**What is Python?**

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

* web development (server-side),
* software development,
* mathematics,
* system scripting.

### **What can Python do?**

* Python can be used on a server to create web applications.
* Python can be used alongside software to create workflows.
* Python can connect to database systems. It can also read and modify files.
* Python can be used to handle big data and perform complex mathematics.
* Python can be used for rapid prototyping, or for production-ready software development.

### **Why Python?**

* Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
* Python has a simple syntax similar to the English language.
* Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
* Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
* Python can be treated in a procedural way, an object-oriented way or a functional way.

### **Good to know**

* The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
* In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

### **Python Syntax compared to other programming languages**

* Python was designed for readability, and has some similarities to the English language with influence from mathematics.
* Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
* Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

## Python Install

Many PCs and Macs will have python already installed.

To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

C:\Users\Your Name>python --version

To check if you have python installed on a Linux or Mac, then on linux open the command line or on Mac open the Terminal and type:

python --version

If you find that you do not have Python installed on your computer, then you can download it for free from the following website: <https://www.python.org/>

## Python Quickstart

Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.

The way to run a python file is like this on the command line:

C:\Users\Your Name>python helloworld.py

Where "helloworld.py" is the name of your python file.

Let's write our first Python file, called helloworld.py, which can be done in any text editor.

helloworld.py

print("Hello, World!")

Simple as that. Save your file. Open your command line, navigate to the directory where you saved your file, and run:

C:\Users\Your Name>python helloworld.py

The output should read:

Hello, World!

Congratulations, you have written and executed your first Python program.

## The Python Command Line

To test a short amount of code in python sometimes it is quickest and easiest not to write the code in a file. This is made possible because Python can be run as a command line itself.

Type the following on the Windows, Mac or Linux command line:

C:\Users\Your Name>python

Or, if the "python" command did not work, you can try "py":

C:\Users\Your Name>py

From there you can write any python, including our hello world example from earlier in the tutorial:

C:\Users\Your Name>python  
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("Hello, World!")

Which will write "Hello, World!" in the command line:

C:\Users\Your Name>python  
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("Hello, World!")  
Hello, World!

Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:

exit()

**Python Data Structures**

**Lists**

Python Lists are ordered collections of data just like arrays in other programming languages. It allows different types of elements in the list. The implementation of Python List is similar to Vectors in C++ or ArrayList in JAVA. The costly operation is inserting or deleting the element from the beginning of the List as all the elements are needed to be shifted. Insertion and deletion at the end of the list can also become costly in the case where the preallocated memory becomes full.

### Example: Creating Python List

* Python3

|  |
| --- |
| List = [1, 2, 3, "NEC", 2.3]  print(List) |

**Output**

[1, 2, 3, 'NEC', 2.3]

List elements can be accessed by the assigned index. In python starting index of the list, a sequence is 0 and the ending index is (if N elements are there) N-1.

### Example: Python List Operations

* Python3

|  |
| --- |
| # Creating a List with  # the use of multiple values  List = ["N", "E", "Dog"]  print("\nList containing multiple values: ")  print(List)  # Creating a Multi-Dimensional List  # (By Nesting a list inside a List)  List2 = [['N', 'E'], ['Dog']]  print("\nMulti-Dimensional List: ")  print(List2)    # accessing a element from the  # list using index number  print("Accessing element from the list")  print(List[0])  print(List[2])  # accessing a element using  # negative indexing  print("Accessing element using negative indexing")  **Other List Methods :-** append() and extend() The **append()** method allows you to add another item to the end of your list. The method takes one required argument, which is the item you wish to add to your list.  Syntax: list.append(item)**toppings = ['pepperoni', 'sausage', 'mushroom']**toppings.append('onion')  --> ['pepperoni', 'sausage', 'mushroom', 'onion']  The **extend()** method is similar to append() in that it allows you to add onto your list; however, the extend() method allows you to add all of the items from another iterable (list, tuple, set, etc.) to the end of your list as separate items instead of one item. The method takes one required argument, the iterable.  Syntax: list.extend(iterable)**toppings = ['pepperoni', 'sausage', 'mushroom'] more\_toppings = ['onion', 'bacon']**toppings.**extend**(more\_toppings) --> ['pepperoni', 'sausage', 'mushroom', 'onion', 'bacon']To contrast...toppings.**append**(more\_toppings)--> ['pepperoni', 'sausage', 'mushroom', ['onion', 'bacon']]  As you can see above, when the extend() method is used with an iterable, each item in the iterable is added to the list as separate items no longer bounded. On the contrary, when the append() method is used with an iterable as the argument, the entire iterable is added to the list as one item. It’s always important to pay close attention to the commas and brackets present in a list. pop() and remove() The **pop()** method allows you to remove an element from your list at a specified index value. The method can take one optional argument, the integer value of the index you wish to remove — by default, pop() will remove the last item in the list, as the default value is -1.  Syntax: list.pop(index)**toppings = ['pepperoni', 'sausage', 'mushroom']**toppings.pop(1) --> ['pepperoni', 'mushroom']toppings.pop() --> ['pepperoni', 'sausage'] If you wanted to retrieve the removed item...extra = toppings.pop(1) extra --> 'sausage'  Like the pop() method, the **remove()** method allows you to remove an item from your list. The remove() method, though, removes the first occurrence of a specified value in a list. The method takes one required argument, the item you wish to remove.  Syntax: list.remove(item)**toppings = ['pepperoni', 'sausage', 'mushroom']**toppings.remove('sausage') --> ['pepperoni', 'mushroom']  For both pop() and remove(), if the argument is out of range or does not exist in the list, respectively, you will get an error. sort() and reverse() The **sort()**method sorts a list by certain criteria. The method can take two optional arguments. The first argument is setting either reverse=True or reverse=False. By default, this argument is set to reverse=False, which will result in alphabetical order if the list consists of only strings, or ascending order if the list consists of only numbers. The second argument allows you to set a key= to a function that you can use to specify how exactly you want your list sorted if it’s more complex than the default ordering that the sort() method does. This could be a built-in Python function, a function you defined elsewhere in your program, or an in-line [lambda function](https://www.youtube.com/watch?v=goPKxNtuxWo) that you write.  Syntax: list.sort(reverse=True|False, key=function)**toppings = ['pepperoni', 'sausage', 'mushroom']**toppings.sort() --> ['mushroom', 'pepperoni', 'sausage']toppings.sort(reverse=True) --> ['sausage', 'pepperoni', 'mushroom']toppings.sort(reverse=True, key=lambda x: len(x)) --> ['pepperoni', 'mushroom', 'sausage'] \* Sorted in reverse order by length of the topping name **prices = [1.50, 2.00, 0.50]**prices.sort(reverse=False) --> [0.50, 1.50, 2.00]prices.sort(reverse=True) --> [2.00, 1.50, 0.50]  **pies = [['bacon', 'ranch'], ['sausage', 'peppers']]**pies.sort(reverse=True) --> [['sausage', 'peppers'], ['bacon', 'ranch']] \* Sorts iterators by their first value  The **reverse()** method simply reverses the order of the items in the list. The method takes no arguments.  Syntax: list.reverse()**toppings = ['pepperoni', 'sausage', 'mushroom']**toppings.reverse() --> ['mushroom', 'sausage', 'pepperoni'] **prices = [1.50, 2.00, 0.50]**prices.reverse() --> [0.50, 2.00, 1.50] count() The **count()**method returns the number of occurrences of a specified item in a list. The method takes one required argument, which is the item you wish to find the count of. This method can be useful if you wish to find out what items appear more than once in a list.  Syntax: list.count(item)**toppings = ['pepperoni', 'sausage', 'mushroom', 'sausage']**toppings.count('sausage') --> 2toppings.count('pepperoni') --> 1toppings.count('bacon') --> 0 index() The **index()** method returns the index of the first occurrence of the specified item. The method takes one required argument, which is the item whose index you wish to find. If the item does not exist in the list, you will get an error.  Syntax: list.index(item)**toppings = ['pepperoni', 'sausage', 'mushroom', 'sausage']**toppings.index('mushroom') --> 2toppings.index('pepperoni') --> 0 insert() The **insert()** method inserts a specified item into a list at a specified index. The method takes two required arguments — the integer index you wish to insert the value at and the item you’d like to insert.  Syntax: list.insert(index, index)**toppings = ['pepperoni', 'sausage', 'mushroom']**toppings.insert(1, 'onion') --> ['pepperoni', 'onion', 'sausage', 'mushroom'] copy() The **copy()** method simply returns a copy of your list. The method takes no arguments.  Syntax: list.copy()**toppings = ['pepperoni', 'sausage', 'mushroom']**toppings**2** = toppings.copy() toppings**2** --> ['pepperoni', 'sausage', 'mushroom'] clear() The **clear()** method simply removes all items from a list, leaving an empty list. The method takes no arguments.  Syntax: list.clear()**toppings = ['pepperoni', 'sausage', 'mushroom']**toppings.clear() --> [] |

## Tuple

[Python tuples](https://www.geeksforgeeks.org/python-tuples/) are similar to lists but Tuples are [immutable](https://www.geeksforgeeks.org/why-do-we-need-immutables-in-python/) in nature i.e. once created it cannot be modified. Just like a List, a Tuple can also contain elements of various types.

In Python, tuples are created by placing a sequence of values separated by ‘comma’ with or without the use of parentheses for grouping of the data sequence.

**Note:**To create a tuple of one element there must be a trailing comma. For example, (8,) will create a tuple containing 8 as the element.

### Example: Python Tuple Operations

* Python3

|  |
| --- |
| # Creating a Tuple with  # the use of Strings  Tuple = ('Dog', 'Cat')  print("\nTuple with the use of String: ")  print(Tuple)  # Creating a Tuple with  # the use of list  list1 = [1, 2, 4, 5, 6]  print("\nTuple using List: ")  Tuple = tuple(list1)  # Accessing element using indexing  print("First element of tuple")  print(Tuple[0])  # Accessing element from last  # negative indexing  print("\nLast element of tuple")  print(Tuple[-1])  print("\nThird last element of tuple")  print(Tuple[-3]) |

**Output**

Tuple with the use of String:

('Dog', 'Cat')

Tuple using List:

First element of tuple

1

Last element of tuple

6

Third last element of tuple

4

**Other Tuple Methods :-**

## ****1. len() method****

This method returns number of elements in a [tuple](https://ladderpython.com/lesson/python-tuple-with-examples-tuple-in-python-3/).  
**Syntax:**  
**len(<tuple>)**  
<tuple> refers to user defined tuple whose length we want to find.  
**Example:**

|  |
| --- |
| **>>> T1=(10,20,30,40)**  **>>> len(T1)**  **4**  **#There are 4 element in tuple.** |

## ****2. max()****

This method returns largest element of a tuple. This method works only if the tuple contains all values of same type. If tuple contains values of different [data types](https://ladderpython.com/lesson/data-types-in-python/) then, it will give error stating that mixed type comparison is not possible:  
**Syntax :**  
**max(<tuple>)**  
<tuple> refers to name of tuple in which we want to find maximum value.  
**Example 1:**

|  |
| --- |
| **>>> T1=[10,20,30,40]**  **>>> max(T1)**  **40**  **# 40 is the maximum  value in tuple T1.** |

**Example 2:**

|  |
| --- |
| **>>> T2=['Sumit','Anil','Rahul']**  **>>> max(T2)**  **'Sumit'**  **#’Sumit’ appears as last member as per English dictionary.** |

**Example 3:**

|  |
| --- |
| **>>> T3=['Sumit',10,'Anil',11,'Rahul',12]**  **>>> max(T3)**  **Traceback (most recent call last):**  **File "<pyshell#7>", line 1, in <module>**  **max(T3)**  **TypeError: '>' not supported between instances of 'int' and 'str'**  **#Maximum can’t be found in mixed values.** |

**Example 4:**

|  |
| --- |
| **>>> max([3,4],(5,6))**  **Traceback (most recent call last):**  **File "<pyshell#0>", line 1, in <module>**  **max([3,4],(5,6))**  **TypeError: '>' not supported between instances of 'tuple' and 'tuple'**    **#Maximum can’t be found among tuple and tuples.** |

## ****3. min()****

This method returns smallest element of a tuple. This method works only if the tuple contains all values of same type. If tuple contains values of different data types then, it will give error stating that mixed type comparison is not possible:  
**Syntax :**  
**min(<tuple>)**  
<tuple> refers to name of tuple in which we want to find minimum value.  
**Example 1:**

|  |
| --- |
| **>>> T1=[10,20,30,40]**  **>>> min(T1)**  **10**  **#10 is the minimum  value.** |

**Example 2:**

|  |
| --- |
| **>>> T2=['Sumit','Anil','Rahul']**  **>>> min(T2)**  **'Anil'**  **#’Anil’ appears first as per English dictionary.** |

**Example 3:**

|  |
| --- |
| **>>> T3=['Sumit',10,'Anil',11,'Rahul',12]**  **>>> min(T3)**  **Traceback (most recent call last):**  **File "<pyshell#2>", line 1, in <module>**  **min(T3)**  **TypeError: '<' not supported between instances of 'int' and 'str'**  **#Minimum can’t be found in mixed values.** |

**Example 4:**

|  |
| --- |
| **>>> min([3,4],(5,6))**  **Traceback (most recent call last):**  **File "<pyshell#3>", line 1, in <module>**  **min([3,4],(5,6))**  **TypeError: '<' not supported between instances of 'tuple' and 'tuple'**  **#Minimum can’t be found among tuple and tuples.** |

## ****4. index()****

This method is used to find first index position of value in a tuple. It returns error if value is not found in the tuple.  
**Syntax:**  
**Tuple.index (<Val>)**  
**Tuple**is user defined tuple.  
**<Val>** refers to the value whose index we want to find in Tuple.  
**Example 1:**

|  |
| --- |
| **>>> T1=[13,18,11,16,18,14]**  **>>> print(T1.index(18))**  **1**  **#Index of first occurance of 18 is shown i.e. 1.** |

**Example 2:**

|  |
| --- |
| **>>> print(T1.index(10))**  **Traceback (most recent call last):**  **File “<pyshell#2>”, line 1, in <module>**  **T1.index(10)**  **ValueError: 10 is not in tuple**  **# Above example shows error as 10 is not in the tuple** |

## ****5. count()****

This function is used to count and return number of times a [value](https://ladderpython.com/lesson/literals-in-python/) exists in a tuple. If the given value is not in the tuple, it returns zero.  
**Syntax:**  
**Tuple.count(<value>)**  
<value> refers to the value whose count we want to find.  
**Example 1:**

|  |
| --- |
| **>>> T1=[13,18,11,16,18,14]**  **>>> T1.count(18)  #18 appears twice in tuple T1.**  **2** |

**Example 2:**

|  |
| --- |
| **>>> T1=[13,18,11,16,18,14]** **>>> T1.count(30)** **0 #0 is the output as 30 doesn’t exist in the tuple T1.** |

## ****6.****tuple()

This method is used to create a tuple from different types of values.  
**Syntax:**  
**Tuple(<sequence>)**  
<sequence> refers to a sequence type object that we want to convert to a tuple.  
**Example 1 – Creating empty tuple**

|  |
| --- |
| **>>> t=tuple()**  **>>> t**  **()** |

**Example 2 – Creating a tuple from a**[**list**](https://ladderpython.com/lesson/list-in-python-3-working-with-list-in-python/)

|  |
| --- |
| **>>>t=tuple([1,2,3])**  **>>>t**  **(1,2,3)** |

**Example 3  – Creating tuple from a**[**string**](https://ladderpython.com/lesson/string-in-python-3-manipulating-strings-in-python/)

|  |
| --- |
| **>>>t=tuple(“abc”)**  **>>>t**  **(‘a’,’b’,’c’)** |

**Example 4  – Creating a tuple from keys of a**[**dictionary**](https://ladderpython.com/lesson/python-dictionary-with-examples/)

|  |
| --- |
| **>>> t1=tuple({1:'a',2:'b'})**  **>>>t1**  **(1,2)** |

## DICTIONARY

## How to Create a Dictionary in Python

A dictionary in Python is made up of key-value pairs.

In the two sections that follow you will see two ways of creating a dictionary.

The first way is by using a set of curly braces, {}, and the second way is by using the built-in dict() function.

### **How to Create A Dictionary With Items in Python**

To create a dictionary with items, you need to include key-value pairs inside the curly braces.

The general syntax for this is the following:

dictionary\_name = {key: value}

Let's break it down:

* dictionary\_name is the variable name. This is the name the dictionary will have.
* = is the assignment operator that assigns the key:value pair to the dictionary\_name.
* You declare a dictionary with a set of curly braces, {}.
* Inside the curly braces you have a key-value pair. Keys are separated from their associated values with colon, :.

Let's see an example of creating a dictionary with items:

#create a dictionary

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

print(my\_information)

#check data type

print(type(my\_information))

#output

#{'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

#<class 'dict'>

In the example above, there is a sequence of elements within the curly braces.

Specifically, there are three key-value pairs: 'name': 'Dionysia', 'age': 28, and 'location': 'Athens'.

The keys are name, age, and location. Their associated values are Dionysia, 28, and Athens, respectively.

When there are multiple key-value pairs in a dictionary, each key-value pair is separated from the next with a comma, ,.

Let's see another example.

Say that you want to create a dictionary with items using the dict() function this time instead.

You would achieve this by using dict() and passing the curly braces with the sequence of key-value pairs enclosed in them as an argument to the function.

#create a dictionary with dict()

my\_information = dict({'name': 'Dionysia' ,'age': 28,'location': 'Athens'})

print(my\_information)

#check data type

print(type(my\_information))

#output

#{'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

#<class 'dict'>

It's worth mentioning the fromkeys() method, which is another way of creating a dictionary.

It takes a predefined sequence of items as an argument and returns a new dictionary with the items in the sequence set as the dictionary's specified keys.

You can optionally set a value for all the keys, but by default the value for the keys will be None.

The general syntax for the method is the following:

dictionary\_name = dict.fromkeys(sequence,value)

Let's see an example of creating a dictionary using fromkeys() without setting a value for all the keys:

#create sequence of strings

cities = ('Paris','Athens', 'Madrid')

#create the dictionary, `my\_dictionary`, using the fromkeys() method

my\_dictionary = dict.fromkeys(cities)

print(my\_dictionary)

#{'Paris': None, 'Athens': None, 'Madrid': None}

Now let's see another example that sets a value that will be the same for all the keys in the dictionary:

#create a sequence of strings

cities = ('Paris','Athens', 'Madrid')

#create a single value

continent = 'Europe'

my\_dictionary = dict.fromkeys(cities,continent)

print(my\_dictionary)

#output

#{'Paris': 'Europe', 'Athens': 'Europe', 'Madrid': 'Europe'}

## An Overview of Keys and Values in Dictionaries in Python

Keys inside a Python dictionary can **only be of a type that is immutable**.

Immutable data types in Python are integers, strings, tuples, floating point numbers, and Booleans.

Dictionary keys **cannot** be of a type that is mutable, such as sets, lists, or dictionaries.

So, say you have the following dictionary:

my\_dictionary = {True: "True", 1: 1, 1.1: 1.1, "one": 1, "languages": ["Python"]}

print(my\_dictionary)

#output

#{True: 1, 1.1: 1.1, 'one': 1, 'languages': ['Python']}

The keys in the dictionary are Boolean, integer, floating point number, and string data types, which are all acceptable.

If you try to create a key which is of a mutable type you'll get an error - specifically the error will be a TypeError.

my\_dictionary = {["Python"]: "languages"}

print(my\_dictionary)

#output

#line 1, in <module>

# my\_dictionary = {["Python"]: "languages"}

#TypeError: unhashable type: 'list'

In the example above, I tried to create a key which was of list type (a mutable data type). This resulted in a TypeError: unhashable type: 'list' error.

When it comes to values inside a Python dictionary there are no restrictions. Values can be of any data type - that is they can be both of mutable and immutable types.

Another thing to note about the differences between keys and values in Python dictionaries, is the fact that keys are **unique**. This means that a key can only appear once in the dictionary, whereas there can be duplicate values.

### **How to Find the Number of key-value Pairs Contained in a Dictionary in Python**

The len() function returns the total length of the object that is passed as an argument.

When a dictionary is passed as an argument to the function, it returns the total number of key-value pairs enclosed in the dictionary.

This is how you calcualte the number of key-value pairs using len():

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

print(len(my\_information))

#output

#3

### **How to View All key-value Pairs Contained in a Dictionary in Python**

To view every key-value pair that is inside a dictionary, use the built-in items() method:

year\_of\_creation = {'Python': 1993, 'JavaScript': 1995, 'HTML': 1993}

print(year\_of\_creation.items())

#output

#dict\_items([('Python', 1993), ('JavaScript', 1995), ('HTML', 1993)])

The items() method returns a list of tuples that contains the key-value pairs that are inside the dictionary.

### **How to View All keys Contained in a Dictionary in Python**

To see all of the keys that are inside a dictionary, use the built-in keys() method:

year\_of\_creation = {'Python': 1993, 'JavaScript': 1995, 'HTML': 1993}

print(year\_of\_creation.keys())

#output

#dict\_keys(['Python', 'JavaScript', 'HTML'])

The keys() method returns a list that contains only the keys that are inside the dictionary.

### **How to View All values Contained in a Dictionary in Python**

To see all of the values that are inside a dictionary, use the built-in values() method:

year\_of\_creation = {'Python': 1993, 'JavaScript': 1995, 'HTML': 1993}

print(year\_of\_creation.values())

#output

#dict\_values([1993, 1995, 1993])

The values() method returns a list that contains only the values that are inside the dictionary.

## How to Access Individual Items in A Dictionary in Python

When working with lists, you access list items by mentioning the list name and using square bracket notation. In the square brackets you specify the item's index number (or position).

You can't do exactly the same with dictionaries.

When working with dictionaries, you can't access an element by referencing its index number, since dictionaries contain key-value pairs.

Instead, you access the item by using the dictionary name and square bracket notation, but this time in the square brackets you specify a key.

Each key corresponds with a specific value, so you mention the key that is associated with the value you want to access.

The general syntax to do so is the following:

dictionary\_name[key]

Let's look at the following example on how to access an item in a Python dictionary:

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

#access the value associated with the 'age' key

print(my\_information['age'])

#output

#28

What happens though when you try to access a key that doesn't exist in the dictionary?

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

#try to access the value associated with the 'job' key

print(my\_information['job'])

#output

#line 4, in <module>

# print(my\_information['job'])

#KeyError: 'job'

It results in a KeyError since there is no such key in the dictionary.

One way to avoid this from happening is to first search to see if the key is in the dictionary in the first place.

You do this by using the in keyword which returns a Boolean value. It returns True if the key is in the dictionary and False if it isn't.

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

#search for the 'job' key

print('job' in my\_information)

#output

#False

Another way around this is to access items in the dictionary by using the get() method.

You pass the key you're looking for as an argument and get() returns the value that corresponds with that key.

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

#try to access the 'job' key using the get() method

print(my\_information.get('job'))

#output

#None

As you notice, when you are searching for a key that does not exist, by default get() returns None instead of a KeyError.

If instead of showing that default None value you want to show a different message when a key does not exist, you can customise get() by providing a different value.

You do so by passing the new value as the second optional argument to the get() method:

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

#try to access the 'job' key using the get() method

print(my\_information.get('job', 'This value does not exist'))

#output

#This value does not exist

Now when you are searching for a key and it is not contained in the dictionary, you will see the message This value does not exist appear on the console.

## How to Modify A Dictionary in Python

Dictionaries are mutable, which means they are changeable.

They can grow and shrink throughout the life of the program.

New items can be added, already existing items can be updated with new values, and items can be deleted.

### How to Add New Items to A Dictionary in Python

To add a key-value pair to a dictionary, use square bracket notation.

The general syntax to do so is the following:

dictionary\_name[key] = value

First, specify the name of the dictionary. Then, in square brackets, create a key and assign it a value.

Say you are starting out with an empty dictionary:

my\_dictionary = {}

print(my\_dictionary)

#output

#{}

Here is how you would add a key-value pair to my\_dictionary:

my\_dictionary = {}

#add a key-value pair to the empty dictionary

my\_dictionary['name'] = "John Doe"

#print dictionary

print(my\_list)

#output

#{'name': 'John Doe'}

Here is how you would add another new key-value pair:

my\_dictionary = {}

#add a key-value pair to the empty dictionary

my\_dictionary['name'] = "John Doe"

# add another key-value pair

my\_dictionary['age'] = 34

#print dictionary

print(my\_dictionary)

#output

#{'name': 'John Doe', 'age': 34}

Keep in mind that if the key you are trying to add already exists in that dictionary and you are assigning it a different value, the key will end up being updated.

Remember that keys need to be unique.

my\_dictionary = {'name': "John Doe", 'age':34}

print(my\_dictionary)

#try to create a an 'age' key and assign it a value

#the 'age' key already exists

my\_dictionary['age'] = 46

#the value of 'age' will now be updated

print(my\_dictionary)

#output

#{'name': 'John Doe', 'age': 34}

#{'name': 'John Doe', 'age': 46}

If you want to prevent changing the value of an already existing key by accident, you might want to check if the key you are trying to add is already in the dictionary.

You do this by using the in keyword as we discussed above:

my\_dictionary = {'name': "John Doe", 'age':34}

#I want to add an `age` key. Before I do so, I check to see if it already exists

print('age' in my\_dictionary)

#output

#True

### **How to Update Items in A Dictionary in Python**

Updating items in a dictionary works in a similar way to adding items to a dictionary.

When you know you want to update one existing key's value, use the following general syntax you saw in the previous section:

dictionary\_name[existing\_key] = new\_value

my\_dictionary = {'name': "John Doe", 'age':34}

my\_dictionary['age'] = 46

print(my\_dictionary)

#output

#{'name': 'John Doe', 'age': 46}

To update a dictionary, you can also use the built-in update() method.

This method is particularly helpful when you want to update more than one value inside a dictionary at the same time.

Say you want to update the name and age key in my\_dictionary, and add a new key, occupation:

my\_dictionary = {'name': "John Doe", 'age':34}

my\_dictionary.update(name= 'Mike Green', age = 46, occupation = "software developer")

print(my\_dictionary)

#output

#{'name': 'Mike Green', 'age': 46, 'occupation': 'software developer'}

The update() method takes a tuple of key-value pairs.

The keys that already existed were updated with the new values that were assigned, and a new key-value pair was added.

The update() method is also useful when you want to add the contents of one dictionary into another.

Say you have one dictionary, numbers, and a second dictionary, more\_numbers.

If you want to merge the contents of more\_numbers with the contents of numbers, use the update() method.

All the key-value pairs contained in more\_numbers will be added to the end of the numbers dictionary.

numbers = {'one': 1, 'two': 2, 'three': 3}

more\_numbers = {'four': 4, 'five': 5, 'six': 6}

#update 'numbers' dictionary

#you update it by adding the contents of another dictionary, 'more\_numbers',

#to the end of it

numbers.update(more\_numbers)

print(numbers)

#output

#{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6}

### **How to Delete Items from A Dictionary in Python**

One of the ways to delete a specific key and its associated value from a dictionary is by using the del keyword.

The syntax to do so is the following:

del dictionary\_name[key]

For example, this is how you would delete the location key from the my\_information dictionary:

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

del my\_information['location']

print(my\_information)

#output

#{'name': 'Dionysia', 'age': 28}

If you want to remove a key, but would also like to save that removed value, use the built-in pop() method.

The pop() method removes but also returns the key you specify. This way, you can store the removed value in a variable for later use or retrieval.

You pass the key you want to remove as an argument to the method.

Here is the general syntax to do that:

dictionary\_name.pop(key)

To remove the location key from the example above, but this time using the pop() method and saving the value associated with the key to a variable, do the following:

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

city = my\_information.pop('location')

print(my\_information)

print(city)

#output

#{'name': 'Dionysia', 'age': 28}

#Athens

If you specify a key that does not exist in the dictionary you will get a KeyError error message:

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

my\_information.pop('occupation')

print(my\_information)

#output

#line 3, in <module>

# my\_information.pop('occupation')

#KeyError: 'occupation'

A way around this is to pass a second argument to the pop() method.

By including the second argument there would be no error. Instead, there would be a silent fail if the key didn't exist, and the dictionary would remain unchanged.

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

my\_information.pop('occupation','Not found')

print(my\_information)

#output

#{'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

The pop() method removes a specific key and its associated value – but what if you only want to delete the **last** key-value pair from a dictionary?

For that, use the built-in popitem() method instead.

This is general syntax for the popitem() method:

dictionary\_name.popitem()

The popitem() method takes no arguments, but removes and returns the last key-value pair from a dictionary.

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

popped\_item = my\_information.popitem()

print(my\_information)

print(popped\_item)

#output

#{'name': 'Dionysia', 'age': 28}

#('location', 'Athens')

Lastly, if you want to delete all key-value pairs from a dictionary, use the built-in clear() method.

my\_information = {'name': 'Dionysia', 'age': 28, 'location': 'Athens'}

my\_information.clear()

print(my\_information)

#output

#{}

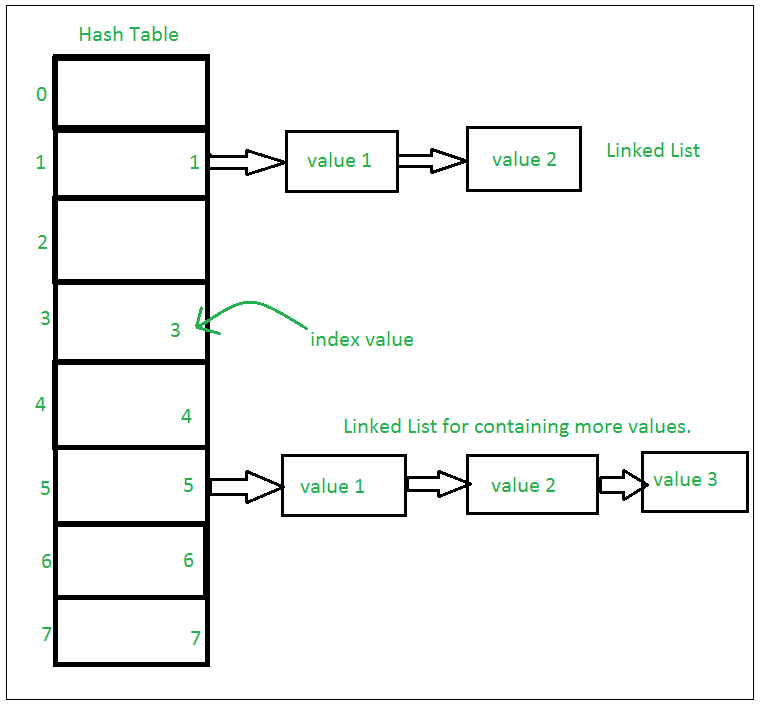
Using this method will leave you with an empty dictionary.

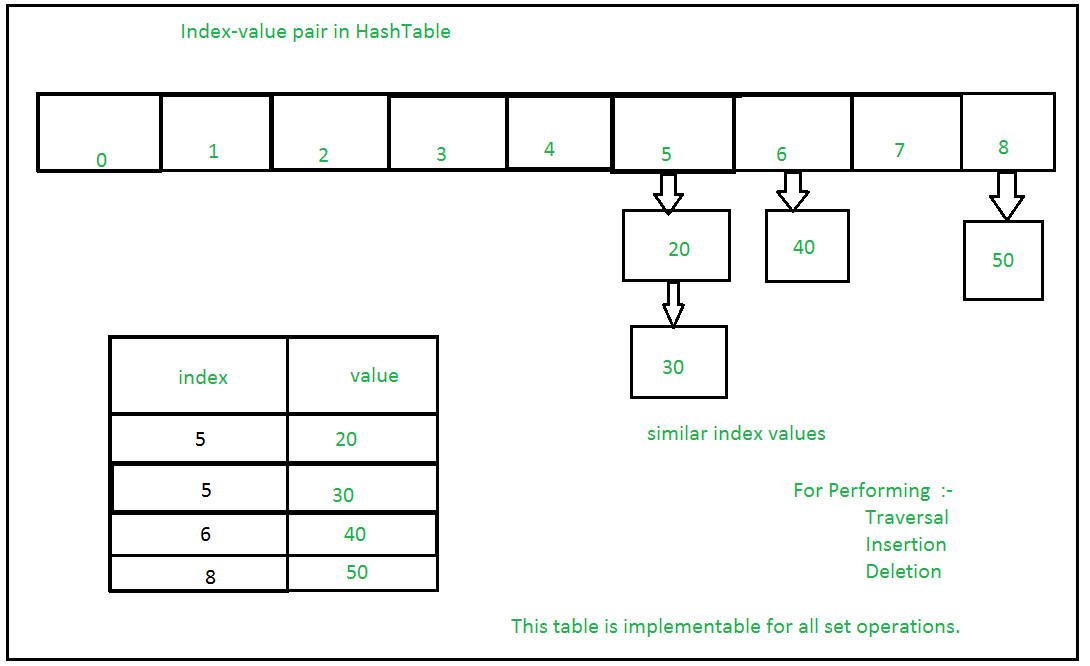
## Set

[Python set](http://www.geeksforgeeks.org/sets-in-python/) is a mutable collection of data that does not allow any duplication. Sets are basically used to include membership testing and eliminating duplicate entries. The data structure used in this is Hashing, a popular technique to perform insertion, deletion, and traversal in O(1) on average.

If Multiple values are present at the same index position, then the value is appended to that index position, to form a Linked List. In, CPython Sets are implemented using a dictionary with dummy variables, where key beings the members set with greater optimizations to the time complexity.

**Set Implementation:**

**Sets with Numerous operations on a single HashTable:**



### Example: Python Set Operations

* Python3

|  |
| --- |
| # Creating a Set with  # a mixed type of values  # (Having numbers and strings)  Set = set([1, 2, 'Dog', 4, 'Cat', 6, 'Dogs'])  print("\nSet with the use of Mixed Values")  print(Set)  # Accessing element using  # for loop  print("\nElements of set: ")  for i in Set:      print(i, end =" ")  print()  # Checking the element  # using in keyword  print("Dog" in Set) |

**Output**

Set with the use of Mixed Values

{1, 2, 4, 6, 'Cat', 'Dog'}

Elements of set:

1 2 4 6 Cat Dogs

True

**Other Set Methods :-**

### [**Python frozenset()**](https://www.programiz.com/python-programming/methods/built-in/frozenset)

returns immutable frozenset object

### [**Python Set add()**](https://www.programiz.com/python-programming/methods/set/add)

adds element to a set

### [**Python Set clear()**](https://www.programiz.com/python-programming/methods/set/clear)

remove all elements from a set

### [**Python Set copy()**](https://www.programiz.com/python-programming/methods/set/copy)

Returns Shallow Copy of a Set

### [**Python Set difference()**](https://www.programiz.com/python-programming/methods/set/difference)

Returns Difference of Two Sets

### [**Python Set difference\_update()**](https://www.programiz.com/python-programming/methods/set/difference_update)

Updates Calling Set With Intersection of Sets

### [**Python Set discard()**](https://www.programiz.com/python-programming/methods/set/discard)

Removes an Element from The Set

### [**Python Set intersection()**](https://www.programiz.com/python-programming/methods/set/intersection)

Returns Intersection of Two or More Sets

### [**Python Set intersection\_update()**](https://www.programiz.com/python-programming/methods/set/intersection_update)

Updates Calling Set With Intersection of Sets

### [**Python Set isdisjoint()**](https://www.programiz.com/python-programming/methods/set/isdisjoint)

Checks Disjoint Sets

### [**Python Set issubset()**](https://www.programiz.com/python-programming/methods/set/issubset)

Checks if a Set is Subset of Another Set

### [**Python Set issuperset()**](https://www.programiz.com/python-programming/methods/set/issuperset)

Checks if a Set is Superset of Another Set

### [**Python Set pop()**](https://www.programiz.com/python-programming/methods/set/pop)

Removes an Arbitrary Element

### [**Python Set remove()**](https://www.programiz.com/python-programming/methods/set/remove)

removes the specified element from the set

### [**Python Set symmetric\_difference()**](https://www.programiz.com/python-programming/methods/set/symmetric_difference)

Returns the symmetric difference of sets

### [**Python Set symmetric\_difference\_update()**](https://www.programiz.com/python-programming/methods/set/symmetric_difference_update)

Updates the Set with symmetric difference

### [**Python Set union()**](https://www.programiz.com/python-programming/methods/set/union)

Returns the union of sets

### [**Python Set update()**](https://www.programiz.com/python-programming/methods/set/update)

Add elements to the set

## Frozen Sets

[Frozen sets](https://www.geeksforgeeks.org/frozenset-in-python/) in Python are immutable objects that only support methods and operators that produce a result without affecting the frozen set or sets to which they are applied. While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.

### Example: Python Frozen set

* Python3

|  |
| --- |
| # Same as {"a", "b","c"}  normal\_set = set(["a", "b","c"])  print("Normal Set")  print(normal\_set)  # A frozen set  frozen\_set = frozenset(["e", "f", "g"])  print("\nFrozen Set")  print(frozen\_set)  # Uncommenting below line would cause error as  # we are trying to add element to a frozen set  # frozen\_set.add("h") |

**Output**

Normal Set

{'a', 'b', 'c'}

Frozen Set

frozenset({'f', 'g', 'e'})

## String

[Python Strings](https://www.geeksforgeeks.org/python-strings/) is the immutable array of bytes representing Unicode characters. Python does not have a character data type, a single character is simply a string with a length of 1.

**Note:** As strings are immutable, modifying a string will result in creating a new copy.

### Example: Python Strings Operations

* Python3

|  |
| --- |
| String = "Welcome to NEC"  print("Creating String: ")  print(String)  # Printing First character  print("First character of String is: ")  print(String[0])  # Printing Last character  print("Last character of String is: ")  print(String[-1]) |

**Output**

Creating String:

Welcome to NEC

First character of String is:

W

Last character of String is: C

**Other String Methods :-**

### [**Python String capitalize()**](https://www.programiz.com/python-programming/methods/string/capitalize)

Converts first character to Capital Letter

### [**Python String casefold()**](https://www.programiz.com/python-programming/methods/string/casefold)

converts to case folded strings

### [**Python String center()**](https://www.programiz.com/python-programming/methods/string/center)

Pads string with specified character

### [**Python String count()**](https://www.programiz.com/python-programming/methods/string/count)

returns occurrences of substring in string

### [**Python String endswith()**](https://www.programiz.com/python-programming/methods/string/endswith)

Checks if String Ends with the Specified Suffix

### [**Python String expandtabs()**](https://www.programiz.com/python-programming/methods/string/expandtabs)

Replaces Tab character With Spaces

### [**Python String find()**](https://www.programiz.com/python-programming/methods/string/find)

Returns the index of first occurrence of substring

### [**Python String format()**](https://www.programiz.com/python-programming/methods/string/format)

formats string into nicer output

### [**Python String format\_map()**](https://www.programiz.com/python-programming/methods/string/format_map)

Formats the String Using Dictionary

### [**Python String index()**](https://www.programiz.com/python-programming/methods/string/index)

Returns Index of Substring

### [**Python String isalnum()**](https://www.programiz.com/python-programming/methods/string/isalnum)

Checks Alphanumeric Character

### [**Python String isalpha()**](https://www.programiz.com/python-programming/methods/string/isalpha)

Checks if All Characters are Alphabets

### [**Python String isdecimal()**](https://www.programiz.com/python-programming/methods/string/isdecimal)

Checks Decimal Characters

### [**Python String isdigit()**](https://www.programiz.com/python-programming/methods/string/isdigit)

Checks Digit Characters

### [**Python String islower()**](https://www.programiz.com/python-programming/methods/string/islower)

Checks if all Alphabets in a String are Lowercase

### [**Python String isnumeric()**](https://www.programiz.com/python-programming/methods/string/isnumeric)

Checks Numeric Characters

### [**Python String isprintable()**](https://www.programiz.com/python-programming/methods/string/isprintable)

Checks Printable Character

### [**Python String isspace()**](https://www.programiz.com/python-programming/methods/string/isspace)

Checks Whitespace Characters

### [**Python String isupper()**](https://www.programiz.com/python-programming/methods/string/isupper)

returns if all characters are uppercase characters

### [**Python String join()**](https://www.programiz.com/python-programming/methods/string/join)

Returns a Concatenated String

### [**Python String lower()**](https://www.programiz.com/python-programming/methods/string/lower)

returns lowercased string

### [**Python String lstrip()**](https://www.programiz.com/python-programming/methods/string/lstrip)

Removes Leading Characters

### [**Python String partition()**](https://www.programiz.com/python-programming/methods/string/partition)

Returns a Tuple

### [**Python String replace()**](https://www.programiz.com/python-programming/methods/string/replace)

Replaces Substring Inside

### [**Python String rfind()**](https://www.programiz.com/python-programming/methods/string/rfind)

Returns the Highest Index of Substring

# **Python Regular Expressions**



Regular expressions are a powerful language for matching text patterns. This page gives a basic introduction to regular expressions themselves sufficient for our Python exercises and shows how regular expressions work in Python. The Python "re" module provides regular expression support.

In Python a regular expression search is typically written as:

match = re.search(pat, str)

The re.search() method takes a regular expression pattern and a string and searches for that pattern within the string. If the search is successful, search() returns a match object or None otherwise. Therefore, the search is usually immediately followed by an if-statement to test if the search succeeded, as shown in the following example which searches for the pattern 'word:' followed by a 3 letter word (details below):

import re  
  
str = 'an example word:cat!!'  
match = re.search(r'word:\w\w\w', str)  
# If-statement after search() tests if it succeeded  
if match:  
  print('found', match.group()) ## 'found word:cat'  
else:  
  print('did not find')

The code match = re.search(pat, str) stores the search result in a variable named "match". Then the if-statement tests the match -- if true the search succeeded and match.group() is the matching text (e.g. 'word:cat'). Otherwise if the match is false (None to be more specific), then the search did not succeed, and there is no matching text.

The 'r' at the start of the pattern string designates a python "raw" string which passes through backslashes without change which is very handy for regular expressions (Java needs this feature badly!). I recommend that you always write pattern strings with the 'r' just as a habit.

## Basic Patterns

The power of regular expressions is that they can specify patterns, not just fixed characters. Here are the most basic patterns which match single chars:

* a, X, 9, < -- ordinary characters just match themselves exactly. The meta-characters which do not match themselves because they have special meanings are: . ^ $ \* + ? { [ ] \ | ( ) (details below)
* . (a period) -- matches any single character except newline '\n'
* \w -- (lowercase w) matches a "word" character: a letter or digit or underbar [a-zA-Z0-9\_]. Note that although "word" is the mnemonic for this, it only matches a single word char, not a whole word. \W (upper case W) matches any non-word character.
* \b -- boundary between word and non-word
* \s -- (lowercase s) matches a single whitespace character -- space, newline, return, tab, form [ \n\r\t\f]. \S (upper case S) matches any non-whitespace character.
* \t, \n, \r -- tab, newline, return
* \d -- decimal digit [0-9] (some older regex utilities do not support \d, but they all support \w and \s)
* ^ = start, $ = end -- match the start or end of the string
* \ -- inhibit the "specialness" of a character. So, for example, use \. to match a period or \\ to match a slash. If you are unsure if a character has special meaning, such as '@', you can try putting a slash in front of it, \@. If its not a valid escape sequence, like \c, your python program will halt with an error.

## Basic Examples

Joke: what do you call a pig with three eyes? piiig!

The basic rules of regular expression search for a pattern within a string are:

* The search proceeds through the string from start to end, stopping at the first match found
* All of the pattern must be matched, but not all of the string
* If match = re.search(pat, str) is successful, match is not None and in particular match.group() is the matching text

  ## Search for pattern 'iii' in string 'piiig'.  
  ## All of the pattern must match, but it may appear anywhere.  
  ## On success, match.group() is matched text.  
  match = re.search(r'iii', 'piiig') # found, match.group() == "iii"  
  match = re.search(r'igs', 'piiig') # not found, match == None  
  
  ## . = any char but \n  
  match = re.search(r'..g', 'piiig') # found, match.group() == "iig"  
  
  ## \d = digit char, \w = word char  
  match = re.search(r'\d\d\d', 'p123g') # found, match.group() == "123"  
  match = re.search(r'\w\w\w', '@@abcd!!') # found, match.group() == "abc"

## Repetition

Things get more interesting when you use + and \* to specify repetition in the pattern

* + -- 1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's
* \* -- 0 or more occurrences of the pattern to its left
* ? -- match 0 or 1 occurrences of the pattern to its left

### Leftmost & Largest

First the search finds the leftmost match for the pattern, and second it tries to use up as much of the string as possible -- i.e. + and \* go as far as possible (the + and \* are said to be "greedy").

## Repetition Examples

  ## i+ = one or more i's, as many as possible.  
  match = re.search(r'pi+', 'piiig') # found, match.group() == "piii"  
  
  ## Finds the first/leftmost solution, and within it drives the +  
  ## as far as possible (aka 'leftmost and largest').  
  ## In this example, note that it does not get to the second set of i's.  
  match = re.search(r'i+', 'piigiiii') # found, match.group() == "ii"  
  
  ## \s\* = zero or more whitespace chars  
  ## Here look for 3 digits, possibly separated by whitespace.  
  match = re.search(r'\d\s\*\d\s\*\d', 'xx1 2   3xx') # found, match.group() == "1 2   3"  
  match = re.search(r'\d\s\*\d\s\*\d', 'xx12  3xx') # found, match.group() == "12  3"  
  match = re.search(r'\d\s\*\d\s\*\d', 'xx123xx') # found, match.group() == "123"  
  
  ## ^ = matches the start of string, so this fails:  
  match = re.search(r'^b\w+', 'foobar') # not found, match == None  
  ## but without the ^ it succeeds:  
  match = re.search(r'b\w+', 'foobar') # found, match.group() == "bar"

## Emails Example

Suppose you want to find the email address inside the string 'xyz alice-b@google.com purple monkey'. We'll use this as a running example to demonstrate more regular expression features. Here's an attempt using the pattern r'\w+@\w+':

  str = 'purple alice-b@google.com monkey dishwasher'  
  match = re.search(r'\w+@\w+', str)  
  if match:  
    print(match.group())  ## 'b@google'

The search does not get the whole email address in this case because the \w does not match the '-' or '.' in the address. We'll fix this using the regular expression features below.

### Square Brackets

Square brackets can be used to indicate a set of chars, so [abc] matches 'a' or 'b' or 'c'. The codes \w, \s etc. work inside square brackets too with the one exception that dot (.) just means a literal dot. For the emails problem, the square brackets are an easy way to add '.' and '-' to the set of chars which can appear around the @ with the pattern r'[\w.-]+@[\w.-]+' to get the whole email address:

  match = re.search(r'[\w.-]+@[\w.-]+', str)  
  if match:  
    print(match.group())  ## 'alice-b@google.com'

(More square-bracket features) You can also use a dash to indicate a range, so [a-z] matches all lowercase letters. To use a dash without indicating a range, put the dash last, e.g. [abc-]. An up-hat (^) at the start of a square-bracket set inverts it, so [^ab] means any char except 'a' or 'b'.

## Group Extraction

The "group" feature of a regular expression allows you to pick out parts of the matching text. Suppose for the emails problem that we want to extract the username and host separately. To do this, add parenthesis ( ) around the username and host in the pattern, like this: r'([\w.-]+)@([\w.-]+)'. In this case, the parenthesis do not change what the pattern will match, instead they establish logical "groups" inside of the match text. On a successful search, match.group(1) is the match text corresponding to the 1st left parenthesis, and match.group(2) is the text corresponding to the 2nd left parenthesis. The plain match.group() is still the whole match text as usual.

  str = 'purple alice-b@google.com monkey dishwasher'  
  match = re.search(r'([\w.-]+)@([\w.-]+)', str)  
  if match:  
    print(match.group())   ## 'alice-b@google.com' (the whole match)  
    print(match.group(1))  ## 'alice-b' (the username, group 1)  
    print(match.group(2))  ## 'google.com' (the host, group 2)

A common workflow with regular expressions is that you write a pattern for the thing you are looking for, adding parenthesis groups to extract the parts you want.

## findall

findall() is probably the single most powerful function in the re module. Above we used re.search() to find the first match for a pattern. findall() finds \*all\* the matches and returns them as a list of strings, with each string representing one match.

  ## Suppose we have a text with many email addresses  
  str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'  
  
  ## Here re.findall() returns a list of all the found email strings  
  emails = re.findall(r'[\w\.-]+@[\w\.-]+', str) ## ['alice@google.com', 'bob@abc.com']  
  for email in emails:  
    # do something with each found email string  
    print(email)

## findall With Files

For files, you may be in the habit of writing a loop to iterate over the lines of the file, and you could then call findall() on each line. Instead, let findall() do the iteration for you -- much better! Just feed the whole file text into findall() and let it return a list of all the matches in a single step (recall that f.read() returns the whole text of a file in a single string):

  # Open file  
  f = open('test.txt', 'r')  
  # Feed the file text into findall(); it returns a list of all the found strings  
  strings = re.findall(r'some pattern', f.read())

## findall and Groups

The parenthesis ( ) group mechanism can be combined with findall(). If the pattern includes 2 or more parenthesis groups, then instead of returning a list of strings, findall() returns a list of \*tuples\*. Each tuple represents one match of the pattern, and inside the tuple is the group(1), group(2) .. data. So if 2 parenthesis groups are added to the email pattern, then findall() returns a list of tuples, each length 2 containing the username and host, e.g. ('alice', 'google.com').

  str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'  
  tuples = re.findall(r'([\w\.-]+)@([\w\.-]+)', str)  
  print(tuples)  ## [('alice', 'google.com'), ('bob', 'abc.com')]  
  for tuple in tuples:  
    print(tuple[0])  ## username  
    print(tuple[1])  ## host

Once you have the list of tuples, you can loop over it to do some computation for each tuple. If the pattern includes no parenthesis, then findall() returns a list of found strings as in earlier examples. If the pattern includes a single set of parenthesis, then findall() returns a list of strings corresponding to that single group. (Obscure optional feature: Sometimes you have paren ( ) groupings in the pattern, but which you do not want to extract. In that case, write the parens with a ?: at the start, e.g. (?: ) and that left paren will not count as a group result.)

## RE Workflow and Debug

Regular expression patterns pack a lot of meaning into just a few characters , but they are so dense, you can spend a lot of time debugging your patterns. Set up your runtime so you can run a pattern and print what it matches easily, for example by running it on a small test text and printing the result of findall(). If the pattern matches nothing, try weakening the pattern, removing parts of it so you get too many matches. When it's matching nothing, you can't make any progress since there's nothing concrete to look at. Once it's matching too much, then you can work on tightening it up incrementally to hit just what you want.

## Options

The re functions take options to modify the behavior of the pattern match. The option flag is added as an extra argument to the search() or findall() etc., e.g. re.search(pat, str, re.IGNORECASE).

* IGNORECASE -- ignore upper/lowercase differences for matching, so 'a' matches both 'a' and 'A'.
* DOTALL -- allow dot (.) to match newline -- normally it matches anything but newline. This can trip you up -- you think .\* matches everything, but by default it does not go past the end of a line. Note that \s (whitespace) includes newlines, so if you want to match a run of whitespace that may include a newline, you can just use \s\*
* MULTILINE -- Within a string made of many lines, allow ^ and $ to match the start and end of each line. Normally ^/$ would just match the start and end of the whole string.

## Greedy vs. Non-Greedy (optional)

This is optional section which shows a more advanced regular expression technique not needed for the exercises.

Suppose you have text with tags in it: <b>foo</b> and <i>so on</i>

Suppose you are trying to match each tag with the pattern '(<.\*>)' -- what does it match first?

The result is a little surprising, but the greedy aspect of the .\* causes it to match the whole '<b>foo</b> and <i>so on</i>' as one big match. The problem is that the .\* goes as far as is it can, instead of stopping at the first > (aka it is "greedy").

There is an extension to regular expression where you add a ? at the end, such as .\*? or .+?, changing them to be non-greedy. Now they stop as soon as they can. So the pattern '(<.\*?>)' will get just '<b>' as the first match, and '</b>' as the second match, and so on getting each <..> pair in turn. The style is typically that you use a .\*?, and then immediately its right look for some concrete marker (> in this case) that forces the end of the .\*? run.

The \*? extension originated in Perl, and regular expressions that include Perl's extensions are known as Perl Compatible Regular Expressions -- pcre. Python includes pcre support. Many command line utils etc. have a flag where they accept pcre patterns.

An older but widely used technique to code this idea of "all of these chars except stopping at X" uses the square-bracket style. For the above you could write the pattern, but instead of .\* to get all the chars, use [^>]\* which skips over all characters which are not > (the leading ^ "inverts" the square bracket set, so it matches any char not in the brackets).

## Substitution (optional)

The re.sub(pat, replacement, str) function searches for all the instances of pattern in the given string, and replaces them. The replacement string can include '\1', '\2' which refer to the text from group(1), group(2), and so on from the original matching text.

Here's an example which searches for all the email addresses, and changes them to keep the user (\1) but have yo-yo-dyne.com as the host.

## Welcome to NumPy!

NumPy (**Numerical Python**) is an open source Python library that’s used in almost every field of science and engineering. It’s the universal standard for working with numerical data in Python, and it’s at the core of the scientific Python and PyData ecosystems. NumPy users include everyone from beginning coders to experienced researchers doing state-of-the-art scientific and industrial research and development. The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages.

The NumPy library contains multidimensional array and matrix data structures (you’ll find more information about this in later sections). It provides **ndarray**, a homogeneous n-dimensional array object, with methods to efficiently operate on it. NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices and it supplies an enormous library of high-level mathematical functions that operate on these arrays and matrices.

Learn more about [NumPy here](https://numpy.org/doc/stable/user/whatisnumpy.html#whatisnumpy)!

## Installing NumPy

To install NumPy, we strongly recommend using a scientific Python distribution. If you’re looking for the full instructions for installing NumPy on your operating system, see [Installing NumPy](https://numpy.org/install/).

If you already have Python, you can install NumPy with:

conda install numpy

or

pip install numpy

If you don’t have Python yet, you might want to consider using [Anaconda](https://www.anaconda.com/). It’s the easiest way to get started. The good thing about getting this distribution is the fact that you don’t need to worry too much about separately installing NumPy or any of the major packages that you’ll be using for your data analyses, like pandas, Scikit-Learn, etc.

## How to import NumPy

To access NumPy and its functions import it in your Python code like this:

**import** numpy **as** np

We shorten the imported name to np for better readability of code using NumPy. This is a widely adopted convention that you should follow so that anyone working with your code can easily understand it.

## Reading the example code

If you aren’t already comfortable with reading tutorials that contain a lot of code, you might not know how to interpret a code block that looks like this:

>>> a **=** np**.**arange**(6)**

>>> a2 **=** a**[**np**.**newaxis**,** **:]**

>>> a2**.**shape

*(1, 6)*

If you aren’t familiar with this style, it’s very easy to understand. If you see >>>, you’re looking at **input**, or the code that you would enter. Everything that doesn’t have >>> in front of it is **output**, or the results of running your code. This is the style you see when you run python on the command line, but if you’re using IPython, you might see a different style. Note that it is not part of the code and will cause an error if typed or pasted into the Python shell. It can be safely typed or pasted into the IPython shell; the >>> is ignored.

## What’s the difference between a Python list and a NumPy array?

NumPy gives you an enormous range of fast and efficient ways of creating arrays and manipulating numerical data inside them. While a Python list can contain different data types within a single list, all of the elements in a NumPy array should be homogeneous. The mathematical operations that are meant to be performed on arrays would be extremely inefficient if the arrays weren’t homogeneous.

**Why use NumPy?**

NumPy arrays are faster and more compact than Python lists. An array consumes less memory and is convenient to use. NumPy uses much less memory to store data and it provides a mechanism of specifying the data types. This allows the code to be optimized even further.

## What is an array?

An array is a central data structure of the NumPy library. An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. It has a grid of elements that can be indexed in [various ways](https://numpy.org/doc/stable/user/quickstart.html#quickstart-indexing-slicing-and-iterating). The elements are all of the same type, referred to as the array dtype.

An array can be indexed by a tuple of nonnegative integers, by booleans, by another array, or by integers. The rank of the array is the number of dimensions. The shape of the array is a tuple of integers giving the size of the array along each dimension.

One way we can initialize NumPy arrays is from Python lists, using nested lists for two- or higher-dimensional data.

For example:

>>> a **=** np**.**array**([1,** **2,** **3,** **4,** **5,** **6])**

or:

>>> a **=** np**.**array**([[1,** **2,** **3,** **4],** **[5,** **6,** **7,** **8],** **[9,** **10,** **11,** **12]])**

We can access the elements in the array using square brackets. When you’re accessing elements, remember that indexing in NumPy starts at 0. That means that if you want to access the first element in your array, you’ll be accessing element “0”.

>>> print**(**a**[0])**

*[1 2 3 4]*

**How to create a basic array**

*This section covers* np.array(), np.zeros(), np.ones(), np.empty(), np.arange(), np.linspace(), dtype

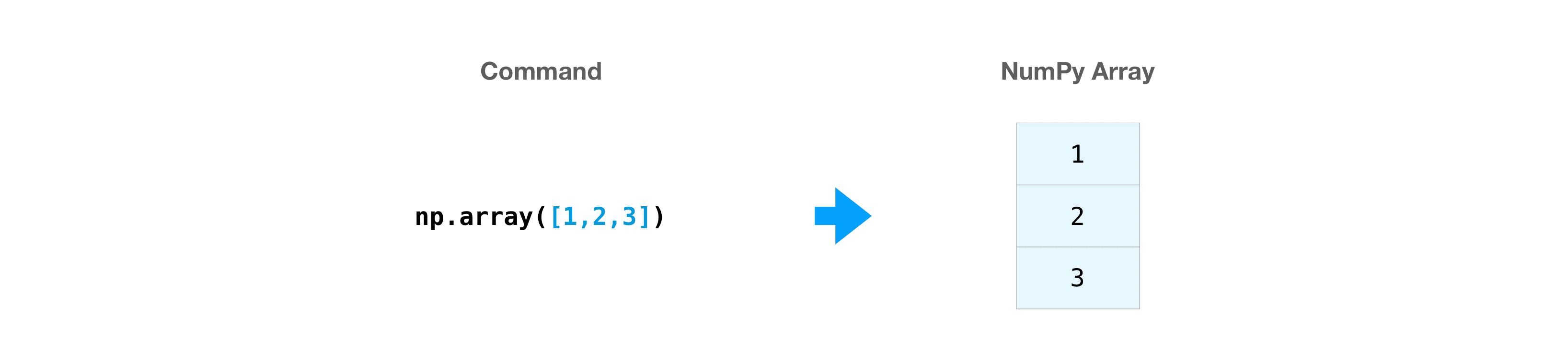
To create a NumPy array, you can use the function np.array().

All you need to do to create a simple array is pass a list to it. If you choose to, you can also specify the type of data in your list. [You can find more information about data types here](https://numpy.org/doc/stable/reference/arrays.dtypes.html#arrays-dtypes).

>>> **import** numpy **as** np

>>> a **=** np**.**array**([1,** **2,** **3])**

You can visualize your array this way:



*Be aware that these visualizations are meant to simplify ideas and give you a basic understanding of NumPy concepts and mechanics. Arrays and array operations are much more complicated than are captured here!*

Besides creating an array from a sequence of elements, you can easily create an array filled with 0’s:

>>> np**.**zeros**(2)**

*array([0., 0.])*

Or an array filled with 1’s:

>>> np**.**ones**(2)**

*array([1., 1.])*

Or even an empty array! The function empty creates an array whose initial content is random and depends on the state of the memory. The reason to use empty over zeros (or something similar) is speed - just make sure to fill every element afterwards!

>>> *# Create an empty array with 2 elements*

>>> np**.**empty**(2)**

*array([3.14, 42. ]) # may vary*

You can create an array with a range of elements:

>>> np**.**arange**(4)**

*array([0, 1, 2, 3])*

And even an array that contains a range of evenly spaced intervals. To do this, you will specify the **first number**, **last number**, and the **step size**.

>>> np**.**arange**(2,** **9,** **2)**

*array([2, 4, 6, 8])*

You can also use np.linspace() to create an array with values that are spaced linearly in a specified interval:

>>> np**.**linspace**(0,** **10,** num**=5)**

*array([ 0. , 2.5, 5. , 7.5, 10. ])*

**Specifying your data type**

While the default data type is floating point (np.float64), you can explicitly specify which data type you want using the dtype keyword.

>>> x **=** np**.**ones**(2,** dtype**=**np**.**int64**)**

>>> x

*array([1, 1])*

**Adding, removing, and sorting elements**

*This section covers* np.sort(), np.concatenate()

Sorting an element is simple with np.sort(). You can specify the axis, kind, and order when you call the function.

If you start with this array:

>>> arr **=** np**.**array**([2,** **1,** **5,** **3,** **7,** **4,** **6,** **8])**

You can quickly sort the numbers in ascending order with:

>>> np**.**sort**(**arr**)**

*array([1, 2, 3, 4, 5, 6, 7, 8])*

In addition to sort, which returns a sorted copy of an array, you can use:

* [**argsort**](https://numpy.org/doc/stable/reference/generated/numpy.argsort.html#numpy.argsort), which is an indirect sort along a specified axis,
* [**lexsort**](https://numpy.org/doc/stable/reference/generated/numpy.lexsort.html#numpy.lexsort), which is an indirect stable sort on multiple keys,
* [**searchsorted**](https://numpy.org/doc/stable/reference/generated/numpy.searchsorted.html#numpy.searchsorted), which will find elements in a sorted array, and
* [**partition**](https://numpy.org/doc/stable/reference/generated/numpy.partition.html#numpy.partition), which is a partial sort.

To read more about sorting an array, see: [**sort**](https://numpy.org/doc/stable/reference/generated/numpy.sort.html#numpy.sort).

If you start with these arrays:

>>> a **=** np**.**array**([1,** **2,** **3,** **4])**

>>> b **=** np**.**array**([5,** **6,** **7,** **8])**

You can concatenate them with np.concatenate().

>>> np**.**concatenate**((**a**,** b**))**

*array([1, 2, 3, 4, 5, 6, 7, 8])*

Or, if you start with these arrays:

>>> x **=** np**.**array**([[1,** **2],** **[3,** **4]])**

>>> y **=** np**.**array**([[5,** **6]])**

You can concatenate them with:

>>> np**.**concatenate**((**x**,** y**),** axis**=0)**

*array([[1, 2],*

*[3, 4],*

*[5, 6]])*

In order to remove elements from an array, it’s simple to use indexing to select the elements that you want to keep.

**Pandas in Python**

Pandas is an open-source library that is made mainly for working with relational or labelled data both easily and intuitively. It provides various data structures and operations for manipulating numerical data and time series. This library is built on top of the NumPy library. Pandas is fast and it has high performance & productivity for users.

### **Advantages of Using Pandas:**

* Fast and efficient for manipulating and analyzing data.
* Data from different file objects can be loaded.
* Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data
* Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects
* Data set merging and joining.
* Flexible reshaping and pivoting of data sets
* Provides time-series functionality.
* Powerful group by functionality for performing split-apply-combine operations on data sets.

## Installing pandas

## The first step of working in pandas is to ensure whether it is installed in the Python folder or not.  If not then we need to install it in our system using****pip command****. Type cmd command in the search box and locate the folder using cd command where ****python-pip file**** has been installed.  After locating it, type the command:

**pip install pandas**

## How to import Pandas

To access pandas and its functions we need to import the library. import it in our Python code like this:

**import pandas as pd**

Here, pd is referred to as an alias to the Pandas. However, it is not necessary to import the library using the alias, it just helps in writing less amount code every time a method or property is called.

Pandas generally provide two data structures for manipulating data, they are:

* **Series**
* **DataFrame**

**Series:**[Pandas Series](https://www.geeksforgeeks.org/python-pandas-series/) is a one-dimensional labelled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called indexes. Pandas Series is nothing but a column in an excel sheet. Labels need not be unique but must be a hash able type. The object supports both integer and label-based indexing and provides a host of methods for performing operations involving the index.

### **Creating an empty Series**

Series() function of Pandas is used to create a series. A basic series, which can be created is an Empty Series.

**Example:**

**import pandas as pd**

**import numpy as np**

**# Creating empty series**

**ser = pd.Series()**

**print(ser)**

**Output:**

**Series([], dtype: float64)**

**\*\*** By default, the data type of Series is float.

**Creating a series from array:**

In order to create a series from NumPy array, we have to import numpy module and have to use array() function.

**#Example**

**# simple array**

**data = np.array(['a', 'r', 'n', 'a', 'b'])**

**ser = pd.Series(data)**

**print(ser)**

**Output:**

**0 a**

**1 r**

**2 n**

**3 a**

**4 b**

**dtype: object**

\*\*By default, the index of the series starts from 0 till the length of series -1.

**Creating a series from array with an index:**

In order to create a series by explicitly proving index instead of the default, we have to provide a list of elements to the index parameter with the same number of elements as it is an array.

**#Example**

**# simple array**

**data = np.array(['a', 'r', 'n', 'a', 'b'])**

**# providing an index**

**ser = pd.Series(data, index=[19, 20, 21, 22, 23])**

**print(ser)**

**Output:**

19 a

20 r

21 n

22 a

23 b

dtype: object

[**Creating a series from Lists**](https://www.geeksforgeeks.org/creating-a-pandas-series-from-lists/)**:**

In order to create a series from list, we have to first create a list after that we can create a series from list.

|  |
| --- |
| **#Example**  **# a simple list**  **list = ['a', 'r', 'n', 'a', 'b']**  **# create series form a list**  **ser = pd.Series(list)**  **print(ser)**  **Output:**  0 a  1 r  2 n  3 a  4 b  dtype: object  [**Creating a series from Dictionary**](https://www.geeksforgeeks.org/creating-a-pandas-series-from-dictionary/)**:**  In order to create a series from the dictionary, we have to first create a dictionary after  that we can make a series using dictionary. Dictionary keys are used to construct indexes  of Series. |

**#Example**

**# a simple dictionary**

**dict = {'Mango': 10,**

**'Apple': 20,**

**'Banana': 30}**

**# create series from dictionary**

**ser = pd.Series(dict)**

[**Creating a series using NumPy functions**](https://www.geeksforgeeks.org/create-pandas-series-using-numpy-functions/)**:**

In order to create a series using numpy function, we can use different function of numpy like [numpy.linspace()](https://www.geeksforgeeks.org/numpy-linspace-python/), [numpy.random.radn()](https://www.geeksforgeeks.org/numpy-random-randn-python/).

**#Example of linspace()**

**# series with numpy linspace()**

**ser1 = pd.Series(np.linspace(3, 33, 3))**

**#Example of random.rand()**

**a=np.random.rand(5,2)**

# **numpy.zeros() in Python:**

**numpy.zeros(shape,dtype=float,order='C')**

# **numpy.unique() in Python:**

**import numpy as np**

**a=np.unique([1,2,3,4,3,6,2,4])**

**Output:**

**array([1, 2, 3, 4, 5])**

# **numpy.concatenate() in Python:**

**#syntax**

**numpy.concatenate((a1, a2, ...), axis)**

**#Example**

**x=np.array([[1,2],[3,4]])**

**y=np.array([[12,30]])**

**z=np.concatenate((x,y))**

### **DataFrame**

[Pandas DataFrame](https://www.geeksforgeeks.org/python-pandas-dataframe/) is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labelled axes (rows and columns). A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the data, rows, and columns.

### **Creating a DataFrame:**

In the real world, a Pandas DataFrame will be created by loading the datasets from existing storage, storage can be SQL Database, CSV file, an Excel file. Pandas DataFrame can be created from the lists, dictionary, and from a list of dictionaries, etc.

**Example:**

**import pandas as pd**

**# Calling DataFrame constructor**

**data = pd.DataFrame()**

**print(data)**

**# list of strings**

**lst = ['he', 'is', 'a', 'good','boy']**

**# Calling DataFrame constructor on list**

**data = pd.DataFrame(lst)**

**print(data)**

# **Pandas DataFrame.append():**

The Pandas **append()** function is used to add the rows of other dataframe to the end of the given dataframe, returning a new dataframe object.

**#Syantax**

**DataFrame.append(other, ignore\_index=False, verify\_integrity=False, sort=None)**

**#Example**

**info1 = pd.DataFrame({"x":[25,15,12,19],**

**"y":[47, 24, 17, 29]})**

**# Create second Dataframe using dictionary**

**Info2 = pd.DataFrame({"x":[25, 15, 12],**

**"y":[47, 24, 17],**

**"z":[38, 12, 45]})**

**# append info2 at end in info1**

**info.append(info2, ignore\_index = True)**

# **Pandas DataFrame.count():**

# The Pandas count() is defined as a method that is used to count the number of non-NA cells for each column or row. It is also suitable to work with the non-floating data.

**#Syntax**

**DataFrame.count(axis=0, level=None, numeric\_only=False)**

**#Example**

**info = pd.DataFrame({"Person":["Parker","Smith", "William", "John"],**

**"Age": [27., 29, np.nan, 32])}**

**info.count()**

# **Pandas DataFrame.describe():**

The describe() method is used for calculating some statistical data like **percentile, mean** and **std** of the numerical values of the Series or DataFrame.

**#Syantax**

**DataFrame.describe(percentiles=None, include=None, exclude=None)**

**#Example**

**a1 = pd.Series([1, 2, 3])**

**a1.describe()**

# **Pandas DataFrame.drop\_duplicates():**

The drop\_duplicates() function performs common data cleaning task that deals with duplicate values in the DataFrame. This method helps in removing duplicate values from the DataFrame.

**#Syntax**

**DataFrame.drop\_duplicates(subset=None, keep='first', inplace=False)**

**#Example**

**emp = {"Name": ["Parker", "Smith", "William", "Parker"],**

**"Age": [21, 32, 29, 21]}**

**info = pd.DataFrame(emp)**

**info = info.drop\_duplicates()**

**print(info)**

# **Pandas DataFrame.groupby():**

In Pandas, **groupby()** function allows us to rearrange the data by utilizing them on real-world data sets.

**#Syntax**

**DataFrame.groupby(by=None, axis=0, level=None, as\_index=True, sort=True, group\_keys=True, squeeze=False, \*\*kwargs)**

**#Example**

**data = {'Name': ['Parker', 'Smith', 'John', 'William'],**

**'Percentage': [82, 98, 91, 87],**

**'Course': ['B.Sc','B.Ed','M.Phill','MCA']}**

**df = pd.DataFrame(data)**

**grouped = df.groupby('Course')**

**print(grouped['Percentage'].agg(np.mean))**

# **Pandas DataFrame.head():**

**#Syntax**

**DataFrame.head(n=5)**

**#Example**

**info = pd.DataFrame({'language':['C', 'C++', 'Python', 'Java','PHP']})**

**info.head()**

**info.head(3)**

# **Pandas DataFrame.join():**

When we want to concatenate our DataFrames, we can add them with each other by stacking them either vertically or side by side. The join() method is often useful when one DataFrame is a lookup table that contains additional data added into the other DataFrame. It is a convenient method that can combine the columns of two differently-indexed DataFrames into a single DataFrame.

# **Pandas DataFrame.merge():**

Pandas **merge()** is defined as the process of bringing the two datasets together into one and aligning the rows based on the common attributes or columns. It is an entry point for all standard database join operations between DataFrame objects:

# **Pandas DataFrame.pivot\_table():**

The Pandas **pivot\_table()** is used to calculate, aggregate, and summarize your data. It is defined as a powerful tool that aggregates data with calculations such as **Sum, Count, Average, Max,** and **Min**.

# **Pandas DataFrame.to\_excel():**

We can export the DataFrame to the excel file by using the to\_excel() function.

**# Example:**

**info\_marks = pd.DataFrame({'name': ['Parker', 'Smith', 'William', 'T erry'],**

**'Maths': [78, 84, 67, 72],**

**'Science': [89, 92, 61, 77],**

**'English': [72, 75, 64, 82]})**

**# render dataframe as html**

**writer = pd.ExcelWriter('output.xlsx')**

**info\_marks.to\_excel(writer)**

**writer.save()**