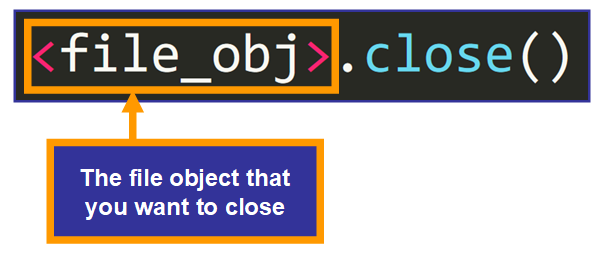
1. filename is the name of the file you want to read.
2. mode is the mode in which you want to open the file (e.g., 'r' for read mode).
3. file\_object is the file object that represents the file.
4. file\_contents is the string that contains the entire contents of the file.



You can use the type() function to confirm that the value returned by f.read() is a string:



❗️**Important:** You need to **close**a file after the task has been completed to free the resources associated to the file. To do this, you need to call the **close()** method, like this:

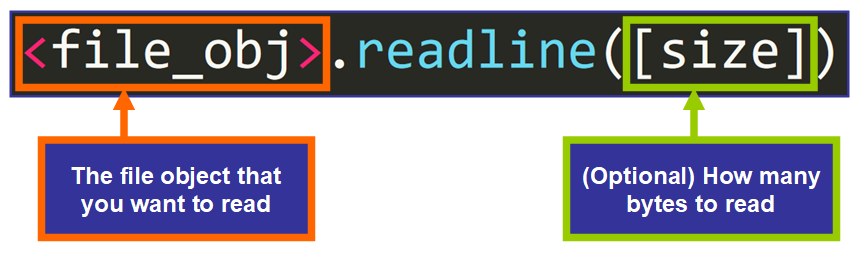


**Readline() vs. Readlines()**

You can read a file line by line with these two methods. They are slightly different, so let's see them in detail.

**readline()** reads **one line**of the file until it reaches the end of that line. A trailing newline character (\n) is kept in the string.

💡 **Tip:** Optionally, you can pass the size, the maximum number of characters that you want to include in the resulting string.



For example:

This is the first line of the file.

In contrast, **readlines()** returns a **list with all the lines** of the file as individual elements (strings). This is the syntax:



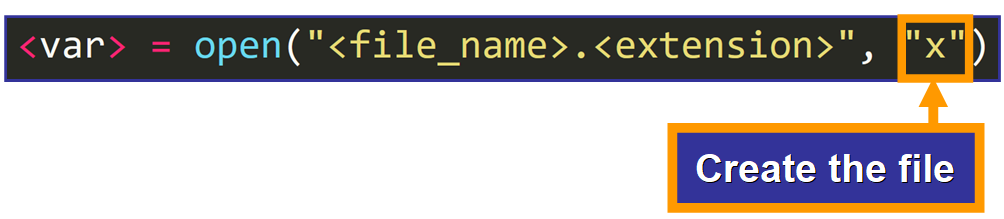
For example:

Those are the main methods used to read file objects. Now let's see how you can create files.

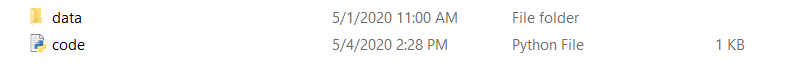
## ****🔹 How to Create a File****

If you need to create a file "dynamically" using Python, you can do it with the "x" mode.

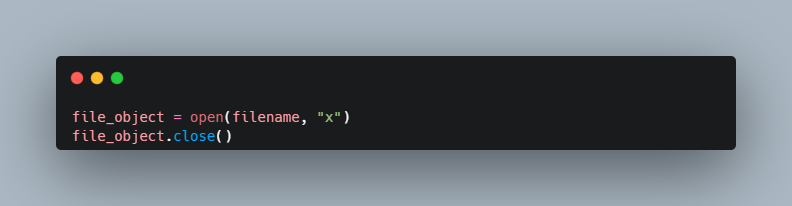
Let's see how. This is the basic syntax:

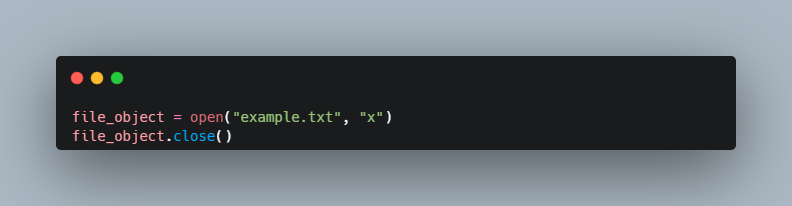


Here's an example. This is my current working directory:



If I run this line of code:





With this mode, you can create a file and then write to it dynamically using methods that you will learn in just a few moments.

💡 **Tip:** The file will be initially empty until you modify it.

A curious thing is that if you try to run this line again and a file with that name already exists, you will see this error:

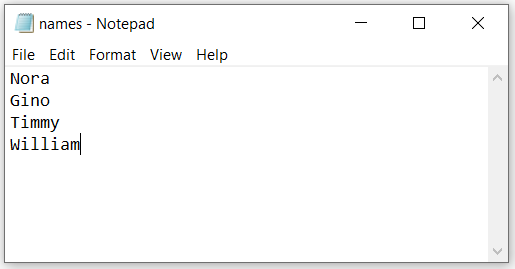
## ****🔸 How to Modify a File****

To modify (write to) a file, you need to use the **write()** method. You have two ways to do it (append or write) based on the mode that you choose to open it with. Let's see them in detail.

### **Append**

"Appending" means adding something to the end of another thing. The "a" mode allows you to open a file to append some content to it.

For example, if we have this file:

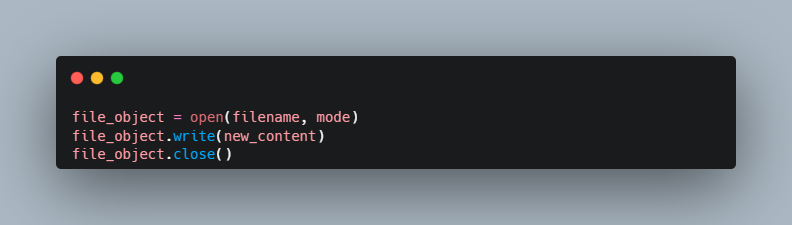


And we want to add a new line to it, we can open it using the **"a"** mode (append) and then, call the **write()** method, passing the content that we want to append as argument.

This is the basic syntax to call the **write()**method:

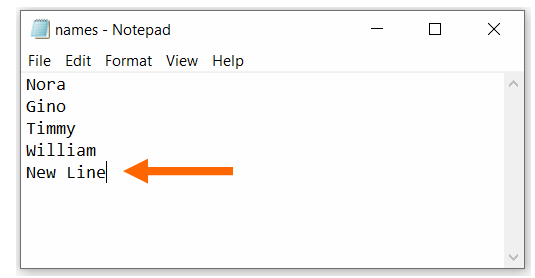


Here's an example:



💡 **Tip:** Notice that I'm adding \n before the line to indicate that I want the new line to appear as a separate line, not as a continuation of the existing line.

This is the file now, after running the script:

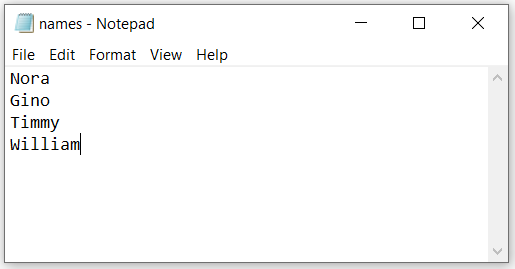


**💡 Tip:** The new line might not be displayed in the file until**f.close()** runs.

### **Write**

Sometimes, you may want to delete the content of a file and replace it entirely with new content. You can do this with the **write()** method if you open the file with the **"w"** mode.

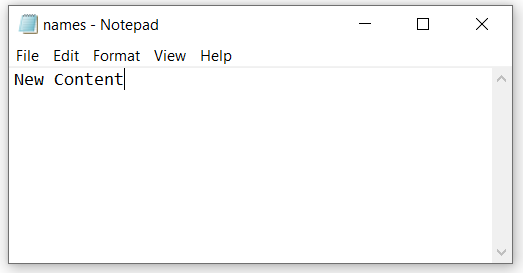
Here we have this text file:



If I run this script:



This is the result:

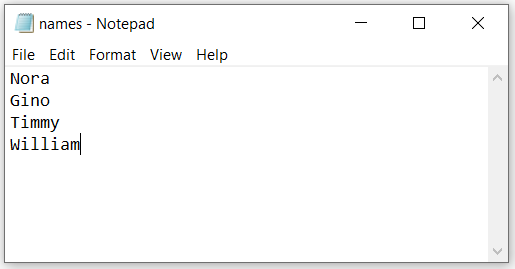


As you can see, opening a file with the **"w"** mode and then writing to it replaces the existing content.

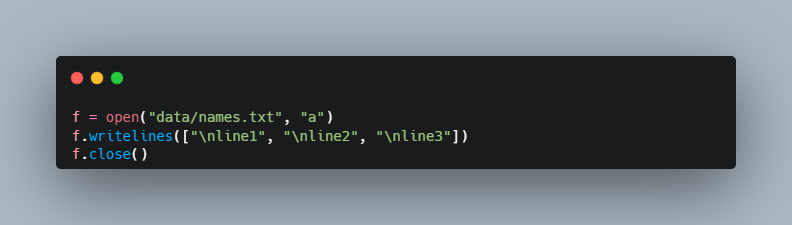
💡 **Tip:** The **write()** method returns the number of characters written.

If you want to write several lines at once, you can use the **writelines()** method, which takes a list of strings. Each string represents a line to be added to the file.

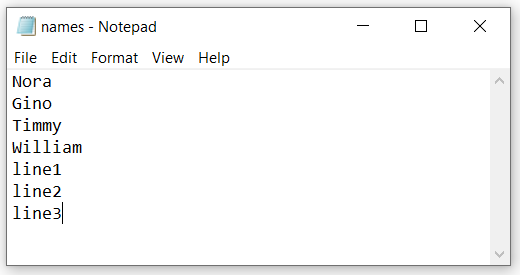
Here's an example. This is the initial file:



If we run this script:



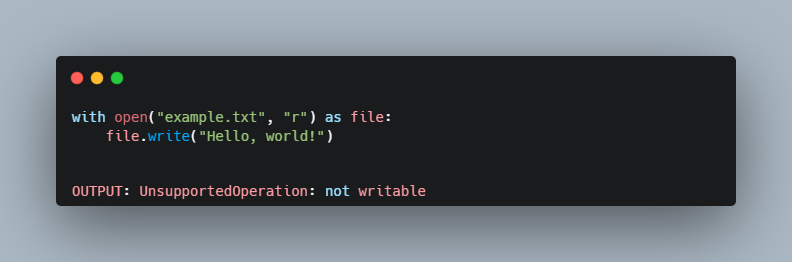
The lines are added to the end of the file:



### **Open File For Multiple Operations**

Now you know how to create, read, and write to a file, but what if you want to do more than one thing in the same program? Let's see what happens if we try to do this with the modes that you have learned so far:

If you open a file in "r" mode (read), and then try to write to it:



The same will occur with the "a" (append) mode.

How can we solve this? To be able to read a file and perform another operation in the same program, you need to add the "+" symbol to the mode, like this:



In this example, we first read the contents of the file using the read() method, and then write to the file using the write() method.

Note that when you open a file in "r+" mode, the file pointer is initially positioned at the beginning of the file. If you want to append to the end of the file, you should open the file in "a+" mode instead.

Very useful, right? This is probably what you will use in your programs, but be sure to include only the modes that you need to avoid potential bugs.

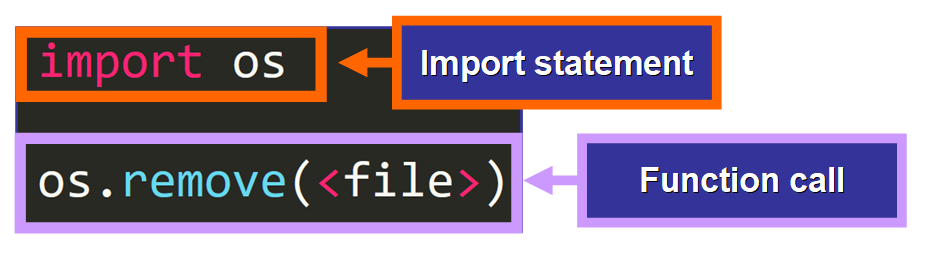
Sometimes files are no longer needed. Let's see how you can delete files using Python.

## ****🔹 How to Delete Files****

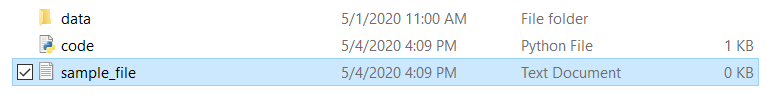
To remove a file using Python, you need to import a module called **os** which contains functions that interact with your operating system.

**💡 Tip:** A **module**is a Python file with related variables, functions, and classes.

Particularly, you need the **remove()**function. This function takes the path to the file as argument and deletes the file automatically.



Let's see an example. We want to remove the file called sample\_file.txt.



To do it, we write this code:



* The first line: import os is called an "import statement". This statement is written at the top of your file and it gives you access to the functions defined in the os module.
* The second line: os.remove("sample\_file.txt") removes the file specified.

💡 **Tip:** you can use an absolute or a relative path.

Now that you know how to delete files, let's see an interesting tool... Context Managers!

## ****🔸 Meet Context Managers****

Context Managers are Python constructs that will make your life much easier. By using them, you don't need to remember to close a file at the end of your program and you have access to the file in the particular part of the program that you choose.

**Syntax**

This is an example of a context manager used to work with files:



💡 **Tip:**The body of the context manager has to be indented, just like we indent loops, functions, and classes. If the code is not indented, it will not be considered part of the context manager.

When the body of the context manager has been completed, the file closes automatically.



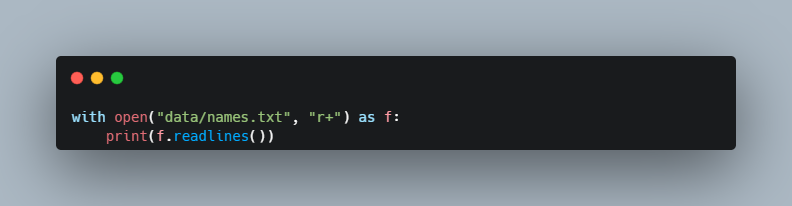
**Example**

Here's an example:



In this example, the MyContextManager class defines the \_\_enter\_\_() and \_\_exit\_\_() methods to be used as context managers. The \_\_enter\_\_() method is called when the with the block is entered, and the \_\_exit\_\_() method is called when the block is exited, either normally or due to an exception.

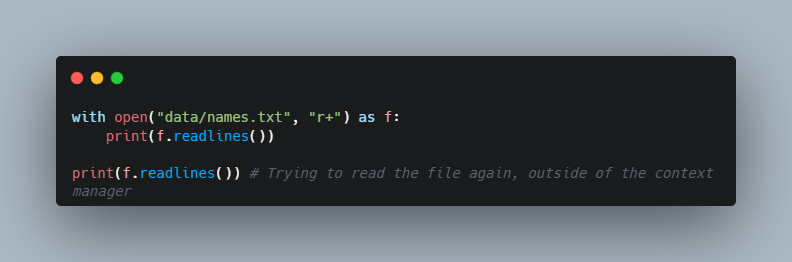
When an exception occurs inside the block, the \_\_exit\_\_() method is called with the exception type, value, and traceback as arguments. If the \_\_exit\_\_() method returns True, the exception is considered handled and the program continues. If it returns False or raises another exception, the original exception is propagated.



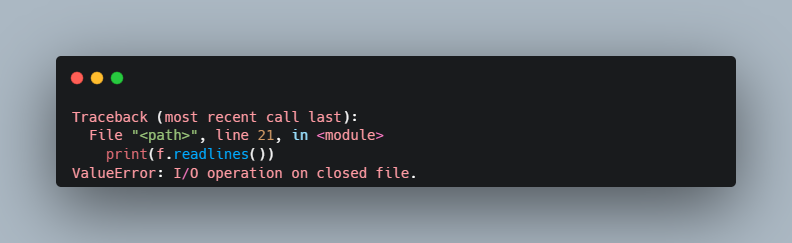
This context manager opens the names.txt file for read/write operations and assigns that file object to the variable f. This variable is used in the body of the context manager to refer to the file object.

**Trying to Read it Again**

After the body has been completed, the file is automatically closed, so it can't be read without opening it again. But wait! We have a line that tries to read it again, right here below:



Let's see what happens:



ValueError: I/O operation on closed file.

This error is thrown because we are trying to read a closed file. Awesome, right? The context manager does all the heavy work for us, it is readable, and concise.

How to handle exceptions and errors when reading/writing files

Content

Discussion

# **How to handle exceptions and errors when reading/writing files**

## ****What we are covering:****

* Exceptions
* The purpose of exception handling
* The try clause
* The except clause
* The else clause
* The finally clause
* How to raise exceptions

## ****Intro to Exceptions****

We will start with exceptions:

* **What**are they?
* **Why**are they relevant?
* **Why**should you handle them?

NB: Errors detected during execution are called **exceptions**and are not unconditionally fatal.

**Exceptions are raised when the program encounters an error during its execution.** They disrupt the normal flow of the program and usually end it abruptly. To avoid this, you can catch them and handle them appropriately.

You've probably seen them during your programming projects.

If you've ever tried to divide by zero in Python, you must have seen this error message:

code here

If you tried to index a string with an invalid index, you definitely got this error message:

code here

These are examples of exceptions.

**Common Exceptions**

There are many different types of exceptions, and they are all raised in particular situations. Some of the exceptions that you will most likely see as you work on your projects are:

* **IndexError** - raised when you try to index a list, tuple, or string beyond the permitted boundaries. For example:

code here

* **KeyError**- raised when you try to access the value of a key that doesn't exist in a dictionary. For example:

code here

* **NameError** - raised when a name that you are referencing in the code doesn't exist. For example:

code here

* **TypeError** - raised when an operation or function is applied to an object of an inappropriate type. For example:

code here

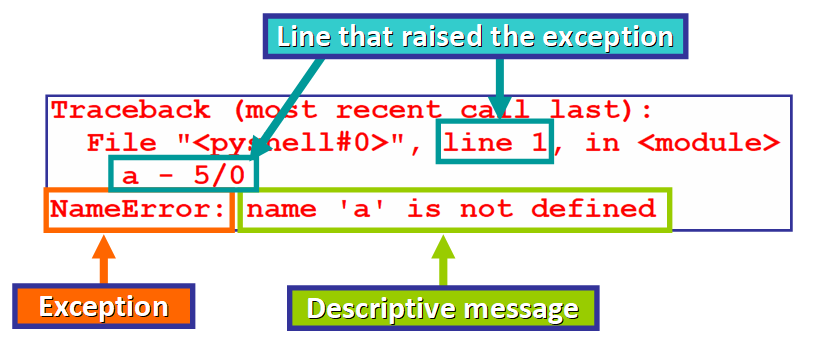
* **ZeroDivisionError**- raised when you try to divide by zero.

code here

💡 **Tips:** To learn more about other types of built-in exceptions, please [refer to this article](https://docs.python.org/3/library/exceptions.html) in the Python Documentation.

**Anatomy of an Exception**

I'm sure that you must have noticed a general pattern in these error messages. Let's break down their general structure piece by piece:



First, we find this line (see below). A **traceback**is basically a list detailing the function calls that were made before the exception was raised.

The traceback helps you during the debugging process because you can analyze the sequence of function calls that resulted in the exception:

Traceback (most recent call last):

Then, we see this line (see below) with the path to the file and the line that raised the exception. In this case, the path was the Python shell <pyshell#0> since the example was executed directly in IDLE.

File "<pyshell#0>", line 1, in <module>

a - 5/0

**💡 Tip:** If the line that raised the exception belongs to a function, <module> is replaced by the name of the function.

Finally, we see a descriptive message detailing the type of exception and providing additional information to help us debug the code:

NameError: name 'a' is not defined

## ****Exception Handling: Purpose & Context****

You may ask: why would I want to handle exceptions? Why is this helpful for me? By handling exceptions, you can provide an alternative flow of execution to avoid crashing your program unexpectedly.

**Example: User Input**

Imagine what would happen if a user who is working with your program enters an invalid input. This would raise an exception because an invalid operation was performed during the process.

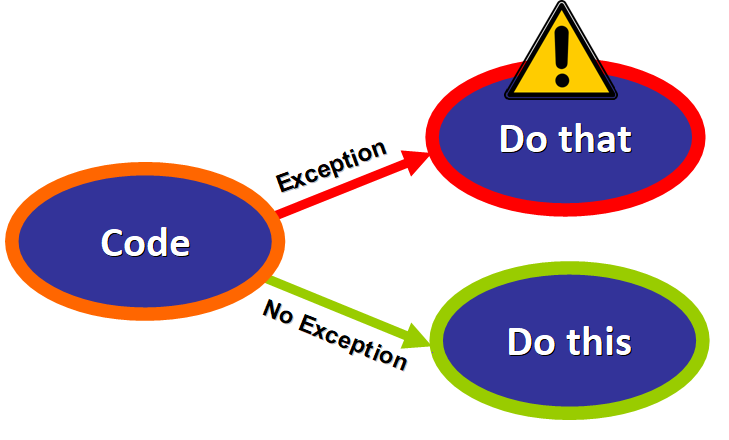
If your program doesn't handle this correctly, it will crash suddenly and the user will have a very disappointing experience with your product.

**But if you do handle the exception, you will be able to provide an alternative to improve the experience of the user.**

Perhaps you could display a descriptive message asking the user to enter a valid input, or you could provide a default value for the input. Depending on the context, you can choose what to do when this happens, and this is the magic of error handling. It can save the day when unexpected things happen. ⭐️

**What Happens Behind the Scenes?**

Basically, when we handle an exception, we are telling the program what to do if the exception is raised. In that case, the "alternative" flow of execution will come to the rescue. If no exceptions are raised, the code will run as expected.



## ****Time to Code: The try ... except Statement****

Now that you know what exceptions are and why you should we handle them, we will start diving into the built-in tools that the Python languages offers for this purpose.

**First, we have the most basic statement: try ... except.**

Let's illustrate this with a simple example. We have this small program that asks the user to enter the name of a student to display his/her age:

students = {"Nora": 15, "Gino": 30}

code here

Notice how we are not validating user input at the moment, so the user might enter invalid values (names that are not in the dictionary) and the consequences would be catastrophic because the program would crash if a KeyError is raised:

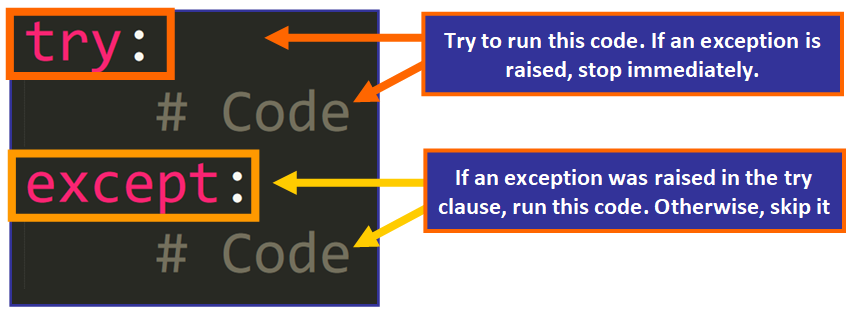
# User Input

Please enter the name of the student: "Daniel"

Code here

**Syntax**

We can handle this nicely using try ... except. This is the basic syntax:



In our example, we would add the try ... except statement within the function. Let's break this down piece by piece:

students = {"Nora": 15, "Gino": 30}

code here

If we "zoom in", we see the try ... except statement:

coder here

* When the function is called, the try clause will run. If no exceptions are raised, the program will run as expected.
* But if an exception is raised in the try clause, the flow of execution will immediately jump to the except clause to handle the exception.

**💡 Note:**This code is contained within a while loop to continue asking for user input if the value is invalid. This is an example:

Please enter the name of the student: "Lulu"

This name is not registered

Please enter the name of the student:

This is great, right? Now we can continue asking for user input if the value is invalid.

At the moment, we are handling all possible exceptions with the same except clause. But what if we only want to handle a specific type of exception? Let's see how we could do this.

**Catching Specific Exceptions**

Since not all types of exceptions are handled in the same way, we can specify which exceptions we would like to handle with this syntax:



This is an example. We are handling the ZeroDivisionError exception in case the user enters zero as the denominator:

code here

This would be the result:

# First iteration

1. Please enter the numerator: 5
2. Please enter the denominator: 0
3. Please enter a valid denominator.

# Second iteration

1. Please enter the numerator: 5
2. Please enter the denominator: 2

We are handling this correctly. But... if another type of exception is raised, the program will not handle it gracefully.

Here we have an example of a ValueError because one of the values is a float, not an int:

Please enter the numerator: 5

Please enter the denominator: 0.5

code here

We can customize how we handle different types of exceptions.

**Multiple Except Clauses**

To do this, we need to add multiple except clauses to handle different types of exceptions differently.

According to the [Python Documentation](https://docs.python.org/3/tutorial/errors.html#handling-exceptions):

A try statement may have **more than one except clause**, to specify handlers for different exceptions. **At most one handler will be executed**.

In this example, we have two except clauses. One of them handles ZeroDivisionError and the other one handles ValueError, the two types of exceptions that could be raised in this try block.

code here

💡 **Tip:**You have to determine which types of exceptions might be raised in the try block to handle them appropriately.

**Multiple Exceptions, One Except Clause**

You can also choose to handle different types of exceptions with the same except clause.

According to the [Python Documentation](https://docs.python.org/3/tutorial/errors.html#handling-exceptions):

An except clause may name **multiple exceptions** as a parenthesized tuple.

This is an example where we catch two exceptions (ZeroDivisionError and ValueError) with the same except clause:

The output would be the same for the two types of exceptions because they are handled by the same except clause:

Please enter the numerator: 5

Please enter the denominator: 0

Please enter valid integers.

Please enter the numerator: 0.5

Please enter valid integers.

Please enter the numerator:

**Handling Exceptions Raised by Functions Called in the try Clause**

An interesting aspect of exception handling is that if an exception is raised in a function that was previously called in the try clause of another function and the function itself does not handle it, the caller will handle it if there is an appropriate except clause.

According to the [Python Documentation](https://docs.python.org/3/tutorial/errors.html#handling-exceptions):

Exception handlers don’t just handle exceptions if they occur immediately in the try clause, but also **if they occur inside functions that are called (even indirectly) in the try clause.**

Let's see an example to illustrate this:

code here

We have the f function and the g function. f calls g in the try clause. With the argument 50, g will raise an IndexError because the index 50 is not valid for the string a.

But g itself doesn't handle the exception. Notice how there is no try ... except statement in the g function. Since it doesn't handle the exception, it "sends" it to f to see if it can handle it, as you can see in the diagram below:



Since f does know how to handle an IndexError, the situation is handled gracefully and this is the output:

Please enter a valid index

**💡 Note:** If f had not handled the exception, the program would have ended abruptly with the default error message for an IndexError.

**Accessing Specific Details of Exceptions**

Exceptions are objects in Python, so you can assign the exception that was raised to a variable. This way, you can print the default description of the exception and access its arguments.

According to the [Python Documentation](https://docs.python.org/3/tutorial/errors.html#handling-exceptions):

The except clause **may specify a variable after the exception name**. The variable is bound to an exception instance with the arguments stored in instance.args.

Here we have an example (see below) were we assign the instance of ZeroDivisionError to the variable e. Then, we can use this variable within the except clause to access the type of the exception, its message, and arguments.

code here

The corresponding output would be:

Please enter the numerator: 5

Please enter the denominator: 0

# Type

code here

# Message

division by zero

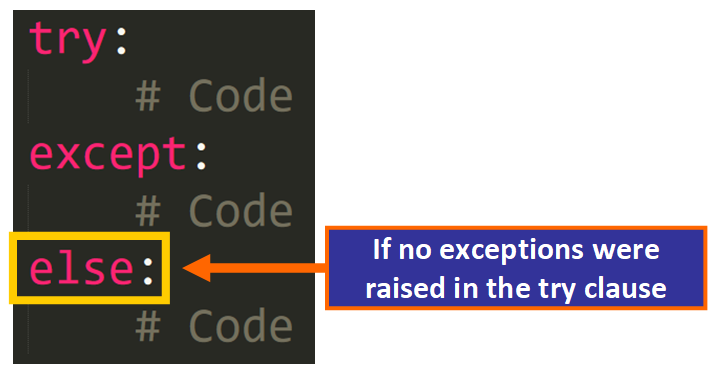
# Args

('division by zero',)

**💡 Tip:** If you are familiar with special methods, according to the [Python Documentation](https://docs.python.org/3/tutorial/errors.html#handling-exceptions): "for convenience, the exception instance defines [\_\_str\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__str__) so the arguments can be printed directly without having to reference .args."

## ****Now Let's Add: The "else" Clause****

The else clause is optional, but it's a great tool because it lets us execute code that should only run if no exceptions were raised in the try clause.



According to the [Python Documentation](https://docs.python.org/3/tutorial/errors.html#handling-exceptions):

The [try](https://docs.python.org/3/reference/compound_stmts.html#try) … [except](https://docs.python.org/3/reference/compound_stmts.html#except) statement has an **optional**else clause, which, when present, must follow all except clauses. It is useful for code that must be executed **if the try clause does not raise an exception.**

Here is an example of the use of the else clause:

code here

If no exception are raised, the result is printed:

Please enter the numerator: 5

Please enter the denominator: 5

But if an exception is raised, the result is not printed:

Please enter the numerator: 5

Please enter the denominator: 0

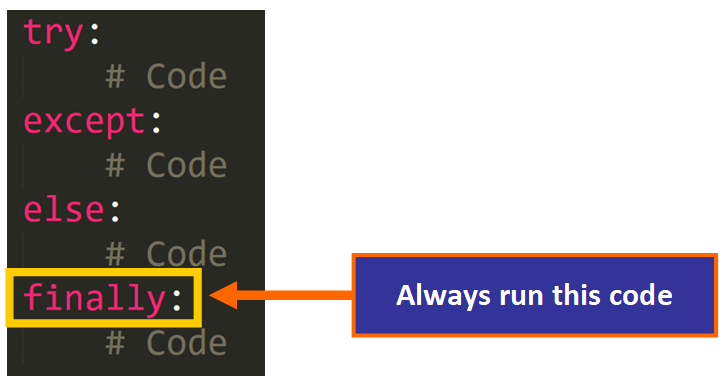
Please enter valid integers. The denominator can't be zero

💡 **Tip:** According to the [Python Documentation](https://docs.python.org/3/tutorial/errors.html#handling-exceptions):

The use of the else clause is better than adding additional code to the [try](https://docs.python.org/3/reference/compound_stmts.html#try) clause because it avoids accidentally catching an exception that wasn’t raised by the code being protected by the try … except statement.

## ****The "finally" Clause****

The finally clause is the last clause in this sequence. It is **optional**, but if you include it, it has to be the last clause in the sequence. The finally clause is **always**executed, even if an exception was raised in the try clause.



According to the [Python Documentation](https://docs.python.org/3/tutorial/errors.html#defining-clean-up-actions):

If a [finally](https://docs.python.org/3/reference/compound_stmts.html#finally) clause is present, the [finally](https://docs.python.org/3/reference/compound_stmts.html#finally) clause will execute as the last task before the [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement completes. The [finally](https://docs.python.org/3/reference/compound_stmts.html#finally) clause **runs whether or not the**[**try**](https://docs.python.org/3/reference/compound_stmts.html#try)**statement produces an exception.**

The finally clause is usually used to perform "clean-up" actions that should always be completed. For example, if we are working with a file in the try clause, we will always need to close the file, even if an exception was raised when we were working with the data.

Here is an example of the finally clause:

code here

This is the output when no exceptions were raised:

Please enter the numerator: 5

Please enter the denominator: 5

Inside the finally clause

This is the output when an exception was raised:

Please enter the numerator: 5

Please enter the denominator: 0

Please enter valid integers. The denominator can't be zero

Inside the finally clause

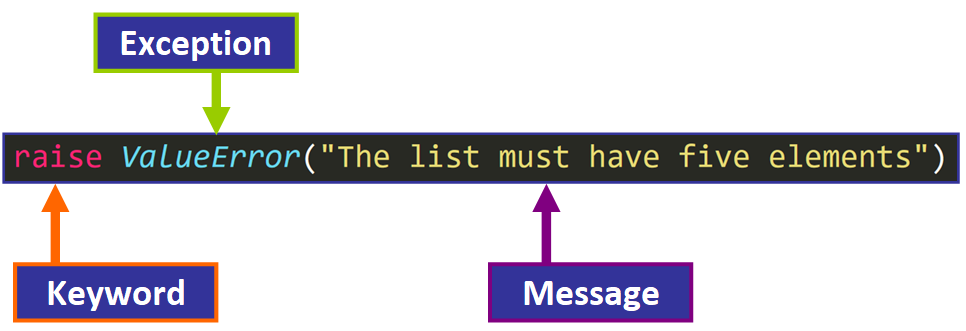
Notice how the finally clause **always**runs.

**❗️Important:** remember that the else clause and the finally clause are optional, but if you decide to include both, the finally clause has to be the last clause in the sequence.

## ****Raising Exceptions****

Now that you know how to handle exceptions in Python, I would like to share with you this helpful tip: **you can also choose when to raise exceptions in your code.**

This can be helpful for certain scenarios. Let's see how you can do this:



This line will raise a ValueError with a custom message.

Here we have an example (see below) of a function that prints the value of the items of a list or tuple, or the characters in a string. But you decided that you want the list, tuple, or string to be of length 5. You start the function with an if statement that checks if the length of the argument data is 5. If it isn't, a ValueError exception is raised:

code here

The output would be:

ValueError: The argument must have five elements

Notice how the last line displays the descriptive message:

ValueError: The argument must have five elements

You can then choose how to handle the exception with a try ... except statement. You could add an else clause and/or a finally clause. You can customize it to fit your needs.

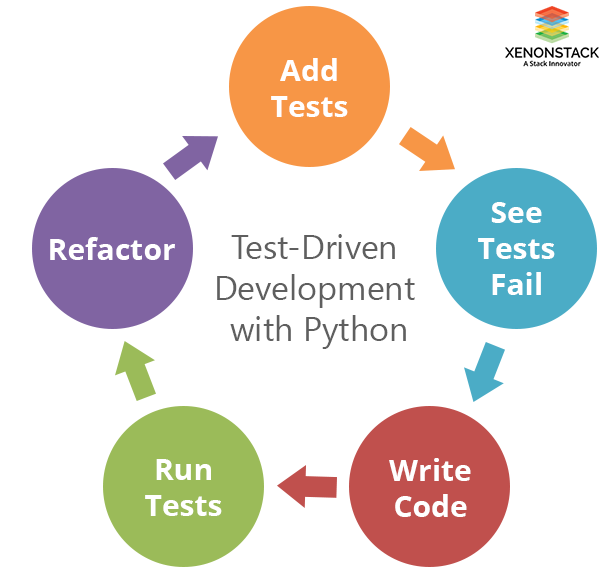
### Practical Lessons

Understanding test-driven development and writing tests

Content

Discussion

Test-driven development (TDD) is a software development practice where developers write tests before writing the actual code. The tests define the desired behavior of the code and serve as a specification for what the code should do. This approach helps ensure that the code meets the requirements and works as expected.



Here are the basic steps of TDD in Python:

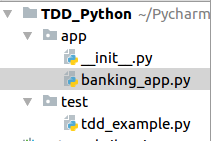
1. Write a test: Write a test that defines the desired behavior of your code. The test should check that the code produces the correct output given a specific input.
2. Run the test: Run the test to see if it fails. This is expected since you haven't written the code yet.
3. Write the code: Write the code to make the test pass. Your goal is to make the test pass, not to write the most efficient or optimal code.
4. Run the tests: Run the tests again to see if the code produces the expected output. If the tests pass, move on to the next test. If the tests fail, debug the code and fix the errors.
5. Refactor the code: After the tests pass, you can refactor the code to make it more efficient or to improve its design. The tests serve as a safety net, ensuring that your changes don't break the code.
6. Refactor the code: After the tests pass, you can refactor the code to make it more efficient or to improve its design. The tests serve as a safety net, ensuring that your changes don't break the code.
7. Repeat the process: Repeat the steps of writing tests, writing code, and refactoring for each feature or component of your application.

Writing tests in Python is done using a testing framework such as unittest, pytest, or nose. These frameworks provide the tools and structure to write and run tests, and they also provide features such as test discovery, test fixtures, and test reporting.

**Test Driven Development (TDD) in Python with Examples**

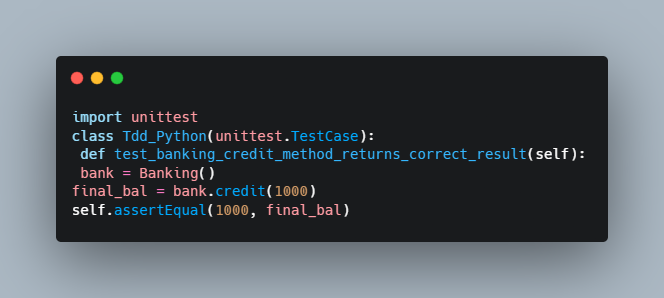
We are going to work on an example problem of banking. Let's say that our banking system introduced a new facility of credit money. So we have to add this to our program. Following the TDD approach before adding this credit feature, we first write our test for this functionality.

**Setting Up Environment For Test Driven Development (TDD)**

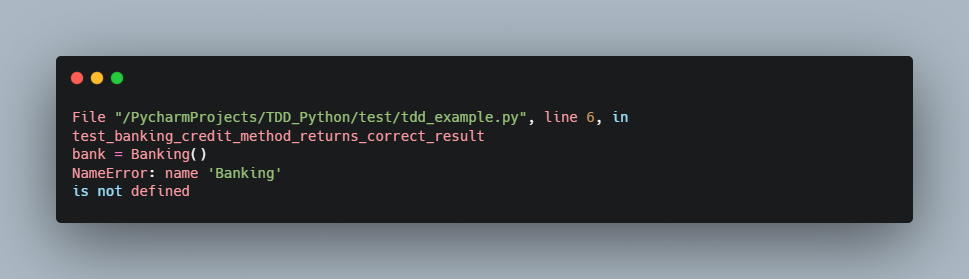


**Writing Test For Test Driven Development (TDD)**

So we write the following code for the unittest in the test/tdd\_example.py



Here first we import the unittest module and then write our test method. One thing which we should notice that every test method should start with the test word. Now we run this program.



We get an error here that is Banking not defined because we have not created it.

Here's another example of a simple test using the unittest framework:



In this example, we've defined a function add that takes two arguments and returns their sum. We've also defined a test case class TestAdd that tests the behavior of the add function. The test\_add method checks that the result of add(10, 20) is equal to 30. To run the tests, we run the script as a regular Python script.

**NUMPY**

**Numpy** stands for numerical python..

"""

 Pandas is an open source python library providing high-performance data manipulation and analysis tool using its powerful data structures

The name pandas is derived from the word panel data - an econometrics from multdimensional data

python with pandas is used in a wide range of fields including academics and commercial domains including finance, economics, statistics, analytics etc.

"""

**Matplotlib**

Is a versatile toolkit that allows for the creation of static, animated, and interactive visualizations in python. It overlays two APIS:  
**The pyplot API:** To make plot using matplotlib pylot;

**Object oriented Api –** a group of objects assembled with greater flexibility than pyplot. It provided direct access to matplotlibs backend layer.

Matplotlib simplifies simple tasks and enables complex tasks too be accomplished.

**Key aspects of matplotlib**

Matplotlib offers to create quality plots.

Matplot offers interactive figures and customizes their visual style that can be manipulated as per need.

Matplotlib offers export to many file formats.

**Plotting a sine wave**

From matplotlib import puplot as plt

Import numpy as np

Import math # needed for pi def

X= np.arange(0, math.pi\*2, 0.05)

Y= np.sin(x)

Plt.plot(x,y)

Plt.xlabel(“angle”)

Plt.ylabel(“sine”)

Plt.title(‘sine wave”)

Plt.show()s

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

DJANGO

Understanding Django fundamentals

Building dynamic web pages with Django templates

Working with Django models and dbs

Implementing user authentication and authorization

Creating RESTful APIS with Django rest framework

**Django installation process**

**Installation**

Create a project folder name

Run **pip install virtaulenv**

Install python intellisense extension

Open the root folder with cmd

Run the code virtualenv projectname – to create a virtual env for the project

Postgress

Pwd – josef

Port - 5432