PYTHON PROGRAMMING LANGUAGE

MODULE 1

INTRODUCTION TO PYTHON PROGRAMMING

Basic Terminologies:

- **Computer Program:** A set of instructions that the computer has to follow to reach the goal.
- **Scripts:** Program that's short, simple, and can be written quickly.
- **Syntax:** Rules for how each instruction is written in programming.
- **Semantics:** The effects that instructions have.
- **Automation:** The process of replacing manual steps with one that happens automatically.

Why Python (Perks of Python):

- Programming in Python usually feels similar to using a human language. This is because Python makes it easy to express what we want to do with syntax that's easy to read and write.
- Python is super popular in the IT industry, making it one of the most common programming languages used today.
- More tools available in Python for a growing range of applications.

Important Python Versions:

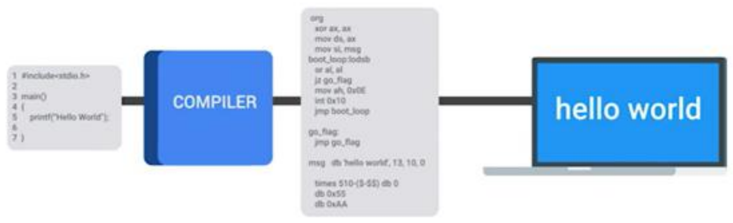
Its first version was released by **Guido van Rossum** back in **1991**.

- Python 2 (2002)
- Python 3 (2008)
- Python 3.7 (2018)

Compiled vs Interpreted Languages:

1. Compiled Language:

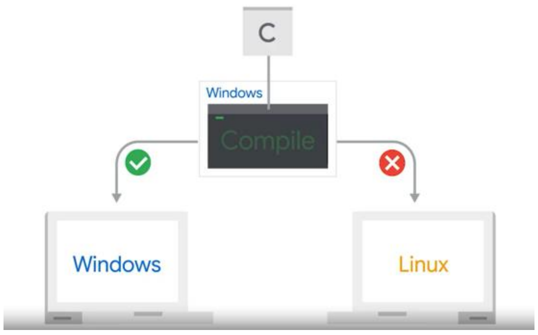
    - Source code is fed into the compiler.
    - Compiler translates the code to machine level language.
    - The computer can read and execute the machine-level code directly.
    - This makes compiled program super efficient to run
    - Examples: C, C++, Go and Rust.

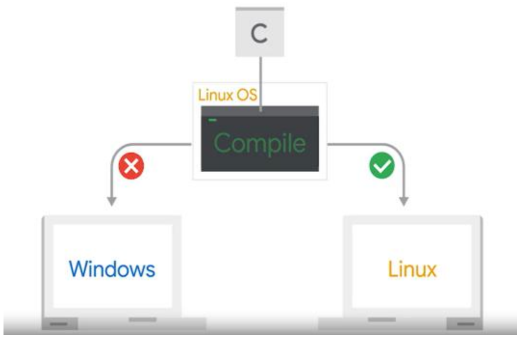


```
#include<stdio.h>

main()
{
  printf("Hello World");

}
```

```
org
  xor ax, ax
  mov ds, ax
  mov si, msg
boot_loop:lodsb
  or al, al
  jz go_flag
  mov ah, 0x0E
  int 0x10
  jmp boot_loop

go_flag:
  jmp go_flag

msg   db 'hello world', 13, 10, 0

  times 510-($-$$) db 0
  db 0x55
  db 0xAA
```

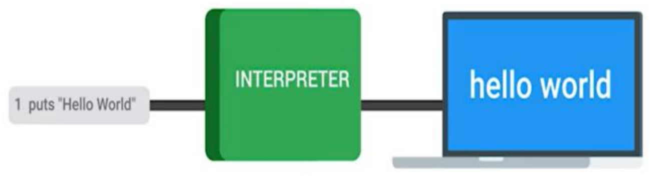
Note that for compiled languages, the same code cannot be used on different platforms as different platforms have different compilers.



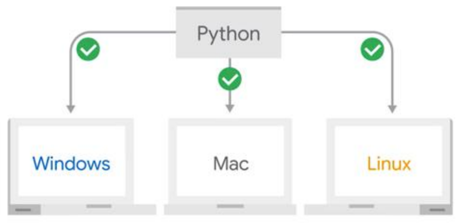


2. <u>Interpreted Languages:</u>

- Execute the program line by line using interpreters.
- Python interpreter is the program that reads what is in the recipe and translates it into instructions for your computer to follow.
- The same code can be read by interpreters running on different operating systems without needing to make any additional changes.



```
puts "Hello World"
```



3. <u>Mixed Approach Languages:</u>

Here, the source code gets complied into an intermediate code and then can be interpreted on different platforms. Ex: Java and C#.

In Java, the java compiler first translates the source code into intermediate code called Bytecode. This Bytecode is platform independent and is then interpreted.

PyPI: -

The Python Package Index is the repository of the software for python programming language. It helps to find and install the software developed and shared by the Python community.

PIP: (Preferred installer program)

Pip Install Packages is a command line tool used as the main method of installing and managing python packages:

To install a package, use **pip install package** (for windows)

!pip install package (for jupyter notebook)

sudo apt install python3-pip module_name (for Linux)

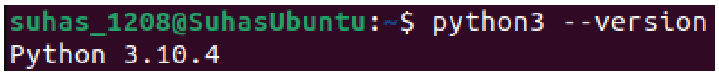
To uninstall a package, use **pip uninstall package**

USING PYTHON ON LINUX

Linux Commands for python modules:

- Checking the version of python installed:

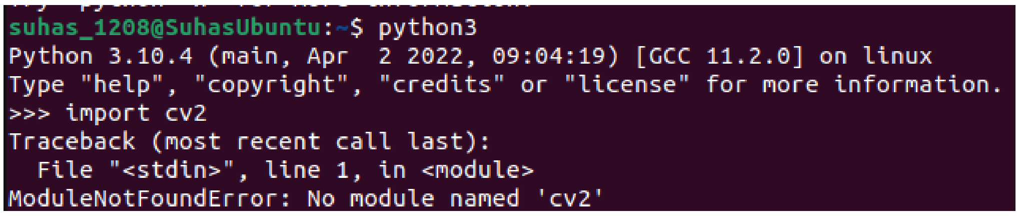
python3 --version



```
suhas_1208@SuhasUbuntu:~$ python3 --version
Python 3.10.4
```

- Checking whether a module is present: (when module is not present)

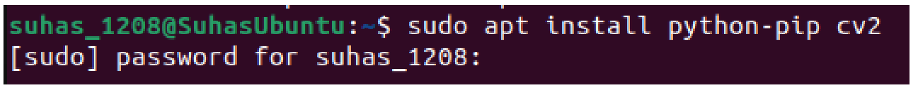
import module_name



```
suhas_1208@SuhasUbuntu:~$ python3
Python 3.10.4 (main, Apr  2 2022, 09:04:19) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cv2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'cv2'
```

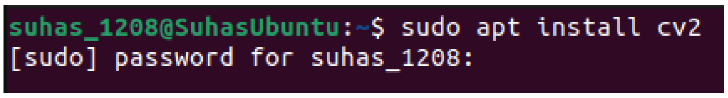
- Installing a python module:

sudo apt install python3-pip module_name



```
suhas_1208@SuhasUbuntu:~$ sudo apt install python-pip cv2
[sudo] password for suhas_1208:
```

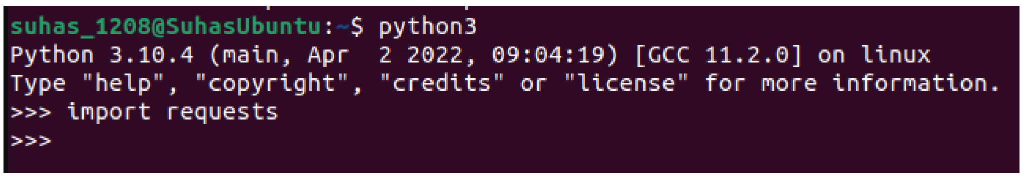
sudo apt install module_name



```
suhas_1208@SuhasUbuntu:~$ sudo apt install cv2
[sudo] password for suhas_1208:
```

- Then checking if module is installed:

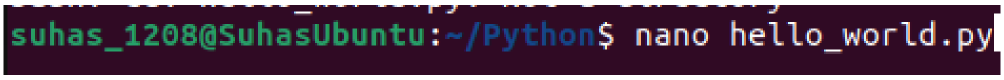
import module_name



```
suhas_1208@SuhasUbuntu:~$ python3
Python 3.10.4 (main, Apr  2 2022, 09:04:19) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>>
```

Running Python on Terminals :

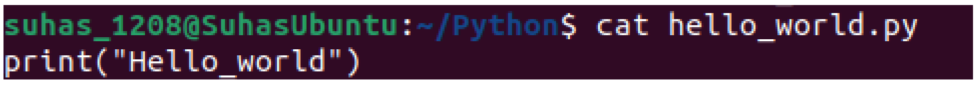
- Go to the directory in which the python file is present and use **editor_name file_name.py** to open the code in editor.



```
suhas_1208@SuhasUbuntu:~/Python$ nano hello_world.py
```

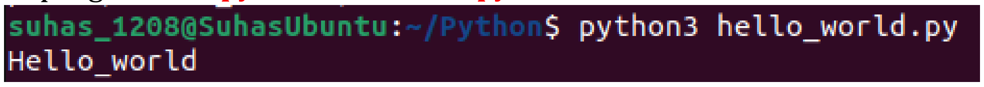
- Write the source code.

- To see the contents of the file, use **cat** command.



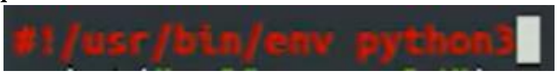
```
suhas_1208@SuhasUbuntu:~/Python$ cat hello_world.py
print("Hello_world")
```

- To run the program use **python3 file_name.py** in the terminal.



```
suhas_1208@SuhasUbuntu:~/Python$ python3 hello_world.py
Hello_world
```

- We can use the **shebang** line in our script which tells the OS what command we want to use for executing that file. After adding shebang line you can use **./file_name.py** command to run the required file.



```
#!/usr/bin/env python3
```

MODULE 2

BASICS OF PYTHON

Keywords:

Keywords are reserved words that are used to construct instructions.

These are some key words:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Data Types:

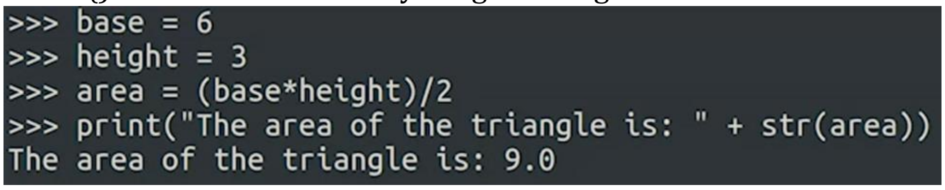
- **String:** In Python, text in between quotes -- either single or double quotes -- is a string data type. Ex: "hello", '2', "3".
- **Integer:** Ex: -1, 0,2, 3, 4.
- **Float:** For fractional part. Ex: 3.14, 6.0.

➢ *You can always check the data type of something using the type() function.*

➢ Type Conversion:

1) Implicit Conversion:
When the interpreter automatically converts one data type to another.
Ex: *print(7+8.5)* gives the output as *15.5* because the interpreter automatically converts the *int 7* into *float 7.*

2) Explicit Conversion:
When we convert one data type to another data type manually using a function.
Ex: *str()* function converts anything to string.



```
>>> base = 6
>>> height = 3
>>> area = (base*height)/2
>>> print("The area of the triangle is: " + str(area))
The area of the triangle is: 9.0
```

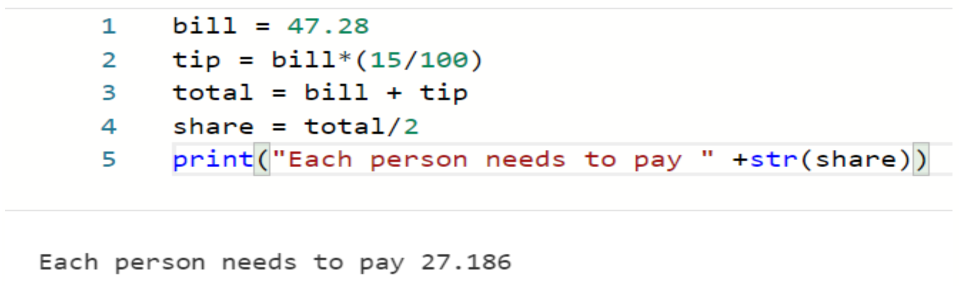
Variables:

Variables are the containers in which we store certain values.

- Assignment: The process of storing value inside variables.
- Expression**:** Combination of numbers, symbols, or other variables that produce a result when evaluated.
- Syntax for naming variable**:** Variable_name= value
- Restrictions for naming variables:
  - Do not use keywords or function names that python reserves for it's own.
  - Variable names must only contain numbers, letters and underscore.
  - Variable names cannot contain space.
  - Variable name can only begin with letter or underscore but not numbers.
  - Variables are case sensitive.

Question:
In this scenario, two friends are eating dinner at a restaurant. The bill comes in the amount of 47.28 dollars. The friends decide to split the bill evenly between them, after adding 15% tip for the service. Calculate the tip, the total amount to pay, and each friend's share, then output a message saying "Each person needs to pay: " followed by the resulting number.

Sol:



```python
bill = 47.28
tip = bill*(15/100)
total = bill + tip
share = total/2
print("Each person needs to pay " +str(share))
```

```
Each person needs to pay 27.186
```

OPERATORS

An operation consists of the following components.

1) Operator: Symbols that carry out operation.
2) Operands: Values on which we carry out operations.

Arithmetic Operators:

- **a + b** = Adds a and b
- **a - b** = Subtracts b from a
- **a * b** = Multiplies a and b
- **a / b** = Divides a by b (Division always results in float)
- **a ** b** = Elevates a to the power of b. For non-integer values of b, this becomes a root (i.e. a**(1/2) is the square root of a)
- **a // b** = The integer part of the integer division of a by b
- **a % b** = The remainder part of the integer division of a by b *(modulo operator).*

Comparison Operators:

- a == b: a is equal to b
- a != b: a is different than b
- a < b: a is smaller than b
- a <= b: a is smaller or equal to b
- a > b: a is bigger than b
- a >= b: a is bigger or equal to b

< and > cannot be operated between a string and an integer

A< B< C< D.........

a < b < c < d......

a>A

Note: In Python uppercase letters are alphabetically sorted before lowercase letters.

Ex: "cat'' > "Cat".

Logical Operators:

1. **and**: To evaluate as *true*, the *and* operator needs both expressions to be *true* at the same time.
2. **or:** To evaluate as *false*, the *or* operator needs both expressions to be *false* at the same time.
3. **not:** The *not* operator *inverts* the value of expression that is in front of it.

MODULE 3

CONTROL STRUCTURES AND FUNCTIONS

CONDITIONALS

If Block:

The program enters the code present inside the if block only when the condition of the if statement is true. Else it skips the if block and starts executing the subsequent lines.

If-Else Block:

The program checks for the condition in if statement.

- If the condition is true then code inside if block is executed.
- If the condition is false then code inside else block is executed.

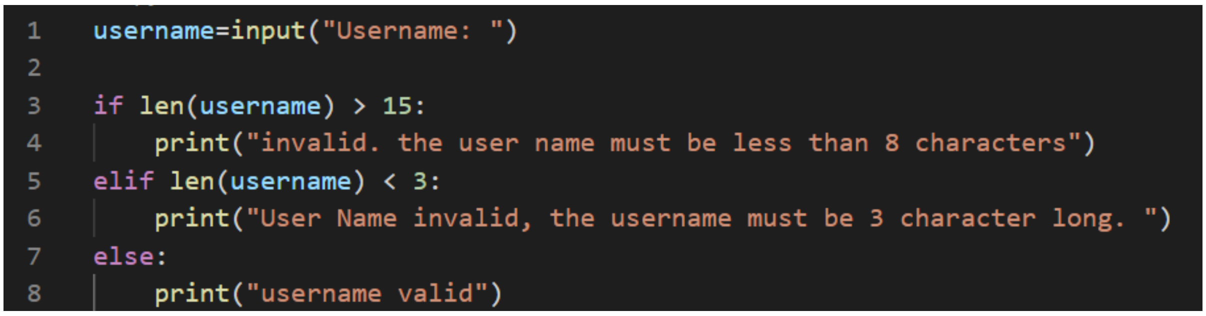
If-Elif-Else Block:

The program checks for condition in if statement.

- If the condition in if statement is true, code inside if block is executed.
- If the condition is false, then condition in elif block is checked and executed.
- If conditions in all the elif blocks are false, then else block is executed.

if condition:
    body
	if condition:
	    body
	
	else:
	    body
		if condition:
		    body
		
		elif condition:
		    body
		
		else:
		    body

Ex: Write a program which says to enter a user name which ranges between 3-15 characters.



```python
username=input("Username: ")

if len(username) > 15:
    print("invalid. the user name must be less than 8 characters")
elif len(username) < 3:
    print("User Name invalid, the username must be 3 character long. ")
else:
    print("username valid")
```

LOOPS

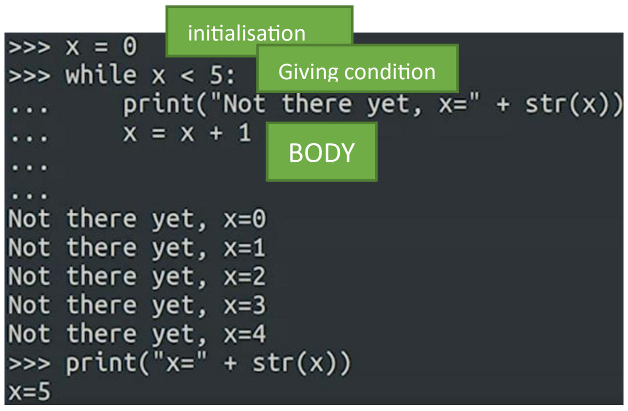
WHILE LOOP:

A while loop executes the body of the loop while the condition remains True.

Anatomy of while loop:

- A *while* loop will continuously execute code depending on the value of a condition.
- It begins with the keyword *while,* followed by a comparison to be evaluated, then a colon.
- On the next line is the code block to be executed, indented to the right.
- Similar to an *if* statement, the code in the body will only be executed if the comparison is evaluated to be true.
- What sets a *while* loop apart, however, is that this code block will keep executing as long as the evaluation statement is true.
- Once the statement is no longer true, the loop exits and the next line of code will be executed.

Ex:



```
>>> x = 0
>>> while x < 5:
...     print("Not there yet, x=" + str(x))
...     x = x + 1
...
...
Not there yet, x=0
Not there yet, x=1
Not there yet, x=2
Not there yet, x=3
Not there yet, x=4
>>> print("x=" + str(x))
x=5
```

Typical use:
While loops are mostly used when there's an unknown number of operations to be performed, and a condition needs to be checked at each iteration.

Note:

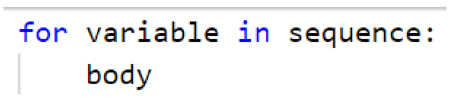
Whenever writing a loop, check that you are initialising all the variables you want to use before them.

Infinite loops:
Loops that keep on executing and never stop.

FOR LOOP:

A for loop iterates over a sequence of elements, executing the body of the loop for each element in the sequence.

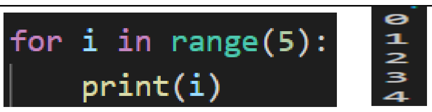
Syntax:

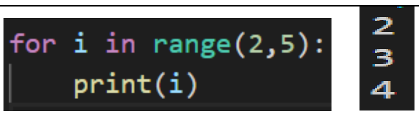


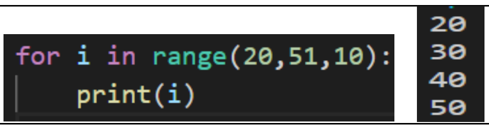
```
for variable in sequence:
    body
```

The range() function:

range() generates a sequence of integer numbers. It can take one, two, or three parameters:

range(n): 0, 1, 2, ... n-1	 `for i in range(5):`
	 `    print(i)`
	 `0`
	 `1`
	 `2`
	 `3`
	 `4`

range(x, y): x, x+1, x+2, ... y-1	 `for i in range(2,5):`
	 `    print(i)`
	 `2`
	 `3`
	 `4`

range(p, q ,r): p, p+r, p+2r, p+3r, ... q-1 (if it's a valid increment)	 `for i in range(20,51,10):`
	 `    print(i)`
	 `20`
	 `30`
	 `40`
	 `50`

Common pitfalls:

- Forgetting that the upper limit of a range() isn't included.
- Iterating over non-sequences. Integer numbers aren't iterable. Strings are iterable letter by letter, but that might not be what you want.

Typical use:
For loops are mostly used when there's a pre-defined sequence or range of numbers to iterate.

BREAK AND CONTINUE:

You can interrupt both while and for loops using the break keyword. We normally do this to interrupt a cycle due to a separate condition.

You can use the continue keyword to skip the current iteration and continue with the next one. This is typically used to jump ahead when some of the elements of the sequence aren't relevant.

FUNCTIONS

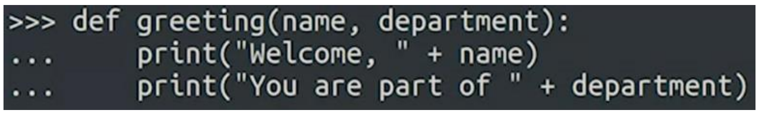
Defining a Function:

Syntax for defining function: **def function_name (parameters):**

body

- After the name, we have the parameters, also called arguments, for the function enclosed in parentheses (). A function can have no parameters, or it can have multiple parameters.
- Lastly, we put a colon at the end of the line.
- After the colon, the function body starts.

Ex:

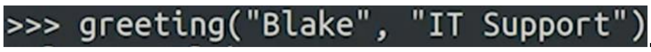


```
>>> def greeting(name, department):
...     print("Welcome, " + name)
...     print("You are part of " + department)
```

Here a function named greeting is defined which has two parameters name and department.

Note that body of function (here the next two lines) must be to the right of the definition.

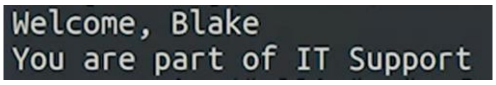
Input:-



```
>>> greeting("Blake", "IT Support")
```

.

Output:-



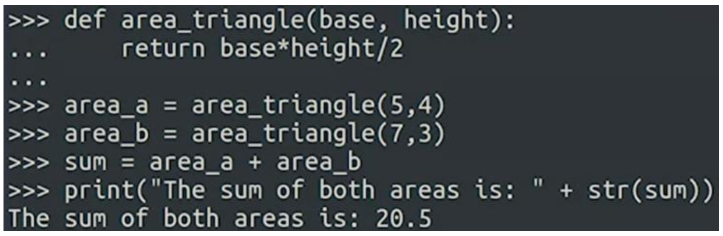
```
Welcome, Blake
You are part of IT Support
```

Returning the values:

Sometimes we don't want a function to simply run and finish. We may want a function to manipulate data we passed it and then return the result to us. This is where the concept of return values comes in handy. We use the return keyword in a function, which tells the function to pass data back. When we call the function, we can store the returned value in a variable. Return values allow our functions to be more flexible and powerful, so they can be reused and called multiple times.

Functions can even return multiple values. Just don't forget to store all returned values in variables! You could also have a function return nothing, in which case the function simply exits.

Ex:



```
>>> def area_triangle(base, height):
...     return base*height/2
...
>>> area_a = area_triangle(5,4)
>>> area_b = area_triangle(7,3)
>>> sum = area_a + area_b
>>> print("The sum of both areas is: " + str(sum))
The sum of both areas is: 20.5
```

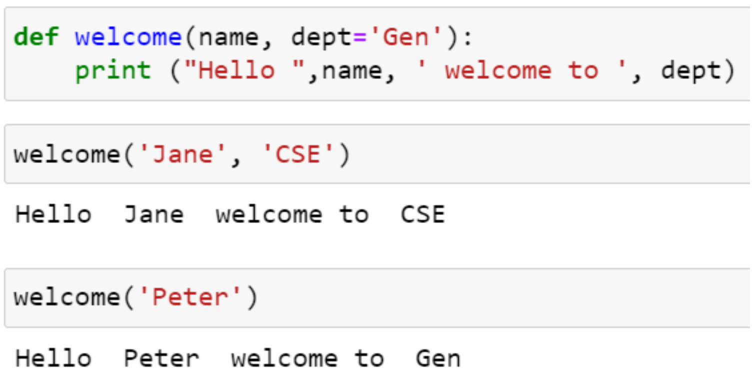
Default Parameters:

You can have the default value for a parameter if no arguments are given by the user to that parameter.

def func (parameter=default):

body

Ex:



```
def welcome(name, dept='Gen'):
    print ("Hello ",name, ' welcome to ', dept)
```

```
welcome('Jane', 'CSE')
```

```
Hello  Jane  welcome to  CSE
```

```
welcome('Peter')
```

```
Hello  Peter  welcome to  Gen
```

MODULE 4

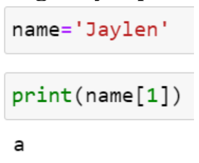
PYTHON DATA STRUCTURES

STRINGS

- A string is a data type in Python that's used to represent a piece of text.
- Strings are sequences of characters and are immutable.
- It's written between quotes, either double quotes or single quotes.

Indexing:

Selecting only a portion of string.



```
name='Jaylen'
```

```
print(name[1])
```

```
a
```

To select from last use negative indexes.

String operations:

- **len(string)** Returns the length of the string
- **for character in string** Iterates over each character in the string
- **if substring in string** Checks whether the substring is part of the string

Slicing:

A slice is the portion of a string that can contain more than one character, also sometimes called a substring.

This allows you to access multiple characters of a string. You can do this by creating a range, using a colon as a separator between the start and end of the range, like [2:5]. This range is similar to the range() function we saw previously. It includes the first number, but goes to one less than the last number. For example:

- **>>> fruit = "Mangosteen" >>> fruit[1:4] 'ang'**

  The slice *includes* the character at index 1, and *excludes* the character at index 4. You can also easily reference a substring at the start or end of the string by only specifying one end of the range. For example, only giving the end of the range.

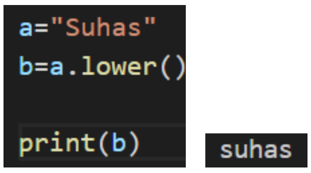
- **>>> fruit[:5] 'Mango'**

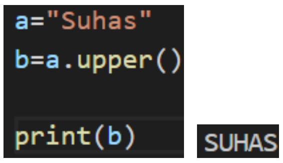
  This gave us the characters from the start of the string through index 4, *excluding* index 5. On the other hand this example gives is the characters *including* index 5, through the end of the string:

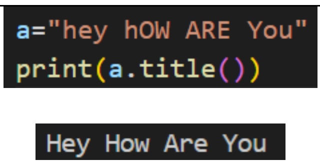
- **>>> fruit[5:] 'steen'**

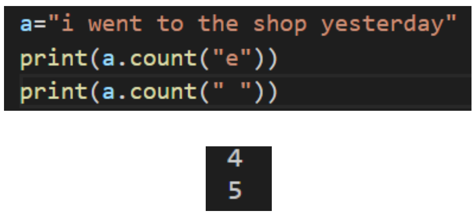
- You might have noticed that if you put both of those results together, you get the original string back!
  >>> fruit[:5] + fruit[5:] 'Mangosteen'

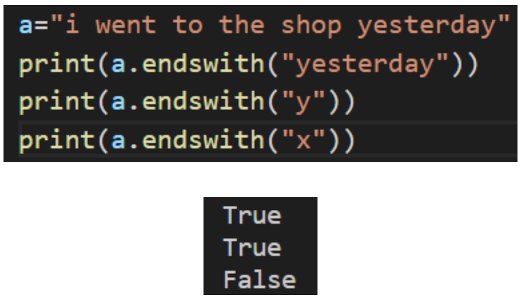
Methods in Strings:

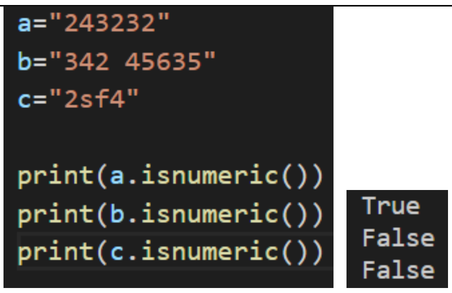
string.lower()
 	
to make all characters of string small
	Returns copy of string with all lower characters.	
		`a="Suhas"`
		`b=a.lower()`
		`print(b)`
		`suhas`

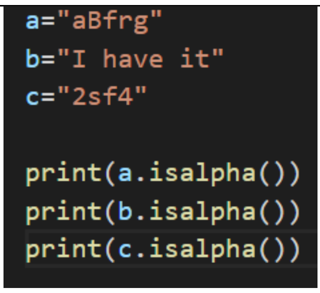
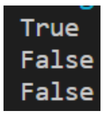
string.upper()
 	
to make all characters of string capital
	Returns copy of string with all upper characters.	
		`a="Suhas"`
		`b=a.upper()`
		`print(b)`
		`SUHAS`

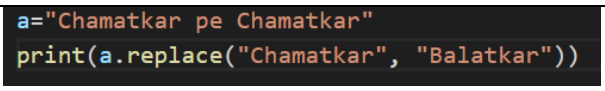
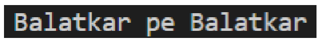
string.title()	Returns each word with first letter as capital.	
		`a="hey hOW ARE You"`
		`print(a.title())`
		`Hey How Are You`

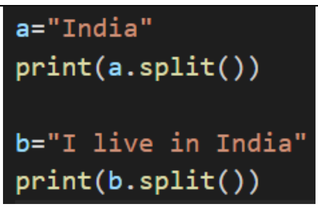
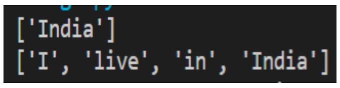
string.count(substring)
 	
to check if a substring is present in a string
	Returns the number of times substring is present in the string.	
		`a="i went to the shop yesterday"`
		`print(a.count("e"))`
		`print(a.count(" "))`
		`4`
		`5`

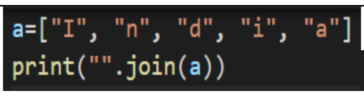
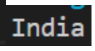
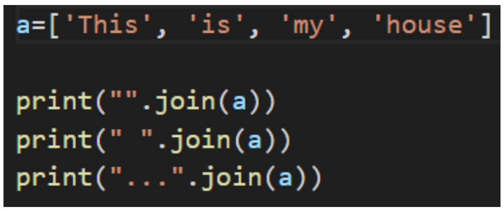
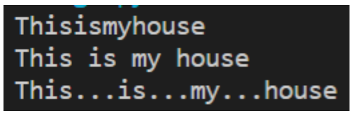
string.endswith(substring)
 	
to check if string ends with a substring
	Returns true if the string ends with substring.	
		`a="i went to the shop yesterday"`
		`print(a.endswith("yesterday"))`
		`print(a.endswith("y"))`
		`print(a.endswith("x"))`
		`True`
		`True`
		`False`

string.isnumeric()
 	
to check it string contains only numbers
	Returns True if there are only numeric characters in the string.	
		`a="243232"`
		`b="342 45635"`
		`c="2sf4"`
		
		`print(a.isnumeric())`
		`print(b.isnumeric())`
		`print(c.isnumeric())`
		`True`
		`False`
		`False`

string.isalpha()
 	
to check if the string contains only alphabets
	Returns True if there are only alphabetic characters in the string.	
		`a="aBfrg"`
		`b="I have it"`
		`c="2sf4"`
		
		`print(a.isalpha())`
		`print(b.isalpha())`
		`print(c.isalpha())`
		
		`True`
		`False`
		`False`

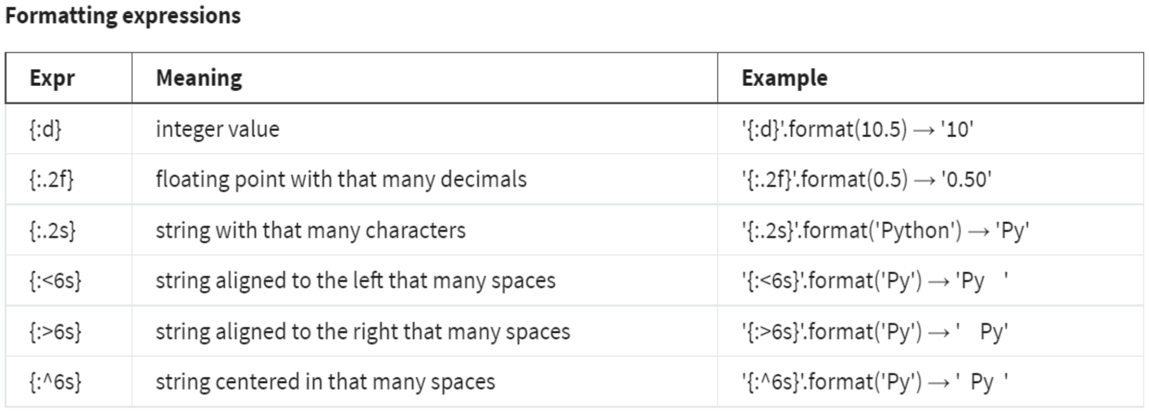
string.replace(old, new)
 	
to replace something in a string with something else
	Returns a new string where all occurrences of old have been replaced by new.	
		`a="Chamatkar pe Chamatkar"`
		`print(a.replace("Chamatkar", "Balatkar"))`
		
		`Balatkar pe Balatkar`

string.split()
 	
to split the words of string into list of words
 	
string.split(delimiter)
	Returns a list of substrings that were separated by whitespace.
	
	Returns a list of substrings that were separated by delimiter.
		
		`a="India"`
		`print(a.split())`
		
		`b="I live in India"`
		`print(b.split())`
		
		`['India']`
		`['I', 'live', 'in', 'India']`

delimiter.join(list of strings)
 	
to join the strings in a list using delimiter
	Returns a new string with all the strings joined by the delimiter.	
		`a=["I", "n", "d", "i", "a"]`
		`print("".join(a))`
		
		`India`
		
		`a=['This', 'is', 'my', 'house']`
		
		`print("".join(a))`
		`print(" ".join(a))`
		`print("...".join(a))`
		
		`Thisismyhouse`
		`This is my house`
		`This...is...my...house`

string.isupper()	Returns True if the specified string is in upper case.	
string.islower()	Returns true if the specified string is in lower case.	
reversed(string)	Returns the reversed string.	

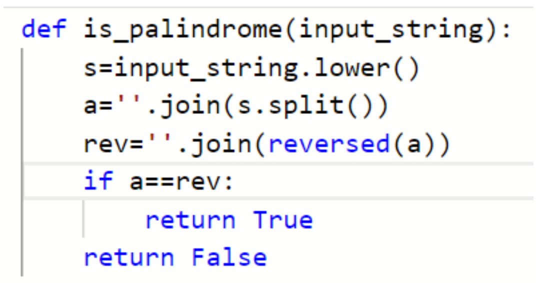
String Formatting:



Formatting expressions

Expr	Meaning	Example
{:d}	integer value	'{:d}'.format(10.5) → '10'
{:.2f}	floating point with that many decimals	'{:.2f}'.format(0.5) → '0.50'
{:.2s}	string with that many characters	'{:.2s}'.format('Python') → 'Py'
{:<6s}	string aligned to the left that many spaces	'{:<6s}'.format('Py') → 'Py '
{:>6s}	string aligned to the right that many spaces	'{:>6s}'.format('Py') → ' Py'
{:^6s}	string centered in that many spaces	'{:^6s}'.format('Py') → ' Py '

Checking palindrome:



```
def is_palindrome(input_string):
    s=input_string.lower()
    a=''.join(s.split())
    rev=''.join(reversed(a))
    if a==rev:
        return True
    return False
```

LISTS

- Lists are sequences of elements of any type.
- Lists in Python are defined using square brackets, with the elements stored in the list separated by commas.

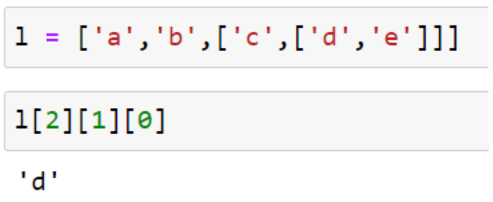
  list_name = [element 1, element 2, ….]
- Lists are mutable, it means that we can remove, add, modify(change) the elements of the list.

Operations on lists:

1) *len(list)* **-** Returns number of elements in the list.
2) *for element in list* **-** Iterates over each element in the list.
3) *element in list* **-** Returns true if the element is present in list.
4) Indexing and Slicing.

Nested Lists:

We can have a list inside a list and we can access the elements using multi-dimensional indexing.

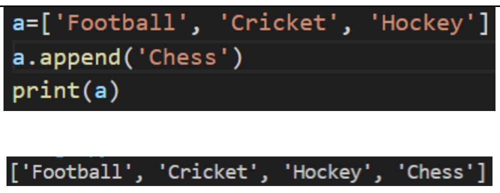
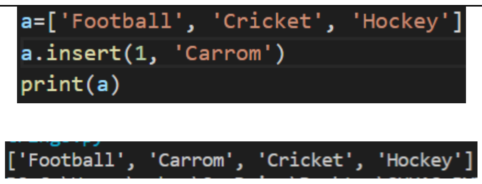
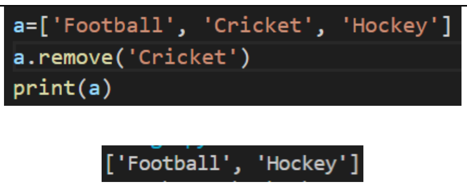


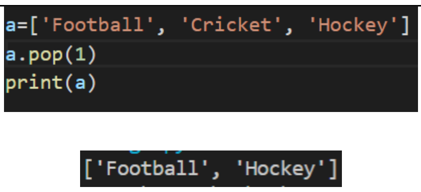
```
l = ['a','b',['c',['d','e']]]
```

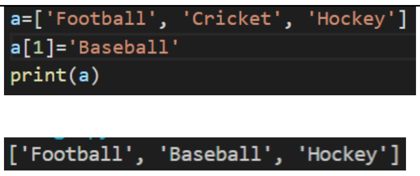
```
l[2][1][0]
```

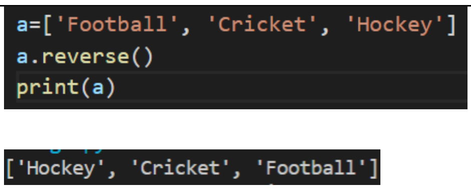
```
'd'
```

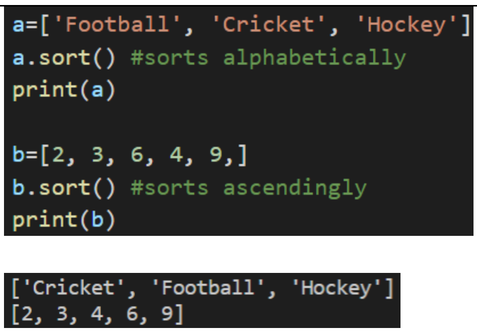
Methods of List Modification:

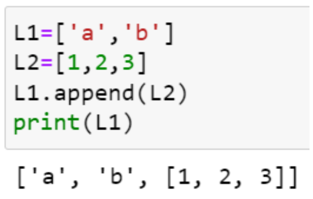
list.append(element)	Adds element to the end of the list.	 `a=['Football', 'Cricket', 'Hockey']` `a.append('Chess')` `print(a)` `['Football', 'Cricket', 'Hockey', 'Chess']`
list.insert(index, element)	Adds element at the given index.	 `a=['Football', 'Cricket', 'Hockey']` `a.insert(1, 'Carrom')` `print(a)` `['Football', 'Carrom', 'Cricket', 'Hockey']`
list.remove(element)	Removes the first occurrence of element from the list.	 `a=['Football', 'Cricket', 'Hockey']` `a.remove('Cricket')` `print(a)` `['Football', 'Hockey']`

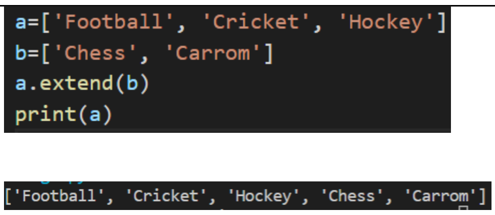
list.pop(index)	Removes element of the given index from the list.	 `a=['Football', 'Cricket', 'Hockey']`
		`a.pop(1)`
		`print(a)`
		
		`['Football', 'Hockey']`

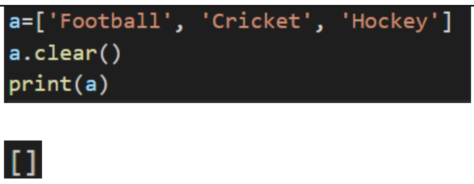
list[index]=new_element	Replaces the old element at given index with new element.	 `a=['Football', 'Cricket', 'Hockey']`
		`a[1]='Baseball'`
		`print(a)`
		
		`['Football', 'Baseball', 'Hockey']`

list.reverse()	Reverses the order of elements in the list.	 `a=['Football', 'Cricket', 'Hockey']`
		`a.reverse()`
		`print(a)`
		
		`['Hockey', 'Cricket', 'Football']`

list.sort()	Sorts the elements in list.	 `a=['Football', 'Cricket', 'Hockey']`
		`a.sort() #sorts alphabetically`
		`print(a)`
		
		`b=[2, 3, 6, 4, 9,]`
		`b.sort() #sorts ascendingly`
		`print(b)`
		
		`['Cricket', 'Football', 'Hockey']`
		`[2, 3, 4, 6, 9]`

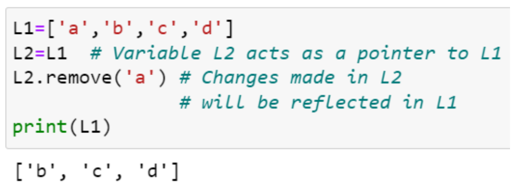
list.append(other_list)	Appends the other list as an element to the list.	 `L1=['a','b']`
		`L2=[1,2,3]`
		`L1.append(L2)`
		`print(L1)`
		
		`['a', 'b', [1, 2, 3]]`

list.extend(other_list)	Appends all the elements of other list at the end of list.	 `a=['Football', 'Cricket', 'Hockey']`
		`b=['Chess', 'Carrom']`
		`a.extend(b)`
		`print(a)`
		
		`['Football', 'Cricket', 'Hockey', 'Chess', 'Carrom']`

list.clear()	Removes all the elements from the list.	 `a=['Football', 'Cricket', 'Hockey']`
		`a.clear()`
		`print(a)`
		
		`[]`

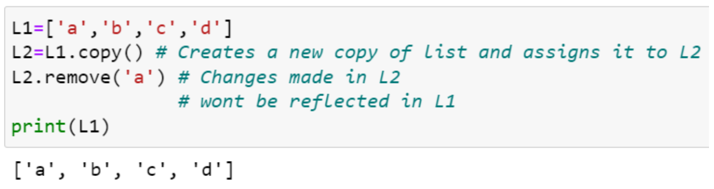
list.copy()	Creates a copy of the list.	
list.count(element)	Returns the number of times the element is present in the list	

Importance of .copy() method:



```
L1=['a','b','c','d']
L2=L1  # Variable L2 acts as a pointer to L1
L2.remove('a') # Changes made in L2
               # will be reflected in L1
print(L1)
```

```
['b', 'c', 'd']
```



```
L1=['a','b','c','d']
L2=L1.copy() # Creates a new copy of list and assigns it to L2
L2.remove('a') # Changes made in L2
               # wont be reflected in L1
print(L1)
```

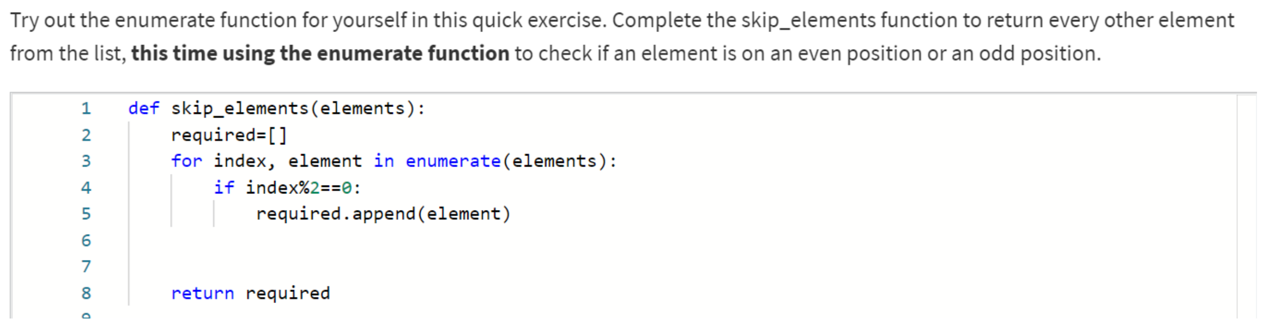
```
['a', 'b', 'c', 'd']
```

Iterating Over Lists Using Enumerate

If you want to access the elements in a list, along with the index of the element at the same time, it can be done using the **enumerate( )** function.

The enumerate() function takes a list as a parameter and returns a tuple for each element in the list. The first value of the tuple is the index and the second value is the element itself.

for index, element in enumerate(sequence)

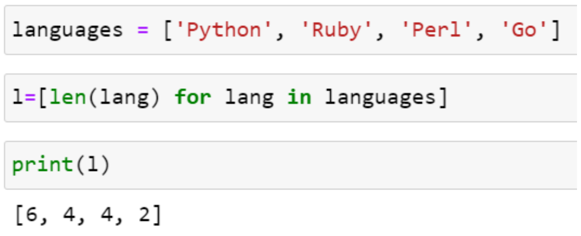


Try out the enumerate function for yourself in this quick exercise. Complete the skip_elements function to return every other element from the list, **this time using the enumerate function** to check if an element is on an even position or an odd position.

```
def skip_elements(elements):
    required=[]
    for index, element in enumerate(elements):
        if index%2==0:
            required.append(element)


    return required
```

List comprehension

- *[expression for variable in sequence]* Creates a new list based on the given sequence. Each element is the result of the given expression.



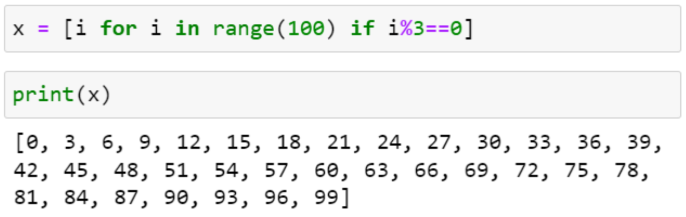
```
languages = ['Python', 'Ruby', 'Perl', 'Go']
```

```
l=[len(lang) for lang in languages]
```

```
print(l)
```

```
[6, 4, 4, 2]
```

- *[expression for variable in sequence if condition]* Creates a new list based on the given sequence. Each element is the result of the given expression; elements only get added if the condition is true.
  Ex: Multiples of 3 from 0 to 100.



```
x = [i for i in range(100) if i%3==0]
```

```
print(x)
```

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39,
42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78,
81, 84, 87, 90, 93, 96, 99]
```

TUPLES

- Tuples are like lists since they are sequences of elements of any data type.
- But unlike lists, tuples are immutable.
- Tuples are specified using parentheses.

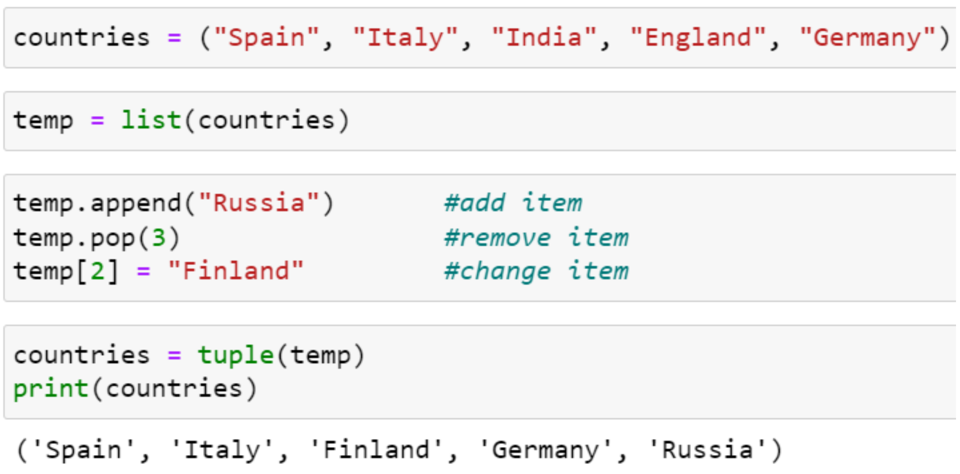
tup_name = (element 1, element 2, ….)

Operations on Tuples:

1. *len(tuple)* – Returns the number of elements in the tuple.
2. *element in tuple* – Returns true if the element is present in tuple.
3. *for element in tuple* – Iterates over all the elements of tuple.
4. Indexing and slicing.

Manipulating tuples:

Tuples are immutable. To make any changes in a tuple, you have to first convert it to a list, then make the necessary changes, and then convert the list back to a tuple.



```
countries = ("Spain", "Italy", "India", "England", "Germany")
```

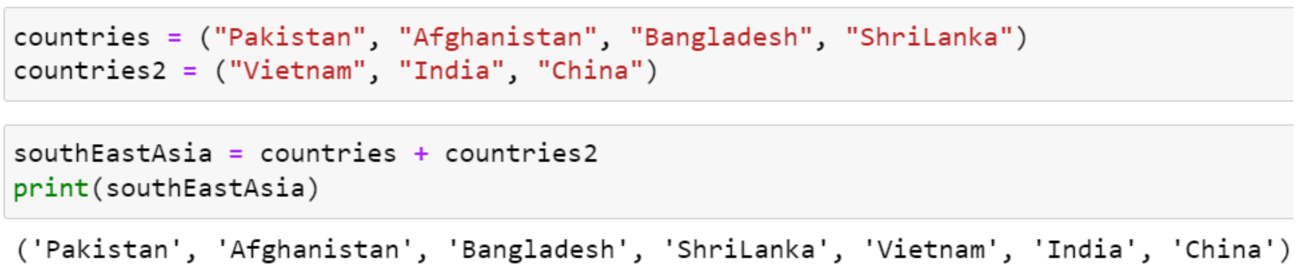
```
temp = list(countries)
```

```
temp.append("Russia")        #add item
temp.pop(3)                  #remove item
temp[2] = "Finland"          #change item
```

```
countries = tuple(temp)
print(countries)
```

```
('Spain', 'Italy', 'Finland', 'Germany', 'Russia')
```

- However, you can concatenate two tuples without converting them to lists using + .



```
countries = ("Pakistan", "Afghanistan", "Bangladesh", "ShriLanka")
countries2 = ("Vietnam", "India", "China")
```

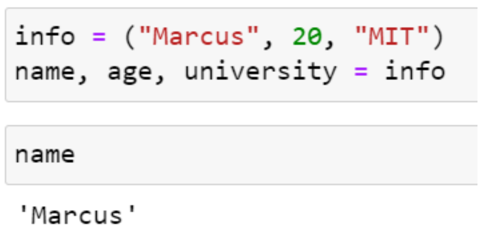
```
southEastAsia = countries + countries2
print(southEastAsia)
```

```
('Pakistan', 'Afghanistan', 'Bangladesh', 'ShriLanka', 'Vietnam', 'India', 'China')
```

Unpacking Tuples:

Loading the elements of tuples into different variables.

var 1, var 2, ……, var n = (ele 1, ele 2, ……, ele n)



```
info = ("Marcus", 20, "MIT")
name, age, university = info
```

```
name
```

```
'Marcus'
```

Why Tuples:

Tuples can be useful when we need to ensure that an element is in a certain position and will not change.

- Lists are mutable, the order of the elements can be changed.
- Since the order of the elements in a tuple can't be changed, the position of the element in a tuple can have meaning.
- A good example of this is when a function returns multiple values. In this case, what gets returned is a tuple, with the return values as elements in the tuple. The order of the returned values is important, and a tuple ensures that the order isn’t going to change.
- Unpacking allows you to take multiple returned values from a function and store each value in its own variable.

DICTIONARIES

- Dictionaries are similar to a list in that they can be used to organize data into collections.
- But data in a dictionary isn't accessed based on its position. (Order doesn't matter).
- Data in a dictionary is organized into pairs of keys and values. You use the key to access the corresponding value.
- Where a list index is always a number, a dictionary key can be a different data type.

d = {key1:value1, key2:value2, ….}

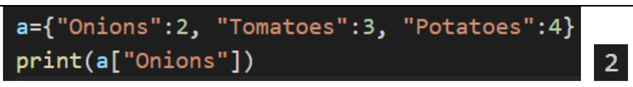
Restrictions that a dictionary must abide by:

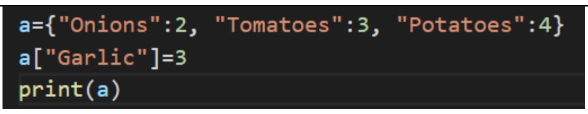
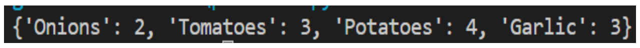
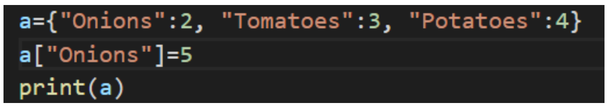
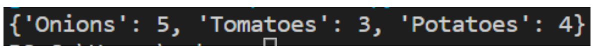
- A given key can appear in a dictionary only once.
  - Duplicate keys are not allowed.
  - A dictionary maps each key to a corresponding value, so it doesn't make sense to map a particular key more than once.
  - If you specify a key a second time during the initial creation of a dictionary, then the second occurrence will override the first.

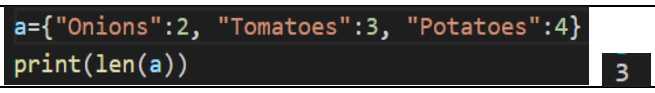
- A dictionary key must be of an immutable type.
  - For example, you can use an integer, float, string, Boolean, or even tuples as a dictionary key.
  - Neither a list nor another dictionary can serve as a dictionary key.

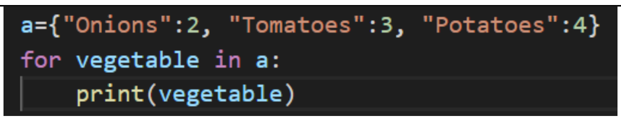
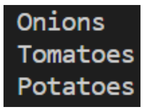
- Values, on the other hand, can be of any type and can be used more than once.

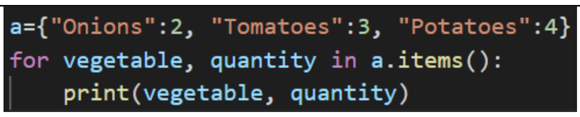
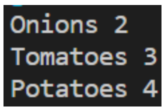
Operations Performed on a Dictionary:

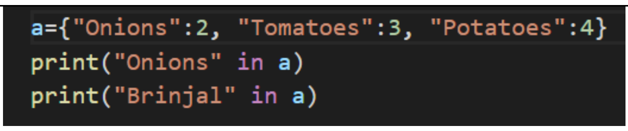
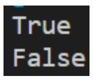
dictionary[x]	Accesses the item with x key of the dictionary	 `a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`print(a["Onions"])`
		`2`

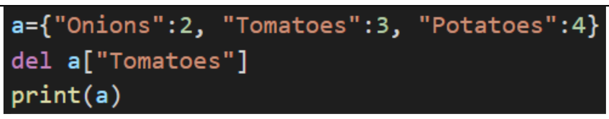
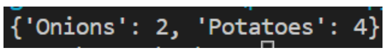
dictionary[x] = y	Sets the value 'y' associated with key 'x'	 `a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`a["Garlic"]=3`
		`print(a)`
		 `{'Onions': 2, 'Tomatoes': 3, 'Potatoes': 4, 'Garlic': 3}`
		 `a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`a["Onions"]=5`
		`print(a)`
		 `{'Onions': 5, 'Tomatoes': 3, 'Potatoes': 4}`

len(dictionary)	Returns the number of items in the dictionary	 `a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`print(len(a))`
		`3`

for keys in dictionary	Iterates over each key in the dictionary	 `a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`for vegetable in a:`
		`    print(vegetable)`
		 `Onions`
		`Tomatoes`
		`Potatoes`

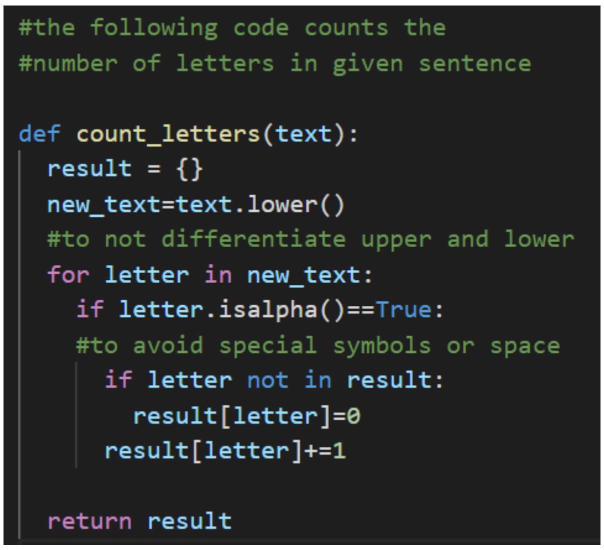
for keys, values in dictionary.items()	Iterates over each key, value pair in the dictionary	 `a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`for vegetable, quantity in a.items():`
		`    print(vegetable, quantity)`
		 `Onions 2`
		`Tomatoes 3`
		`Potatoes 4`

key in dictionary	Returns true if the given key is present in the dictionary.	 `a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`print("Onions" in a)`
		`print("Brinjal" in a)`
		 `True`
		`False`

del dictionary[key]	Removes the item with key x from the dictionary	 `a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`del a["Tomatoes"]`
		`print(a)`
		 `{'Onions': 2, 'Potatoes': 4}`

del dictionary	Deletes the dictionary.	

Counting letters:



```
#the following code counts the
#number of letters in given sentence

def count_letters(text):
    result = {}
    new_text=text.lower()
    #to not differentiate upper and lower
    for letter in new_text:
        if letter.isalpha()==True:
        #to avoid special symbols or space
            if letter not in result:
                result[letter]=0
            result[letter]+=1

    return result
```

Dictionary Methods:

dict.get(key/default)	Returns the value corresponding to key, or default if it's not present	`a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`print(a.get("Onions"))`
		
		`2`

dict.keys()	Returns a sequence containing the keys in the dictionary	`a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`print(a.keys())`
		
		`dict_keys(['Onions', 'Tomatoes', 'Potatoes'])`

dict.values()	Returns a sequence containing the values in the dictionary	`a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`print(a.values())`
		
		`dict_values([2, 3, 4])`
		
		`a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`for count in a.values():`
		`    print(count)`
		
		`2`
		`3`
		`4`

dict.update (other_dictionary)	Updates the dictionary with the items coming from the other dictionary.
	Existing entries will be replaced; new entries will be added.
		`a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`b={"Onions":5, "Garlic":6, "Ginger":7}`
		`a.update(b)`
		`print(a)`
		
		`{'Onions': 5, 'Tomatoes': 3, 'Potatoes': 4, 'Garlic': 6, 'Ginger': 7}`

dict.pop(key)	Returns the value of the corresponding key and thus deleting that key value pair in dictionary.	`d={'a':1, 'b':2, 'c':3}`
		
		`d.pop('b')`
		
		`2`

dict.clear()	Removes all the items of the dictionary.
	
	(Truncates the dictionary)
		`a={"Onions":2, "Tomatoes":3, "Potatoes":4}`
		`a.clear()`
		`print(a)`
		
		`{}`

Multi-dimensional Indexing:

```
# Given this nested dictionary grab the word "hello"
d = {'k1':[1,2,3,{'tricky':['oh','man','inception',{'target':[1,2,3,'hello']}]}]}
```

```
print(d['k1'][3]['tricky'][3]['target'][3])
```

```
hello
```

SETS

- Sets are an unordered collection of items of any data type.
- Sets items are separated by commas and enclosed within curly brackets {}.
- Sets are immutable.
- Sets do not contain duplicate items.

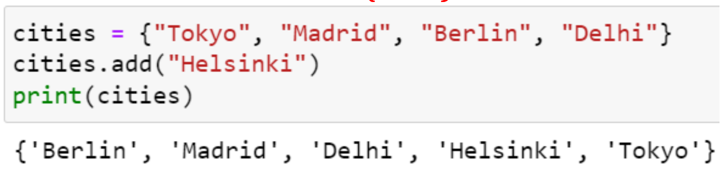
set_name = {item 1, item 2,….}

Operations on Sets:

1. *len(set)* – Returns number of items in the set.
2. *item in set* – Returns true if the item is present in it.
3. *for item in set* – Iterates over each item.

Adding elements to a set:

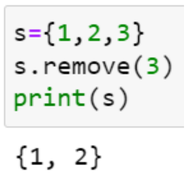
set.add(item)

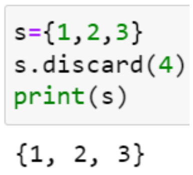


```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities.add("Helsinki")
print(cities)
```

```
{'Berlin', 'Madrid', 'Delhi', 'Helsinki', 'Tokyo'}
```

Removing items from a set:

set.remove(item)	Removes the specified item from the set.
	
	Raises an error if the item is not present in the set.
		 `s={1,2,3}`
		`s.remove(3)`
		`print(s)`
		
		`{1, 2}`

set.discard(item)	Removes the specified item from the set.
	
	Does not raise an error if the item is not present in the set.
		 `s={1,2,3}`
		`s.discard(4)`
		`print(s)`
		
		`{1, 2, 3}`

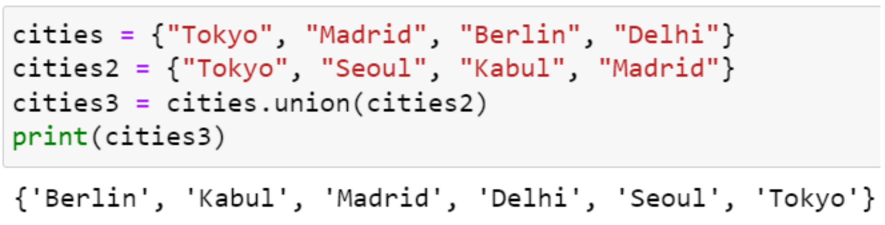
set.pop()	Returns an item and hence deleting it from the set.	
set.clear()	Truncates the set.	
del set	Deletes the set.	

SET ALGEBRA:

1. **Union:**

   The union() and update() methods combine all items that are present in the two sets.

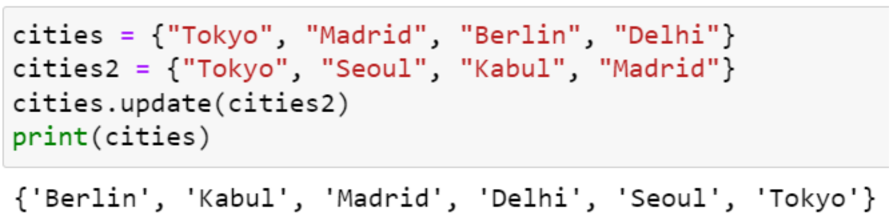
- *set1.union(set2)* returns a new set.



```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
cities3 = cities.union(cities2)
print(cities3)
```

```
{'Berlin', 'Kabul', 'Madrid', 'Delhi', 'Seoul', 'Tokyo'}
```

- *set1.update(set2)* method adds items into the existing set1 from set2.



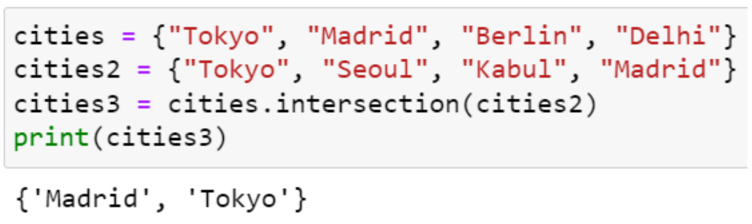
```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
cities.update(cities2)
print(cities)
```

```
{'Berlin', 'Kabul', 'Madrid', 'Delhi', 'Seoul', 'Tokyo'}
```

2. **Intersection:**

   The intersection() and intersection_update() methods keep only items that are present in both sets.

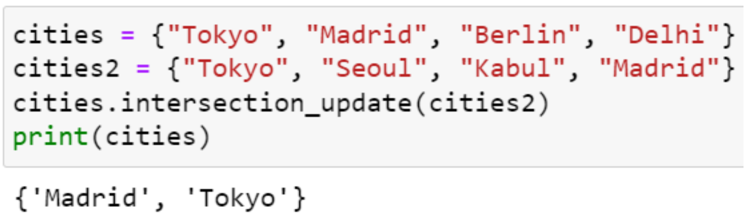
- *set1.intersection(set2)* returns a new set.



```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
cities3 = cities.intersection(cities2)
print(cities3)
```

```
{'Madrid', 'Tokyo'}
```

- *set1.intersection_update(set2)* updates into the existing set1 from set2.



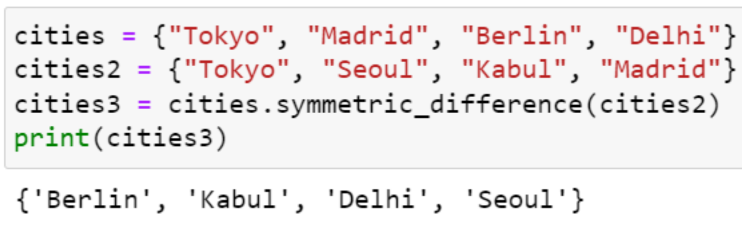
```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
cities.intersection_update(cities2)
print(cities)
```

```
{'Madrid', 'Tokyo'}
```

3. **Symmetric Difference:**

The symmetric_difference() and symmetric_difference_update() methods keep only items that are not similar to both the sets.

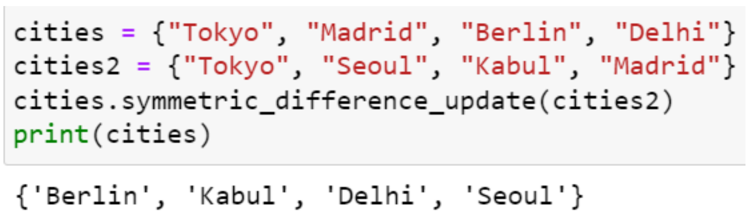
- ***set1.symmetric_difference(set2)*** returns a new set.



```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
cities3 = cities.symmetric_difference(cities2)
print(cities3)
```

```
{'Berlin', 'Kabul', 'Delhi', 'Seoul'}
```

- ***set1.symmetric_difference_update(set2)*** updates into the existing set1 from set2.



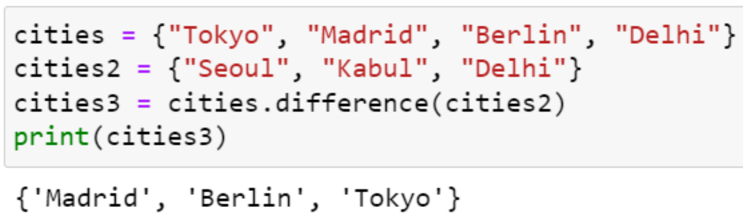
```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
cities.symmetric_difference_update(cities2)
print(cities)
```

```
{'Berlin', 'Kabul', 'Delhi', 'Seoul'}
```

4. **Difference:**

The difference() and difference_update() methods prints only items that are only present in the original set and not in both the sets.

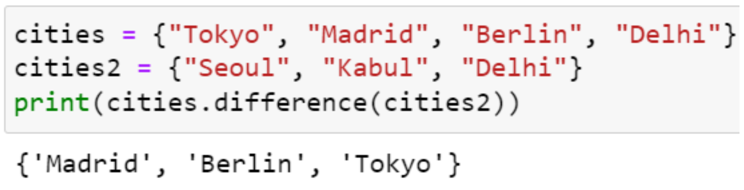
- ***set1.difference(set2)*** returns a new set.



```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Seoul", "Kabul", "Delhi"}
cities3 = cities.difference(cities2)
print(cities3)
```

```
{'Madrid', 'Berlin', 'Tokyo'}
```

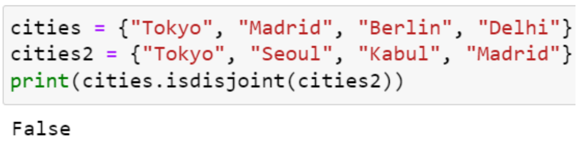
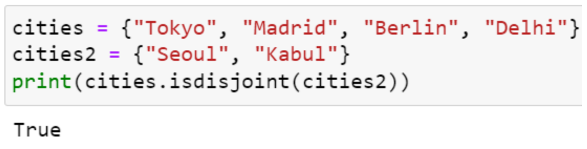
- ***set1.difference_update(set2)*** updates into the existing set1 from set2.

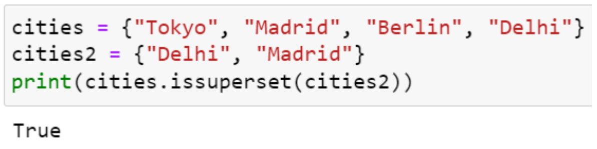
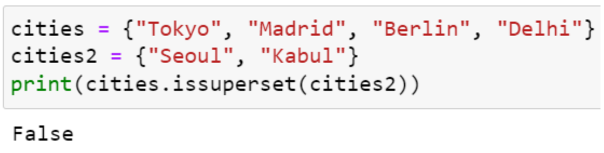


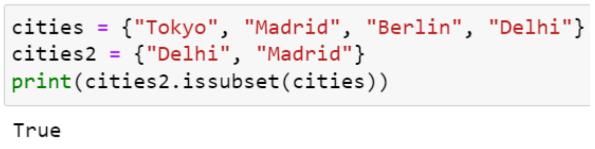
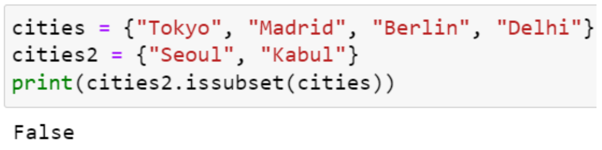
```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Seoul", "Kabul", "Delhi"}
print(cities.difference(cities2))
```

```
{'Madrid', 'Berlin', 'Tokyo'}
```

Set Methods:

set1.isjoint(set2)	Returns true if both sets do not have any common item.	 `cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}`
		`cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}`
		`print(cities.isdisjoint(cities2))`
		`False`
		 `cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}`
		`cities2 = {"Seoul", "Kabul"}`
		`print(cities.isdisjoint(cities2))`
		`True`

set1.isupper(set2)	Returns true if set2 is subset of set1.	 `cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}`
		`cities2 = {"Delhi", "Madrid"}`
		`print(cities.issuperset(cities2))`
		`True`
		 `cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}`
		`cities2 = {"Seoul", "Kabul"}`
		`print(cities.issuperset(cities2))`
		`False`

set1.issubset(set2)	Returns true if set1 is a subset of set2.	 `cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}`
		`cities2 = {"Delhi", "Madrid"}`
		`print(cities2.issubset(cities))`
		`True`
		 `cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}`
		`cities2 = {"Seoul", "Kabul"}`
		`print(cities2.issubset(cities))`
		`False`

MODULE 5

FILE HANDLING

READING AND WRITING FILES

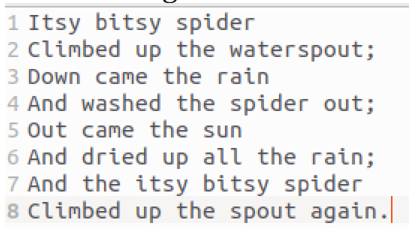
OPENING AND CLOSING FILES:

Opening the file:

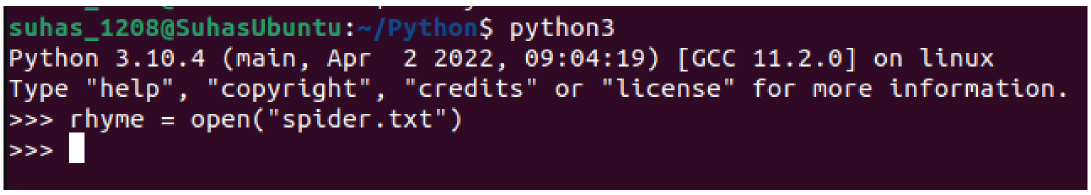
Syntax: **variable=open("file_name , mode")**

Here, open is the function and file name is the parameter passed to it.

Let the spider.txt file consist the following contents.



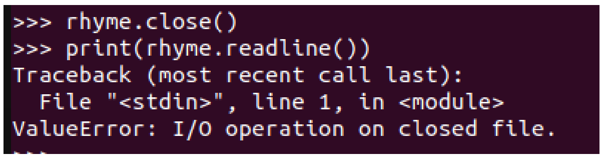
```
Itsy bitsy spider
Climbed up the waterspout;
Down came the rain
And washed the spider out;
Out came the sun
And dried up all the rain;
And the itsy bitsy spider
Climbed up the spout again.
```



```
suhas_1208@SuhasUbuntu:~/Python$ python3
Python 3.10.4 (main, Apr  2 2022, 09:04:19) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> rhyme = open("spider.txt")
>>> 
```

Closing the file:

This can be done using the close method. Syntax: **variable_name.close()**



```
>>> rhyme.close()
>>> print(rhyme.readline())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

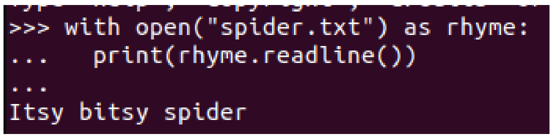
Importance of Closing a File:

1) When a file is opening your script, your file system usually locks it down so no other programs or scripts can use it until you're finished.
2) There's a limited number of file descriptors that you can create before your file system runs out of them. Although this number might be high, it's possible to open a lot of files and deplete your file system resources. This can happen if we're opening files in a loop, for example.
3) Leaving open files hanging around can lead to race conditions which occur when multiple processes try to modify and read from one resource at the same time and can cause all sorts of unexpected behaviour.

Best Practices:

1. Open-Use-Close approach:
   When we open the file, use it and later close the file manually.

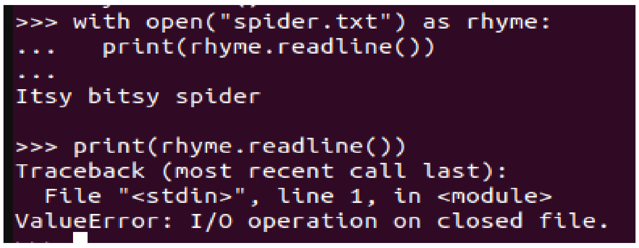
2. Closing a file automatically after using:
   This is done using 'with' keyword.
   Syntax : ***with open("file_name", "mode") as variable_name:***
   body



```
>>> with open("spider.txt") as rhyme:
...     print(rhyme.readline())
...
Itsy bitsy spider
```

Once the code is executed, the file is automatically closed.

Checking if the file is open,



```
>>> with open("spider.txt") as rhyme:
...     print(rhyme.readline())
...
Itsy bitsy spider

>>> print(rhyme.readline())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

File Opening Modes:

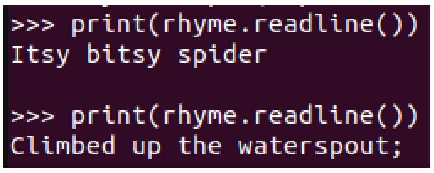
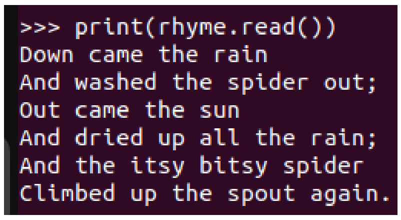
File objects can be opened in several different modes. A mode is similar to a file permission. It governs what you can do with the file you've just opened.

By default, the open function uses the **r** mode, which stands for read only. Since read only is the default, we don't have to pass the r as a second argument when we just want to read the file.

Mode	Symbol	Meaning
Read Only	r	• Used when we just want to read the file. • You get an error if you try to write to a file opened in read-only mode. • There must be an existing file to read.
Write Only	w	• Used when we just want to write. • You can't read the file contents. If you try to, the interpreter raises an error. • If the file doesn't exist, python will create the file. • If the file does exist, then its current contents will be overwritten by whatever we decide to write using our scripts.
Append	a	• Appending content at the end of an existing file.
Read-Write	r+	• To read contents and overwrite it. • There must be an existing file.

READING FILES:

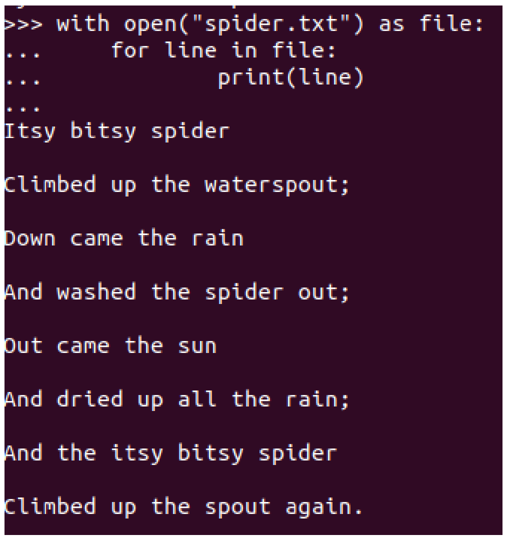
- When we open a file, the operating system checks that we have permissions to access that file and then gives our code a file descriptor.
- **File descriptor** is a token generated by the OS that allows programs to do more operations with the file.
- In Python, this file descriptor is stored as an attribute of the files object.
- The file object gives us a bunch of methods that we can use to operate with the file.

variable_name.readline()	Prints a single line from current position.	 ```>>> print(rhyme.readline())``` ```Itsy bitsy spider``` ```>>> print(rhyme.readline())``` ```Climbed up the waterspout;```
variable_name.read()	Prints till the end of the file from current position.	 ```>>> print(rhyme.read())``` ```Down came the rain``` ```And washed the spider out;``` ```Out came the sun``` ```And dried up all the rain;``` ```And the itsy bitsy spider``` ```Climbed up the spout again.```

Iterating Through Files:

We can iterate through files just as we iterate through lists and strings. We can use all the methods as that of strings and lists.

Printing contents of file line by line:



```
>>> with open("spider.txt") as file:
...     for line in file:
...             print(line)
...
Itsy bitsy spider

Climbed up the waterspout;

Down came the rain

And washed the spider out;

Out came the sun

And dried up all the rain;

And the itsy bitsy spider

Climbed up the spout again.
```

Reason for empty lines:

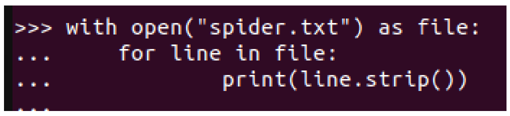
- File has a new line character at the end of each line.

So when Python reads the file line by line, the line variable will always have a new line character at the end.

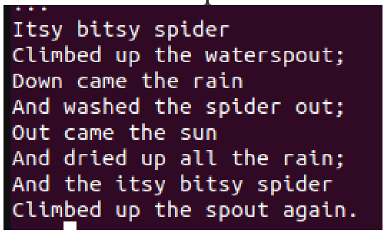
- In other words, the newline character is not removed when calling read line.
- When we ask Python to print the line, the print function adds another new line character, creating an empty line.

Printing without line gap:

We can use the **strip()** method to get rid of white spaces that are present.



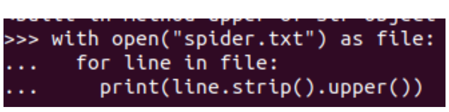
```
>>> with open("spider.txt") as file:
...     for line in file:
...             print(line.strip())
...
```



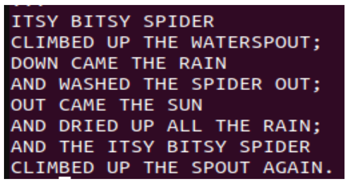
```
Itsy bitsy spider
Climbed up the waterspout;
Down came the rain
And washed the spider out;
Out came the sun
And dried up all the rain;
And the itsy bitsy spider
Climbed up the spout again.
```

Using methods of strings:

We can use many methods as that of strings. For example if we want to print every line in capital letters.



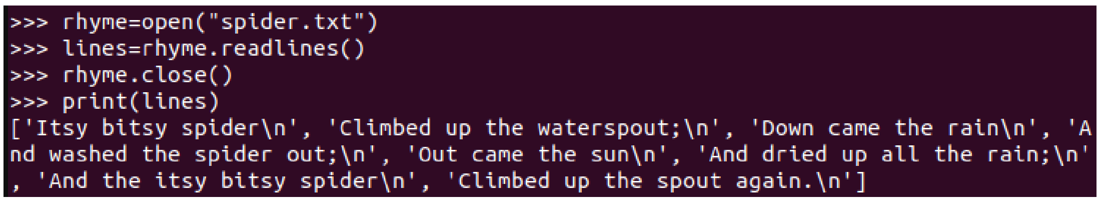
```
>>> with open("spider.txt") as file:
...   for line in file:
...     print(line.strip().upper())
```



```
ITSY BITSY SPIDER
CLIMBED UP THE WATERSPOUT;
DOWN CAME THE RAIN
AND WASHED THE SPIDER OUT;
OUT CAME THE SUN
AND DRIED UP ALL THE RAIN;
AND THE ITSY BITSY SPIDER
CLIMBED UP THE SPOUT AGAIN.
```

Reading the file lines into a list:

This can be done using the **readlines()** method. Syntax: **list_name=file.readlines()**

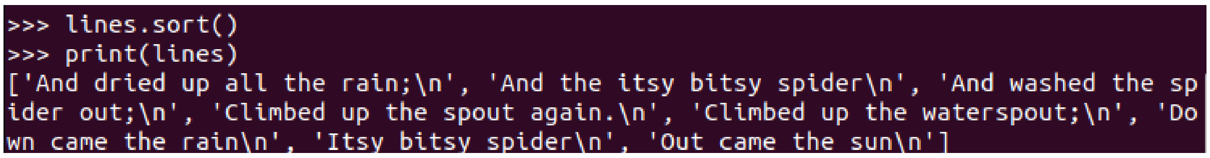


```
>>> rhyme=open("spider.txt")
>>> lines=rhyme.readlines()
>>> rhyme.close()
>>> print(lines)
['Itsy bitsy spider\n', 'Climbed up the waterspout;\n', 'Down came the rain\n', 'A
nd washed the spider out;\n', 'Out came the sun\n', 'And dried up all the rain;\n'
, 'And the itsy bitsy spider\n', 'Climbed up the spout again.\n']
```

Here \n indicates that there is a newline character at the end of the line.

Using the list:

Now once we have read the lines of the file into a list, we can use list operations on it. For example, let's sort the list.



```
>>> lines.sort()
>>> print(lines)
['And dried up all the rain;\n', 'And the itsy bitsy spider\n', 'And washed the sp
ider out;\n', 'Climbed up the spout again.\n', 'Climbed up the waterspout;\n', 'Do
wn came the rain\n', 'Itsy bitsy spider\n', 'Out came the sun\n']
```

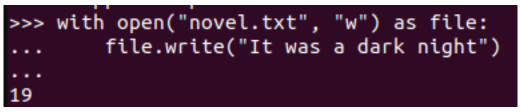
WRITING TO FILES:

This is a phenomenon in which we add new words or lines or any content to a file. Its syntax is similar to that of reading files.

Syntax: **variable_name.write("Required Things")**

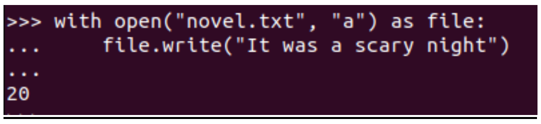
In the end, if writing is successful, it returns the number of characters we have written.

Using Write Only Mode:



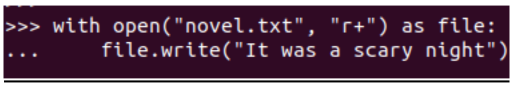
```
>>> with open("novel.txt", "w") as file:
...     file.write("It was a dark night")
...
19
```

Using Append Mode:



```
>>> with open("novel.txt", "a") as file:
...     file.write("It was a scary night")
...
20
```

Using Read-Write Mode:

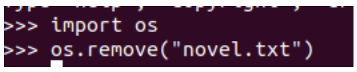
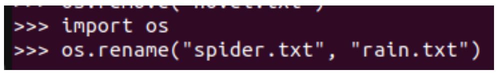


```
>>> with open("novel.txt", "r+") as file:
...     file.write("It was a scary night")
```

MANAGING FILES AND DIRECTORIES

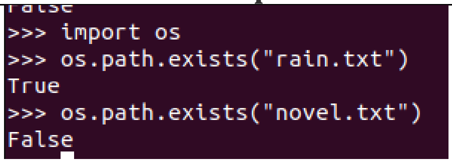
WORKING WITH FILES:

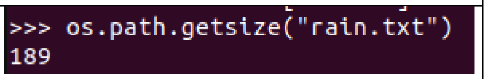
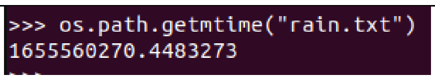
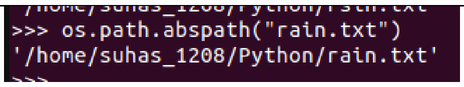
For this, we will be using the functions from **OS module.**
It allows us to interact with the underlying system. The OS module lets us do pretty much all the same tasks that we can normally do when working with files from the command line.

Function	Use	Example
remove	Delete any file that exists.	 `>>> import os` `>>> os.remove("novel.txt")`
rename	Rename any file that exists.	 `>>> import os` `>>> os.rename("spider.txt", "rain.txt")`

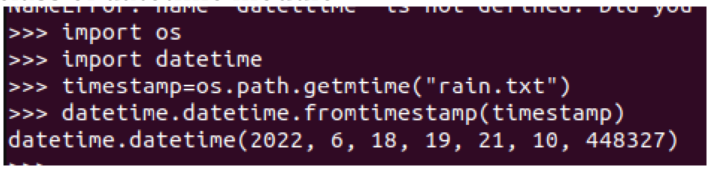
Path submodule in OS module:
There's a sub-module inside the OS module for dealing with things related to file information like whether they exist or not. This is called the **OS path** sub-module.

Function	Use	Example

path.exists(name)	To check if the file exists.	 `>>> import os`
		`>>> os.path.exists("rain.txt")`
		`True`
		`>>> os.path.exists("novel.txt")`
		`False`

path.getsize(name)	Returns the size of the file in bytes	 `>>> os.path.getsize("rain.txt")` `189`
path.getmtime(name)	Returns the time at which the file was modified in terms of timestamp.(In this case Unix timestamp. It returns the number of seconds since Jan 1st, 1970)	 `>>> os.path.getmtime("rain.txt")` `1655560270.4483273`
path.aspath(name)	Returns absolute path of the system.	 `>>> os.path.abspath("rain.txt")` `'/home/suhas_1208/Python/rain.txt'`

Datetime Module:
In this module, we can change the timestamp into a readable date using **fromdatetime** function in datetime class of datetime module.

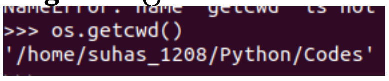


```
>>> import os
>>> import datetime
>>> timestamp=os.path.getmtime("rain.txt")
>>> datetime.datetime.fromtimestamp(timestamp)
datetime.datetime(2022, 6, 18, 19, 21, 10, 448327)
```

DIRECTORIES:

Checking your current working directory:

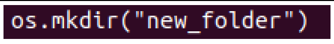
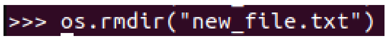
This can be done with the help of **getcwd**() method.

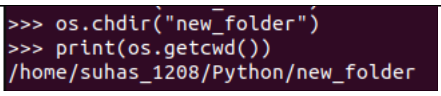


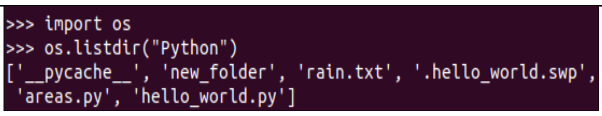
```
>>> os.getcwd()
'/home/suhas_1208/Python/Codes'
```

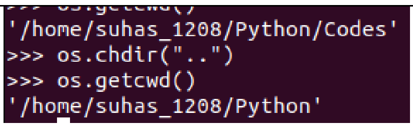
Useful Functions for directories:

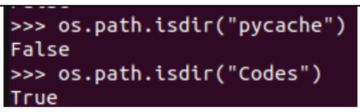
These are some functions that can be used from OS module.

mkdir(name)	Create a new directory.	 `os.mkdir("new_folder")`
rmdir(name)	Remove an empty directory.	 `>>> os.rmdir("new_file.txt")`

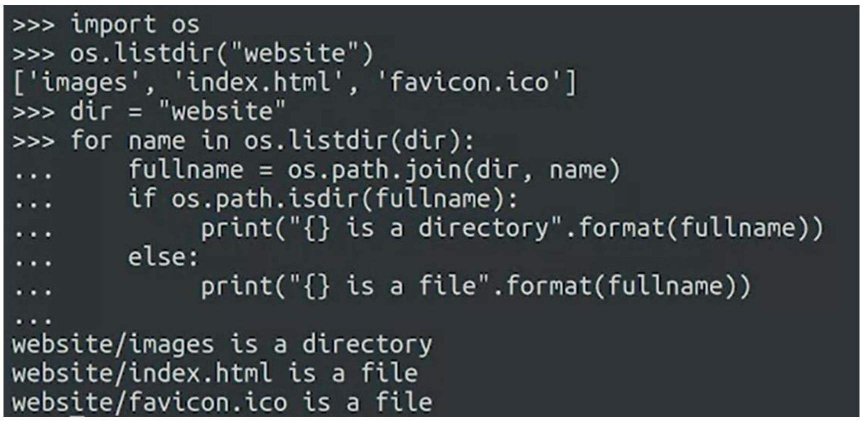
chdir(name)	Changing from current directory to any subdirectory.	 `>>> os.chdir("new_folder")`
		`>>> print(os.getcwd())`
		`/home/suhas_1208/Python/new_folder`

listdir(name)	Returns the contents of the specified subdirectory but doesn't show whether if it's a file or a directory.	 `>>> import os`
		`>>> os.listdir("Python")`
		`['__pycache__', 'new_folder', 'rain.txt', '.hello_world.swp',`
		`'areas.py', 'hello_world.py']`

chdir("..")	Changes from current directory to parent directory	 `'/home/suhas_1208/Python/Codes'`
		`>>> os.chdir("..")`
		`>>> os.getcwd()`
		`'/home/suhas_1208/Python'`

path.isdir(name)	Returns true if there is a directory with given name. in current directory.	 `>>> os.path.isdir("pycache")`
		`False`
		`>>> os.path.isdir("Codes")`
		`True`

Code to check if the content is a file or directory:



```
>>> import os
>>> os.listdir("website")
['images', 'index.html', 'favicon.ico']
>>> dir = "website"
>>> for name in os.listdir(dir):
...     fullname = os.path.join(dir, name)
...     if os.path.isdir(fullname):
...         print("{} is a directory".format(fullname))
...     else:
...         print("{} is a file".format(fullname))
...
website/images is a directory
website/index.html is a file
website/favicon.ico is a file
```

READING AND WRITING CSV FILES

Parsing:

Analyzing a file's content to correctly structure the data. (occurs while reading). It can also be defined as using rules to understand a file or data stream as structured data.

CSV Files:

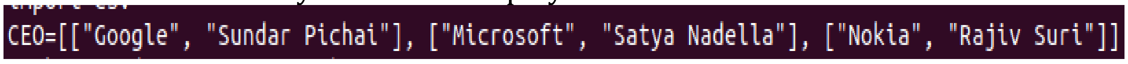
CSV or Comma Separated Values is a common data format used to store data as a segment of text separated by commas.

CSV files can be thought as spreadsheet applications like MS Excel, Google sheets where each line corresponds to a row and each comma-separated field corresponds to a column.

CSV module is used to process CSV files.

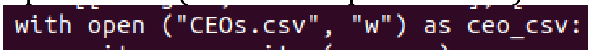
Generating (writing) CSV files:

1. Import the CSV module.
2. Create the list of lists you want to display.



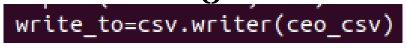
```
CEO=[["Google", "Sundar Pichai"], ["Microsoft", "Satya Nadella"], ["Nokia", "Rajiv Suri"]]
```

3. Open the file(with the required name) in the write mode and store it as a variable.



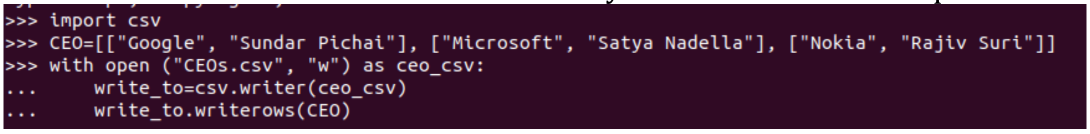
```
with open ("CEOs.csv", "w") as ceo_csv:
```

4. Call the **writer()** function of CSV Module and pass the file variable as parameter.



```
write_to=csv.writer(ceo_csv)
```

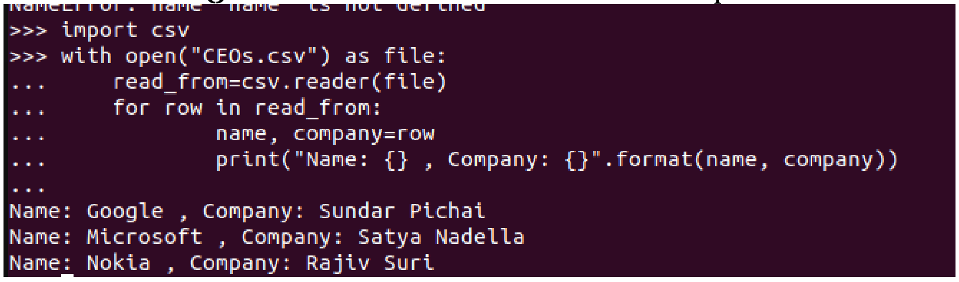
5. Now you can use **writerow()** function to write only single row and **writerows()** function to write all the rows. Pass the list name that you have created as the parameter.



```
>>> import csv
>>> CEO=[["Google", "Sundar Pichai"], ["Microsoft", "Satya Nadella"], ["Nokia", "Rajiv Suri"]]
>>> with open ("CEOs.csv", "w") as ceo_csv:
...     write_to=csv.writer(ceo_csv)
...     write_to.writerows(CEO)
```

Reading CSV Files:

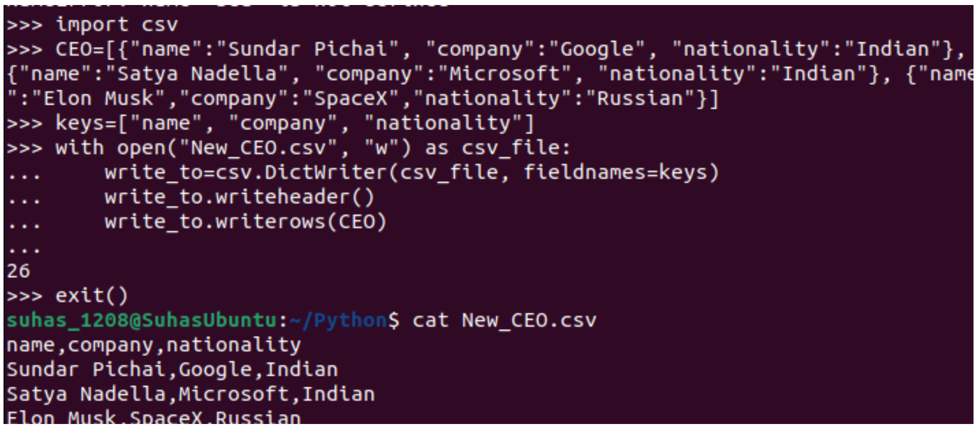
1. Import the CSV Module.
2. Open the file with open() function with filename as parameter and store it as a variable.
3. Use the **reader()** function with the variable name as parameter.



```
>>> import csv
>>> with open("CEOs.csv") as file:
...     read_from=csv.reader(file)
...     for row in read_from:
...             name, company=row
...             print("Name: {} , Company: {}".format(name, company))
...
Name: Google , Company: Sundar Pichai
Name: Microsoft , Company: Satya Nadella
Name: Nokia , Company: Rajiv Suri
```

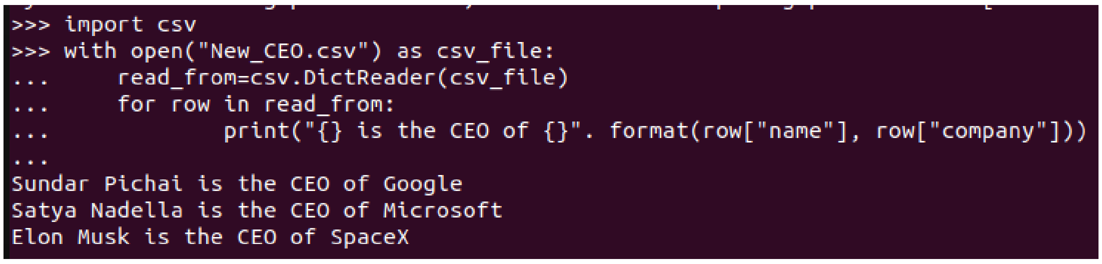
Using Dictionaries:

- **Writing:**
    1. Create a list of dictionaries.
    2. List the keys.
    3. Open the file in write mode and save that as file variable.
    4. Call the **DitWriter** function and pass the file variable and **fileldnames=**list_of_keys as parameters.
    5. Use the **writeheader**() method to write headings.
    6. Use the **writerows**() method to write rows and pass the list of dictionaries.



```
>>> import csv
>>> CEO=[{"name":"Sundar Pichai", "company":"Google", "nationality":"Indian"},
{"name":"Satya Nadella", "company":"Microsoft", "nationality":"Indian"}, {"name
":"Elon Musk","company":"SpaceX","nationality":"Russian"}]
>>> keys=["name", "company", "nationality"]
>>> with open("New_CEO.csv", "w") as csv_file:
...     write_to=csv.DictWriter(csv_file, fieldnames=keys)
...     write_to.writeheader()
...     write_to.writerows(CEO)
...
26
>>> exit()
suhas_1208@SuhasUbuntu:~/Python$ cat New_CEO.csv
name,company,nationality
Sundar Pichai,Google,Indian
Satya Nadella,Microsoft,Indian
Elon Musk,SpaceX,Russian
```

- **Reading:**
    1. Open file in read mode.
    2. Use the DictReader function and pass file variable as parameter.



```
>>> import csv
>>> with open("New_CEO.csv") as csv_file:
...     read_from=csv.DictReader(csv_file)
...     for row in read_from:
...             print("{} is the CEO of {}". format(row["name"], row["company"]))
...
Sundar Pichai is the CEO of Google
Satya Nadella is the CEO of Microsoft
Elon Musk is the CEO of SpaceX
```