UNIT-III

Functions in Python: Defining a Function, Calling a Function, Parameters, Recursive Functions.

List: Creating Lists using range() Function, Operations on Lists, Methods to Process List, Sorting the List Elements.

Tuple: Creating Tuples, Accessing the Tuple Elements, Operations on Tuple, Functions to Process Tuples.

Dictionaries: Operations on Dictionaries, Dictionary Methods, Sorting the Elements of a Dictionary using Lambdas, Converting Lists into Dictionary, Converting Strings into Dictionary, Passing Dictionaries to Functions.

Functions in Python

Functions are the structured or procedural programming way of organizing the logic in your programs. Large blocks of code can be neatly segregated into manageable chunks.

A function is a block of code that performs a specific task.

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Python Function Declaration

The **syntax** to declare a function is:

def function_name(arguments):

```
    # function body

    return
```

Here,

- **def** - keyword used to declare a function
- **function_name** - any name given to the function
- **arguments** - any value passed to function
- **return** (optional) - returns value from a function

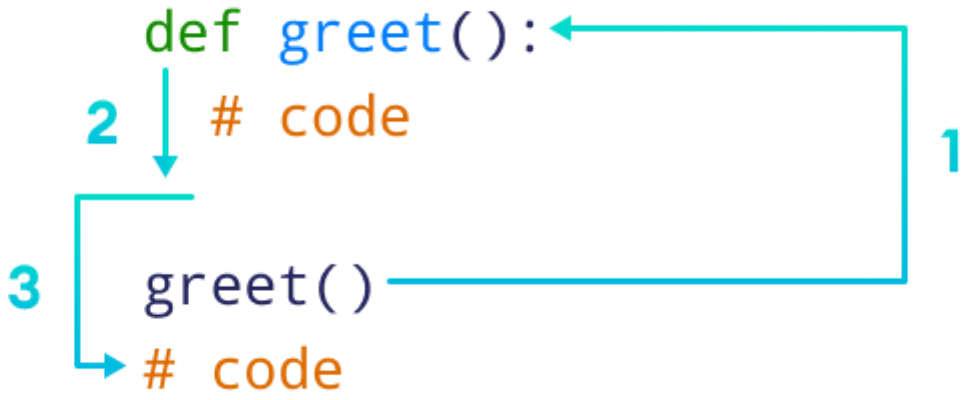
In [1]:

```
def greet():
    print('Hello World!')
```

In [2]:

```
def hello():
    """This function says hello and greets you"""
    print("Hello")
    print("Glad to meet you")
```

Calling a Function



In [3]:

```
greet()
```

```
Hello World!
```

In [4]:

```
hello()
```

```
Hello
Glad to meet you
```

In [5]:

```
def greet():
    print('Hello World!')

# call the function
greet()

print('Outside function')
```

```
Hello World!
Outside function
```

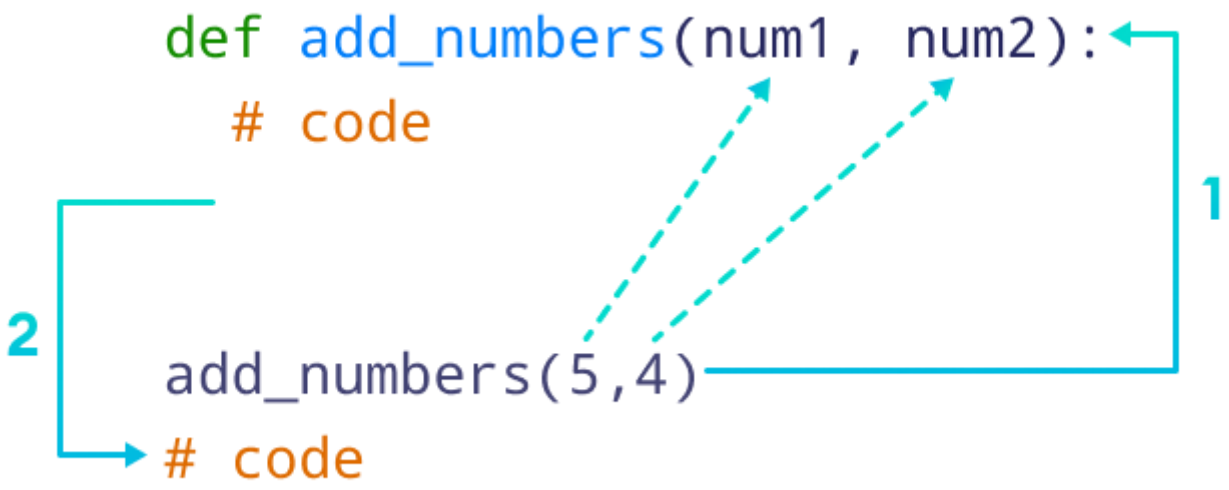
In [6]:

```
def hello():
    print("Hello")
    print("Glad to meet you")

print(type(hello))
print(type("hello"))

hello()
print("Hey, that just printed two lines with one line of code!")
hello()  # do it again, just because we can...
```

```
<class 'function'>
<class 'str'>
Hello
Glad to meet you
Hey, that just printed two lines with one line of code!
Hello
Glad to meet you
```

Function Arguments

function can also have arguments. An argument is a value that is accepted by a function.



In [7]:

```
def hello2(s):
    print("Hello " + s)
    print("Glad to meet you")
hello2("II CSE A Class")
```

```
Hello II CSE A Class
Glad to meet you
```

In [8]:

```
def hello2(s):
    print("Hello " + s)
    print("Glad to meet you")

hello2("Iman" + " and Jackie")
hello2("Class " * 3)
```

```
Hello Iman and Jackie
Glad to meet you
Hello Class Class Class
Glad to meet you
```

In [9]:

```
# function with two arguments
def add_numbers(num1, num2):
    sum = num1 + num2
    print('Sum: ',sum)
# function call with two values
add_numbers(5, 4)
```

```
Sum:  9
```

In [10]:

```
def hello3(s, n):
    greeting = "Hello {} ".format(s)
    print(greeting*n)

hello3("Wei", 4)
hello3("", 1)
hello3("Kitty", 11)
```

```
Hello Wei Hello Wei Hello Wei Hello Wei
Hello
Hello Kitty Hello Kitty Hello Kitty Hello Kitty Hello Kitty Hello Kitty He
llo Kitty Hello Kitty Hello Kitty Hello Kitty Hello Kitty
```

In [11]:

```
def print_many(x, y):
    """Print out string x, y times."""
    for i in range(y):
        print(x)
print_many(2,5)
print_many("f",5)
```

```
2
2
2
2
2
f
f
f
f
f
```

In [12]:

```
def cyu(s1, s2):
    if len(s1) > len(s2):
        print(s1)
    else:
        print(s2)

cyu("Hello", "Goodbye")
```

```
Goodbye
```

In [14]:

```
def square(x):
    y = x * x
    print(y)   # Bad! This is confusing! Should use return instead!

toSquare = 10
squareResult = square(toSquare)
print("The result of {} squared is {}.".format(toSquare, squareResult))
```

```
100
The result of 10 squared is None.
```

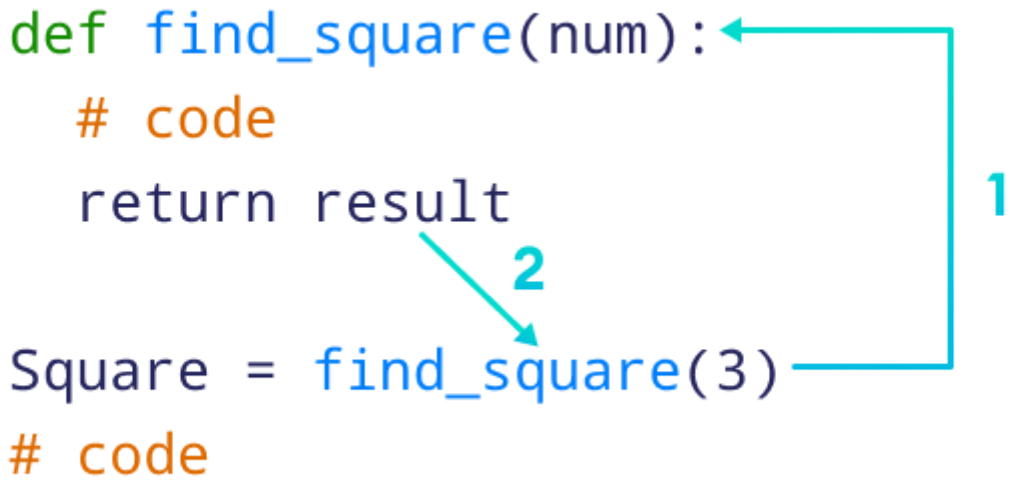
return Statement

A function may or may not return a value. If we want our function to return some value to a function call, we use the return statement.

def add_numbers():

```
    ...

    return sum
```

Note: The return statement also denotes that the function has ended. Any code after return is not executed.



In [15]:

```
# function definition
def find_square(num):
    result = num * num
    return result

# function call
square = find_square(3)

print('Square:',square)
```

```
Square: 9
```

In [13]:

```
def square(x):
    y = x * x
    return y

toSquare = 10
result = square(toSquare)
print("The result of {} squared is {}.".format(toSquare, result))
```

```
The result of 10 squared is 100.
```

In [16]:

```
# function that adds two numbers
def add_numbers(num1, num2):
    sum = num1 + num2
    return sum

# calling function with two values
result = add_numbers(5, 4)

print('Sum: ', result)
```

```
Sum:  9
```

In [17]:

```
def weird():
    print("here")
    return 5
    print("there")
    return 10

x = weird()
print(x)
```

```
here
5
```

In [18]:

```
def longer_than_five(list_of_names):
    for name in list_of_names: # iterate over the list to look at each name
        if len(name) > 5: # as soon as you see a name longer than 5 letters,
            return True # then return True!
            # If Python executes that return statement, the function is over and the res
    return False # You will only get to this line if you
    # iterated over the whole list and did not get a name where
    # the if expression evaluated to True, so at this point, it's correct to return Fals

# Here are a couple sample calls to the function with different lists of names. Try runn

list1 = ["Sam","Tera","Sal","Amita"]
list2 = ["Rey","Ayo","Lauren","Natalie"]

print(longer_than_five(list1))
print(longer_than_five(list2))
```

```
False
True
```

In [19]:

```
def square(x):
    y = x * x
    return y

print(square(square(2)))
```

```
16
```

In [20]:

```
def square(x):
    print("square")
    return x*x

def g(y):
    print("g")
    return y + 3

print(square(g(2)))
```

```
g
square
25
```

In [21]:

```
def cyu2(s1, s2):
    x = len(s1)
    y = len(s2)
    return x-y

z = cyu2("Yes", "no")
if z > 0:
    print("First one was longer")
else:
    print("Second one was at least as long")
```

```
First one was longer
```

In [22]:

```
def mylen(seq):
    c = 0 # initialize count variable to 0
    for _ in seq:
        c = c + 1   # increment the counter for each item in seq
    return c

print(mylen("hello"))
print(mylen([1, 2, 7]))
print(mylen((1,5,7,8)))
```

```
5
3
4
```

Variables and parameters are local An assignment statement in a function creates a local variable for the variable on the left hand side of the assignment operator. It is called local because this variable only exists inside the function and you cannot use it outside. For example, consider again the square function:

In [23]:

```
def square(x):
    y = x * x
    return y

z = square(10)
print(y)
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[23], line 6
      3     return y
      5 z = square(10)
----> 6 print(y)

NameError: name 'y' is not defined
```

In [24]:

```
def adding(x):
    y = 3
    z = y + x
    return z

def producing(x):
    y=2
    z = x * y
    return z

print(producing(adding(4)))
```

```
14
```

Global Variables Variable names that are at the top-level, not inside any function definition, are called global.

In [25]:

```
def badsquare(x):
    y = x ** power
    return y

power = 2
result = badsquare(10)
print(result)
```

```
100
```

In [3]:

```
def powerof(x,p):
    power = p   # Another dumb mistake
    y = x ** power
    print(power)
    return y

power = 3
result = powerof(10,2)
print(result)
print(power)
```

```
2
100
3
```

In [27]:

```
def double(y):
    y = 2 * y

def changeit(lst):
    lst[0] = "Michigan"
    lst[1] = "Wolverines"

y = 5
double(y)
print(y)

mylst = ['our', 'students', 'are', 'awesome']
changeit(mylst)
print(mylst)
```

```
5
['Michigan', 'Wolverines', 'are', 'awesome']
```

In [28]:

```
def changeit(lst):
    lst[0] = "Michigan"
    lst[1] = "Wolverines"
    return lst

mylst = ['106', 'students', 'are', 'awesome']
newlst = changeit(list(mylst))
print(mylst)
print(newlst)
```

```
['106', 'students', 'are', 'awesome']
['Michigan', 'Wolverines', 'are', 'awesome']
```

In [29]:

```
def square(x):
    y = x * x
    return y

def sum_of_squares(x,y,z):
    a = square(x)
    b = square(y)
    c = square(z)

    return a+b+c

a,b,c = -5,2,10
result = sum_of_squares(a,b,c)
print(result)
```

```
129
```

In [30]:

```
initial = 7
def f(x, y =3, z=initial):
    print("x, y, z, are: " + str(x) + ", " + str(y) + ", " + str(z))

f(2)
f(2, 5)
f(2, 5, 8)
f(x=2,z=8)
```

```
x, y, z, are: 2, 3, 7
x, y, z, are: 2, 5, 7
x, y, z, are: 2, 5, 8
x, y, z, are: 2, 3, 8
```

In [31]:

```
def adder(x,y,z):
    print("sum:",x+y+z)

adder(5,10,15)
```

```
sum: 30
```

- Non-keyword Variable Arguments (Tuple) def function_name([formal_args,] *vargs_tuple): "function_documentation_string" function_body_suite The asterisk operator (*) is placed in front of the variable that will hold all remaining arguments once all the formal parameters if have been exhausted. The tuple is empty if there are no additional arguments given.

In [32]:

```
def adder(*num):
    sum = 0

    for n in num:
        sum = sum + n

    print("Sum:",sum)

adder(3,5)
adder(4,5,6,7)
adder(1,2,3,5,6)
```

```
Sum: 8
Sum: 22
Sum: 17
```

In [33]:

```
def avg(*args):
    return sum(args)/len(args)

av = avg(20,21,27)
print("Average of given numbers is :", av)
```

```
Average of given numbers is : 22.666666666666668
```

- Keyword Variable Arguments (Dictionary) def function_name([formal_args,][*vargst,] **vargsd): function_documentation_string function_body_suite

In [34]:

```
def printPetNames(owner, **pets):
    print("Owner Name: " + owner)
    for pet,name in pets.items():
        print(pet," : ", name)
printPetNames("Jonathan", dog="Brock", fish="Larry", turtle="Shelldon")
```

```
Owner Name: Jonathan
dog  :  Brock
fish  :  Larry
turtle  :  Shelldon
```

In [35]:

```
def intro(**data):
    print("\nData type of argument:",type(data))

    for key, value in data.items():
        print("{} is {}".format(key,value))

intro(Firstname="Sita", Lastname="Sharma", Age=22, Phone=1234567890)
intro(Firstname="John", Lastname="Wood", Email="johnwood@nomail.com",
      Country="Wakanda", Age=25, Phone=9876543210)
```

```
Data type of argument: <class 'dict'>
Firstname is Sita
Lastname is Sharma
Age is 22
Phone is 1234567890

Data type of argument: <class 'dict'>
Firstname is John
Lastname is Wood
Email is johnwood@nomail.com
Country is Wakanda
Age is 25
Phone is 9876543210
```

- Python Anonymouns Function - Lambda function Python allows one to create anonymous functions using the lambda keyword. They are “anonymous” because they are not declared in the standard manner, i.e., using the def statement. lambda [arg1[, arg2, ... argN]]: expression

def true(a): return True

lambda a:True

In [36]:

```
def add(x,y):
    return x+y
print(add(3,4))
print((lambda x,y: x+y)(3,4))
```

```
7
7
```

In [37]:

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)
```

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

In [38]:

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```

```
[4, 6, 8, 12]
```

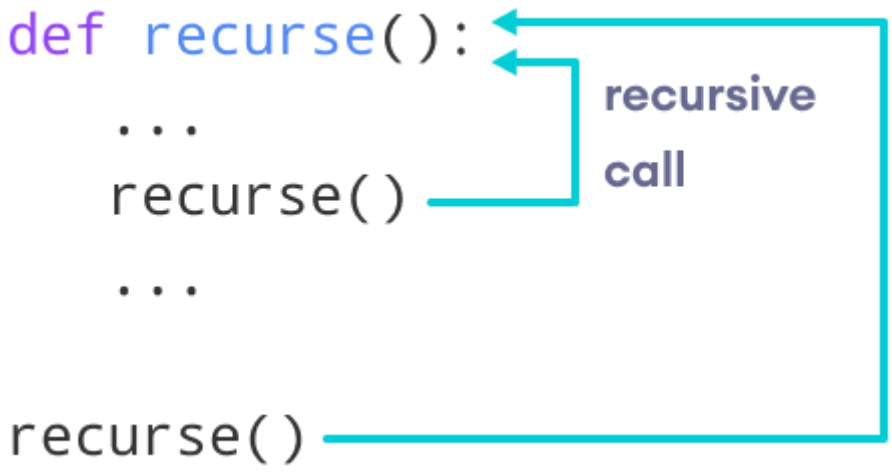
In [39]:

```
def factorial(n):
    if n == 0 or n == 1: # 0! = 1! = 1
        return 1
    else:
        return (n * factorial(n-1))
print(factorial(6))
```

```
720
```

Recursive Functions

a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.



In [40]:

```
#find the factorial of an integer
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```

```
The factorial of 3 is 6
```

By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in RecursionError.

In [ ]:

```
def recursor():
    recursor()
recursor()
```

In [1]:

```
# Program to print the fibonacci series upto n_terms

# Recursive function
def recursive_fibonacci(n):
    if n <= 1:
        return n
    else:
        return(recursive_fibonacci(n-1) + recursive_fibonacci(n-2))

n_terms = 10

# check if the number of terms is valid
if n_terms <= 0:
    print("Invalid input ! Please input a positive value")
else:
    print("Fibonacci series:")
for i in range(n_terms):
    print(recursive_fibonacci(i))
```

```
Fibonacci series:
0
1
1
2
3
5
8
13
21
34
```

Python Collections

There are four collection data types in the Python programming language:

1. **List** is a collection which is ordered and changeable. Allows duplicate members.
2. **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.
3. **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
4. **Set** is a collection which is unordered and unindexed. No duplicate members.

Lists

List is a collection which is ordered and changeable. Allows duplicate members. Lists are objects. There are many methods associated to them.

A list is a sequential collection of Python data values, where each value is identified by an index. The values that make up a list are called its elements.

Lists are similar to strings, which are ordered collections of characters, except that the elements of a list can have any type and for any one list, the items can be of different types.

A list in Python is used to store the sequence of various types of data. Python lists are mutable type its mean we can modify its element after it created. However, Python consists of six data-types that are capable to store the sequences, but the most common and reliable type is the list.

The important properties of Python lists are as follows:

-**Lists are ordered** – Lists remember the order of items inserted.

-**Accessed by index** – Items in a list can be accessed using an index.

-**Lists can contain any sort of object** – It can be numbers, strings, tuples and even other lists.

-**Lists are changeable (mutable)** – You can change a list in-place, add new items, and delete or update

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

There are several ways to create a new list.

[10, 20, 30, 40]

["spam", "bungee", "swallow"]

The first example is a list of four integers. The second is a list of three strings.

As we said above, the elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list.

["hello", 2.0, 5, [10, 20]]

https://youtu.be/mrwSbE5MDn0 (https://youtu.be/mrwSbE5MDn0)

In [1]:

```
# accessing list items
l = [1, 2, 3]
l[0]
```

Out[1]:

```
1
```

In [2]:

```
lst=["abc",2.34,"c"]
print(lst[0])
```

```
abc
```

In [3]:

```
# type of lists
print(type(l))
print(type(lst))
```

```
<class 'list'>
<class 'list'>
```

In [4]:

```
#Negative Indexing
thislist = ["apple", "banana", "cherry"]
print(thislist[-2])
```

```
banana
```

In [5]:

```
a = [1,2,"Peter",4.50,"Ricky",5,6]
b = [1,2,5,"Peter",4.50,"Ricky",6]
print(a == b)
```

```
False
```

In [6]:

```
numbers = [17, 123, 87, 34, 66, 8398, 44]
print(numbers[2])
print(numbers[9-8])
print(numbers[-2])
```

```
87
123
8398
```

In [7]:

```
prices = (1.99, 2.00, 5.50, 20.95, 100.98)
print(prices[0])
print(prices[-1])
print(prices[3-5])
```

```
1.99
100.98
20.95
```

List Creation

4 Ways to Create a List in Python

1. Create a List with List function in Python
2. Create a List with range function in Python
3. Create a List with append method in Python
4. Create a List with list comprehension in Python

Create a List with List function in Python

Python list() function takes any iterable as a parameter and returns a list. In Python iterable is the object we can iterate over. Some examples of iterables are tuples, strings, and lists.

We can use this list function creates a list from an iterable object.

Syntax: list(iterable)

iterable: an object that could be a sequence (string, tuples) or collection (set, dictionary) or any iterator object.

In [1]:

```
string1 = "ABCDEF"
list1 = list(string1)
print(list1)
```

```
['A', 'B', 'C', 'D', 'E', 'F']
```

In [1]:

```
dict1 = {"name":"Eyong","age":30,"gender":"Male"} # define a dict
list2 = list(dict1)
print(list2)
```

```
['name', 'age', 'gender']
```

Create a List with range function in Python

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.In Python 3, we can combine the range function and list function to create a new list.

Range function Syntax

range(start, stop, step) – range() takes mainly three arguments having the same use in both definitions:

start – integer starting from which the sequence of integers is to be returned.

stop – integer before which the sequence of integers is to be returned. The range of integers ends at stop – 1.

step (Optional) – integer value which determines the increment between each integer in the sequence

In [2]:

```
print(list(range(2, 4)))
```

```
[2, 3]
```

In [3]:

```
print(list(range(2, 6)))
```

```
[2, 3, 4, 5]
```

In [4]:

```
print(list(range(2, 10, 2)))
```

```
[2, 4, 6, 8]
```

In [5]:

```
print(list(range(5,-1,-1)))
```

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

In [10]:

```
My_list = [range(10, 20, 1)]

# Print the list
print(My_list)
```

```
[range(10, 20)]
```

In [11]:

```
My_list = [*range(10, 21, 1)]

# Print the list
print(My_list)
```

```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

In [12]:

```
# Create an empty list
My_list = []

# Value to begin and end with
start, end = 10, 20

# Check if start value is smaller than end value
if start < end:
    # unpack the result
    My_list.extend(range(start, end))
    # Append the last value
    My_list.append(end)

# Print the list
print(My_list)
```

```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Create a List with append method in Python

The list **append()** method in Python adds a single item to the end of the existing list. We commonly use append() to add the first element to an empty list.

After appending to the list, the size of the list increases by one.

SYNTAX – list_name.append(item)

PARAMETERS – The append() method takes a single item as an input parameter and adds that to the end of the list.

Return Value – The append() method only modifies the original list. It doesn't return any value as a return but will just modify the created list.

In [2]:

```
characters1=[]
characters1.append('Google')
print('Updated list:', characters1)
```

```
Updated list: ['Google']
```

In [3]:

```
L = ['red', 'green', 'blue']
L.append('yellow')
print(L)
```

```
['red', 'green', 'blue', 'yellow']
```

Create a List with list comprehension in Python

List comprehension offers a shorter syntax when we want to create a new list based on the values of an existing list.

List comprehension is a sublime way to define and build lists with the help of existing lists.

In comparison to normal functions and loops, List comprehension is usually more compact and faster for creating lists.

However, we should always avoid writing very long list comprehensions in one line to confirm that code is user-friendly.

Remember, every list comprehension is rewritten in for loop, but every for loop can't be rewritten within the kind of list comprehension.

In [5]:

```
separated_letters = [ letter for letter in 'Google' ]
print( separated_letters)
```

```
['G', 'o', 'o', 'g', 'l', 'e']
```

In [6]:

```
even_list = [ i for i in range(10) if i % 2 == 0]
print(even_list)
```

```
[0, 2, 4, 6, 8]
```

In [7]:

```
new_list = [num * 2 for num in range(5)]
print(new_list)
```

```
[0, 2, 4, 6, 8]
```

In [8]:

```
list = ['even' if y%2==0 else 'odd' for y in range(5)]
print(list)
```

```
['even', 'odd', 'even', 'odd', 'even']
```

In [9]:

```
matrix = [[1, 2], [3,4], [5,6], [7,8]]
transpose_matrix = [[row[i] for row in matrix] for i in range(2)]
print (transpose_matrix)
```

```
[[1, 3, 5, 7], [2, 4, 6, 8]]
```

Lists are Mutable

Unlike strings, lists are mutable. This means we can change an item in a list by accessing it directly as part of the assignment statement.

Using the indexing operator (square brackets) on the left side of an assignment, we can update one of the list items. https://youtu.be/fnSijYDKz3c (https://youtu.be/fnSijYDKz3c)

In [13]:

```
fruit = ["banana", "apple", "cherry"]
print(fruit)

fruit[0] = "pear"
fruit[-1] = "orange"
print(fruit)
```

```
['banana', 'apple', 'cherry']
['pear', 'apple', 'orange']
```

In [14]:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

```
['apple', 'blackcurrant', 'cherry']
```

In [15]:

```
#By combining assignment with the slice operator we can update several elements at once.
alist = ['a', 'b', 'c', 'd', 'e', 'f']
alist[1:3] = ['x', 'y']
print(alist)
```

```
['a', 'x', 'y', 'd', 'e', 'f']
```

In [16]:

```
#We can also remove elements from a list by assigning the empty list to them.
alist = ['a', 'b', 'c', 'd', 'e', 'f']
alist[1:3] = []
print(alist)
```

```
['a', 'd', 'e', 'f']
```

In [17]:

```
#We can even insert elements into a list by squeezing them into an empty slice at the de
alist = ['a', 'd', 'f']
alist[1:1] = ['b', 'c']
print(alist)
alist[4:4] = ['e']
print(alist)
```

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

In [18]:

```
alist = [4, 2, 8, 6, 5]
alist[2:2] = "false"
print(alist)
```

```
[4, 2, 'f', 'a', 'l', 's', 'e', 8, 6, 5]
```

List Operators

The concatenation (+) and repetition (*) operators work in the same way as they were working with the strings.

In [19]:

```
#The + operator can be used to extend a list:
my_list = [1]
my_list += [2]
print(my_list)
```

```
[1, 2]
```

In [20]:

```
my_list += [3, 4]
print(my_list)
```

```
[1, 2, 3, 4]
```

In [21]:

```
#The * operator ease the creation of list with similar values:
my_list1 = [1, 2]
my_list1 = my_list1 * 2
print(my_list1)
```

```
[1, 2, 1, 2]
```

In [22]:

```
lst=[1,'CMR']
print(lst[1]*3)
print(lst*3)
```

```
CMRCMRCMR
[1, 'CMR', 1, 'CMR', 1, 'CMR']
```

List Slicing

Slicing uses the symbol : to access to part of a list:

```
    list[first index:last index]
```

In [23]:

```
a = [0, 1, 2, 3, 4, 5]
print(a[2:])
print(a[:2])
print(a[2:-1])
```

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

In [24]:

```
a_list = ['a', 'b', 'c', 'd', 'e', 'f']
print(a_list[1:3])
print(a_list[:4])
print(a_list[3:])
print(a_list[:])
```

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

In [25]:

```
list = [1,2,3,4,5,6,7]
print(list[0:6])
print(list[:])
print(list[2:5])
print(list[1:6:3])
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
[3, 4, 5]
[2, 5]
```

In [5]:

```
#the following example where we will use negative indexing to access
#the elements of the list
list = [1,2,3,4,5]
print(list[-1])
print(list[-3:])
print(list[:-1])
print(list[-3:-1])
print(list[-1:-3])
```

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

In [27]:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

```
['orange', 'kiwi', 'melon']
```

In [28]:

```
#update the values inside the list.
list = [1, 2, 3, 4, 5, 6]
print(list)

list[2] = 10
print(list)

list[1:3] = [89, 78]
print(list)

list[-1] = 25
print(list)
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 10, 4, 5, 6]
[1, 89, 78, 4, 5, 6]
[1, 89, 78, 4, 5, 25]
```

List Built-in functions

Python provides the following built-in functions, which can be used with the lists.

len(list)

It is used to calculate the length of the list.

In [29]:

```
L1 = [1,2,3,4,5,6,7,8]
print(len(L1))
```

```
8
```

In [31]:

```
a = ["bee", "moth", "ant"]
print(len(a))
```

```
3
```

max(list)

It returns the maximum element of the list.

In [32]:

```
L1 = [12,34,26,48,72]
print(max(L1))
```

```
72
```

In [33]:

```
x = ["bee", "moth", "ant"]
print(max(x))

y = ["bee", "moth", "wasp"]
print(max(y))

a = [1, 2, 3, 4,5,6]
b = [1, 2, 3]
print(max(a, b))
```

```
moth
wasp
[1, 2, 3, 4, 5, 6]
```

min(list)

It returns the minimum element of the list.

In [34]:

```
L1 = [12,34,26,48,72]
print(min(L1))
```

```
12
```

In [35]:

```
a = ["bee", "moth", "wasp"]
print(min(a))

a = ["bee", "moth", "ant"]
print(min(a))

a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 4]
print(min(a, b))
```

```
bee
ant
[1, 2, 3, 4]
```

list(seq)

It converts any sequence to the list.

The list() constructor returns a mutable sequence list of elements. The iterable argument is optional. You can provide any sequence or collection (such as a string, list, tuple, set, dictionary, etc). If no argument is supplied, an empty list is returned.

In [36]:

```
str = "Johnson"
s = str.split()
print(s)
print(type(s))
```

```
['Johnson']
<class 'list'>
```

In [1]:

```
print(list())
print(list([]))
print(list(["bee", "moth", "ant"]))
print(list([["bee", "moth"], ["ant"]]))

a = "bee"
print(list(a))

a = ("I", "am", "a", "tuple")
print(list(a))

a = {"I", "am", "a", "set"}
print(list(a))
```

```
[]
[]
['bee', 'moth', 'ant']
[['bee', 'moth'], ['ant']]
['b', 'e', 'e']
['I', 'am', 'a', 'tuple']
['a', 'I', 'set', 'am']
```

range(stop) or range(start, stop[, step])

Represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

It can be used along with list() to return a list of items between a given range.

In [1]:

```
print(range(10))
print(range(1,11))
print(range(1,11,2))
```

```
range(0, 10)
range(1, 11)
range(1, 11, 2)
```

In [4]:

```
list1=list(range(10))
list2=list(range(1,11))
list3=list(range(1,11,2))
print(list1)
print(list2)
print(list3)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 3, 5, 7, 9]
```

List Methods

A list object has a number of methods. These can be grouped arbitrarily into transformations, which change the list, and information, which returns a fact about a list.

append()

The append() method is used to add elements at the end of the list. This method can only add a single element at a time. To add multiple elements, the append() method can be used inside a loop.

In [3]:

```
a = ["bee", "moth"]
print(a)
a.append("ant")
print(a)
```

```
['bee', 'moth']
['bee', 'moth', 'ant']
```

In [8]:

```
a = ["apple", "banana", "cherry"]
b = ["Ford", "BMW", "Volvo"]
print(len(a))
a.append(b)
print(a)
print(len(a))
print(a[3])
print(a[3][1])
```

```
3
['apple', 'banana', 'cherry', ['Ford', 'BMW', 'Volvo']]
4
['Ford', 'BMW', 'Volvo']
BMW
```

In [5]:

```
myList = [1, 2, 3, 'EduCBA', 'makes learning fun!']
myList.append(4)
myList.append(5)
myList.append(6)
print(myList)
for i in range(7, 9):
    myList.append(i)
print(myList)
```

```
[1, 2, 3, 'EduCBA', 'makes learning fun!', 4, 5, 6]
[1, 2, 3, 'EduCBA', 'makes learning fun!', 4, 5, 6, 7, 8]
```

extend()

The extend() method is used to add more than one element at the end of the list. Although it can add more than one element unlike append(), it adds them at the end of the list like append().

In [11]:

```
a = ["bee", "moth"]
print(a)
a.extend(["ant", "fly"])
print(a)
a.append(["mosquito","insect"])
print(a)
```

```
['bee', 'moth']
['bee', 'moth', 'ant', 'fly']
['bee', 'moth', 'ant', 'fly', ['mosquito', 'insect']]
```

In [7]:

```
fruits = ['apple', 'banana', 'cherry']
cars = ['Ford', 'BMW', 'Volvo']
fruits.extend(cars)
print(fruits)
```

```
['apple', 'banana', 'cherry', 'Ford', 'BMW', 'Volvo']
```

In [8]:

```
fruits = ['apple', 'banana', 'cherry']
points = (1, 4, 5, 9)
fruits.extend(points)
print(fruits)
```

```
['apple', 'banana', 'cherry', 1, 4, 5, 9]
```

insert()

The insert() method can add an element at a given position in the list. Thus, unlike append(), it can add elements at any position but like append(), it can add only one element at a time.

This method takes two arguments. The first argument specifies the position and the second argument specifies the element to be inserted.

In [9]:

```
fruits = ['apple', 'banana', 'cherry']
fruits.insert(1, "orange")
print(fruits)
```

```
['apple', 'orange', 'banana', 'cherry']
```

In [10]:

```
myList = [1, 2, 3, 'EduCBA', 'makes learning fun!']
myList.insert(3, 'x')
myList.insert(4, 'y')
myList.insert(5, 'z')
print(myList)
```

```
[1, 2, 3, 'x', 'y', 'z', 'EduCBA', 'makes learning fun!']
```

remove()

The remove() method is used to remove an element from the list. In the case of multiple occurrences of the same element, only the first occurrence is removed.

In [13]:

```
fruits = ['apple', 'banana', 'cherry']
#fruits.remove(1)
fruits.remove("banana")
print(fruits)
```

```
['apple', 'cherry']
```

In [12]:

```
my_list = ['a','b','c','b', 'a']
my_list.remove('a')
print(my_list)
```

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

In [13]:

```
myList = [1, 2, 3, 'EduCBA', 'makes learning fun!']
myList.remove('makes learning fun!')
myList.insert(4, 'makes')
myList.insert(5, 'learning')
myList.insert(6, 'so much fun!')
print(myList)
```

```
[1, 2, 3, 'EduCBA', 'makes', 'learning', 'so much fun!']
```

In [14]:

```
#The del keyword removes the specified index
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

```
['banana', 'cherry']
```

In [15]:

```
#The del keyword can also delete the list completely
thislist = ["apple", "banana", "cherry"]
del thislist
print(thislist)
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[15], line 4
      2 thislist = ["apple", "banana", "cherry"]
      3 del thislist
----> 4 print(thislist)

NameError: name 'thislist' is not defined
```

In [16]:

```
a = ['one', 'two', 'three']
del a[1]
print(a)

alist = ['a', 'b', 'c', 'd', 'e', 'f']
del alist[1:5]
print(alist)
```

```
['one', 'three']
['a', 'f']
```

pop()

The method pop() can remove an element from any position in the list. The parameter supplied to this method is the index of the element to be removed.

In [17]:

```
# No index specified
a = ["bee", "moth", "ant"]
print(a)
a.pop()
print(a)
```

```
['bee', 'moth', 'ant']
['bee', 'moth']
```

In [18]:

```
my_list = ['a','b','c','b', 'a']
my_list.pop()
print(my_list)
```

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

In [19]:

```
# Index specified
a = ["bee", "moth", "ant"]
print(a)
a.pop(1)
print(a)
```

```
['bee', 'moth', 'ant']
['bee', 'ant']
```

In [20]:

```
fruits = ['apple', 'banana', 'cherry']
fruits.pop(1)
print(fruits)
```

```
['apple', 'cherry']
```

In [21]:

```
fruits = ['apple', 'banana', 'cherry']
x = fruits.pop(1)
print(x)
```

```
banana
```

In [22]:

```
myList = [1, 2, 3, 'EduCBA', 'makes learning fun!']
myList.pop(4)
print(myList)
myList.insert(4, 'makes')
myList.insert(5, 'learning')
print(myList)
myList.insert(6, 'so much fun!')
print(myList)
myList.pop(5)
print(myList)
```

```
[1, 2, 3, 'EduCBA']
[1, 2, 3, 'EduCBA', 'makes', 'learning']
[1, 2, 3, 'EduCBA', 'makes', 'learning', 'so much fun!']
[1, 2, 3, 'EduCBA', 'makes', 'so much fun!']
```

reverse()

The reverse() operation is used to reverse the elements of the list. This method modifies the original list.

To reverse a list without modifying the original one, we use the slice operation with negative indices. Specifying negative indices iterates the list from the rear end to the front end of the list.

In [23]:

```
my_list = ['a','b','c','b', 'a',1]
my_list.reverse()
print(my_list)
```

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

In [24]:

```
a = ["bee", "wasp", "moth", "ant"]
a.reverse()
print(a)
```

```
['ant', 'moth', 'wasp', 'bee']
```

In [25]:

```
myList = [1, 2, 3, 'EduCBA', 'makes learning fun!']
print(myList)
print(myList[::-1])
print(myList)
myList.reverse()
print(myList)
```

```
[1, 2, 3, 'EduCBA', 'makes learning fun!']
['makes learning fun!', 'EduCBA', 3, 2, 1]
[1, 2, 3, 'EduCBA', 'makes learning fun!']
['makes learning fun!', 'EduCBA', 3, 2, 1]
```

count()

The function count() returns the number of occurrences of a given element in the list.

In [26]:

```
myList = [1, 2, 3, 'EduCBA', 'makes learning fun!']
print(myList.count(3))
```

```
1
```

In [27]:

```
my_list = ['a','b','c','b', 'a']
my_list.count('b')
```

Out[27]:

```
2
```

In [28]:

```
a = ["bee", "ant", "moth", "ant"]
print(a.count("bee"))
print(a.count("ant"))
print(a.count(""))
```

```
1
2
0
```

index()

The index() method returns the position of the first occurrence of the given element. It takes two optional parameters – the begin index and the end index.

These parameters define the start and end position of the search area on the list. When supplied, the element is searched only in the sub-list bound by the begin and end indices. When not supplied, the element is searched in the whole list.

In [29]:

```
fruits = [4, 55, 64, 32, 16, 32]
x = fruits.index(32)
print(x)
```

```
3
```

In [3]:

```
my_list = ['a','b','c','b', 'a']
my_list.index('b')
```

Out[3]:

```
1
```

In [6]:

```
#my_list.index('b', 2)
#my_list.index('b', 1)
#my_list.index('b', 3)
#my_list.index('b', 4)
my_list.index('b', 1,3)
```

Out[6]:

```
1
```

In [32]:

```
a = ["bee", "ant", "moth", "ant"]
print(a.index("ant"))
print(a.index("ant", 2))
```

```
1
3
```

In [33]:

```
myList = [1, 2, 3, 'EduCBA', 'makes learning fun!']
print(myList.index('EduCBA'))
print(myList)
print(myList.index('EduCBA',0,1))
```

```
3
[1, 2, 3, 'EduCBA', 'makes learning fun!']

---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Cell In[33], line 4
      2 print(myList.index('EduCBA'))
      3 print(myList)
----> 4 print(myList.index('EduCBA',0,1))

ValueError: 'EduCBA' is not in list
```

sort()

The sort method sorts the list in ascending order. This operation can only be performed on homogeneous lists, i.e. lists having elements of similar type.

In [34]:

```
my_list = ['a','b','c','b', 'a']
my_list.sort()
my_list
```

Out[34]:

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

In [35]:

```
yourList = [4, 2, 6, 5, 0, 1]
yourList.sort()
print(yourList)
```

```
[0, 1, 2, 4, 5, 6]
```

In [36]:

```
# sort in the reverse order:
my_list = ['a','b','c','b', 'a']
my_list.sort(reverse=True)
my_list
```

Out[36]:

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

In [37]:

```
a = [3,6,5,2,4,1]
a.sort()
print(a)

a = [3,6,5,2,4,1]
a.sort(reverse=True)
print(a)

a = ["bee", "wasp", "moth", "ant"]
a.sort()
print(a)

a = ["bee", "wasp", "butterfly"]
a.sort(key=len)
print(a)

a = ["bee", "wasp", "butterfly"]
a.sort(key=len, reverse=True)
print(a)
```

```
[1, 2, 3, 4, 5, 6]
[6, 5, 4, 3, 2, 1]
['ant', 'bee', 'moth', 'wasp']
['bee', 'wasp', 'butterfly']
['butterfly', 'wasp', 'bee']
```

In [38]:

```
# A function that returns the length of the value:
def myFunc(e):
  return len(e)

cars = ['Ford', 'Mitsubishi', 'BMW', 'VW']
cars.sort(key=myFunc)
print(cars)
```

```
['VW', 'BMW', 'Ford', 'Mitsubishi']
```

In [39]:

```
def myFunc(e):
  return len(e)

cars = ['Ford', 'Mitsubishi', 'BMW', 'VW']

cars.sort(reverse=True, key=myFunc)
print(cars)
```

```
['Mitsubishi', 'Ford', 'BMW', 'VW']
```

clear()

This function erases all the elements from the list and empties it.

In [40]:

```
myList = [1, 2, 3, 'EduCBA', 'makes learning fun!']
myList1= [1, 2, 3, 'EduCBA', 'makes learning fun!']
myList.clear()
del myList1
print(myList)
print(myList1)
```

```
[]

---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[40], line 6
      4 del myList1
      5 print(myList)
----> 6 print(myList1)

NameError: name 'myList1' is not defined
```

In [41]:

```
a = ["bee", "moth", "ant"]
print(a)
a.clear()
print(a)
```

```
['bee', 'moth', 'ant']
[]
```

copy()

Returns a shallow copy of the list. Equivalent to a[:]

Use the copy() method when you need to update the copy without affecting the original list. If you don't use this method (eg, if you do something like list2 = list1), then any updates you do to list2 will also affect list1.

In [42]:

```
fruits = ['apple', 'banana', 'cherry', 'orange']
x = fruits.copy()
print(x)
print(fruits)
```

```
['apple', 'banana', 'cherry', 'orange']
['apple', 'banana', 'cherry', 'orange']
```

In [43]:

```
# WITHOUT copy()
a = ["bee", "wasp", "moth"]
b = a
b.append("ant")
print(a)
print(b)
```

```
['bee', 'wasp', 'moth', 'ant']
['bee', 'wasp', 'moth', 'ant']
```

In [44]:

```
# WITH copy()
a = ["bee", "wasp", "moth"]
b = a.copy()
b.append("ant")
print(a)
print(b)
```

```
['bee', 'wasp', 'moth']
['bee', 'wasp', 'moth', 'ant']
```

In [45]:

```
#The preceding techniques for copying a list create shallow copies.
#it means that nested objects will not be copied.
a = [1, 2, [3, 4]]
b = a[:]
a[2][0] = 10
print(a)
print(b)
```

```
[1, 2, [10, 4]]
[1, 2, [10, 4]]
```

In [46]:

```
a = [1, 2, [3, 4]]
b = a.copy()
a[2][0] = 10
print(a)
print(b)
```

```
[1, 2, [10, 4]]
[1, 2, [10, 4]]
```

In [47]:

```
#To get around this problem, you must perform a deep copy:
import copy
a = [1, 2, [3, 4]]
b = copy.deepcopy(a)
b[2][0] = 10
print(a)
print(b)
```

```
[1, 2, [3, 4]]
[1, 2, [10, 4]]
```

Cloning Lists

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called cloning, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the slice operator.

Taking any slice of a creates a new list. In this case the slice happens to consist of the whole list.

In [48]:

```
a = [81,82,83]

b = a[:]
print(a == b)
print(a is b)

b[0] = 5

print(a)
print(b)
```

```
True
False
[81, 82, 83]
[5, 82, 83]
```

In [5]:

```
alist = [4,2,8,6,5]
blist = alist * 2
blist[3] = 999
print(alist)
print(blist)
```

```
[4, 2, 8, 6, 5]
[4, 2, 8, 999, 5, 4, 2, 8, 6, 5]
```

The Accumulator Pattern with Lists

We can accumulate values into a list rather than accumulating a single numeric value.

In [50]:

```
nums = [3, 5, 8]
accum = []
for w in nums:
    x = w**2
    accum.append(x)
print(accum)
```

```
[9, 25, 64]
```

In [51]:

```
alist = [4,2,8,6,5]
blist = [ ]
for item in alist:
   blist.append(item+5)
print(blist)
```

```
[9, 7, 13, 11, 10]
```

In [52]:

```
lst= [3,0,9,4,1,7]
new_list=[]
for i in range(len(lst)):
   new_list.append(lst[i]+5)
print(new_list)
```

```
[8, 5, 14, 9, 6, 12]
```

Check if Item Exists

To determine if a specified item is present in a list use the in keyword

In [53]:

```
#Check if "apple" is present in the list or not:
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

```
Yes, 'apple' is in the fruits list
```

In [54]:

```
list = ['larry', 'curly', 'moe']
if 'curlly' in list:
    print ('yes')
```

Loop Through a List

You can loop through the list items by using a for loop

In [55]:

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

```
apple
banana
cherry
```

In [56]:

```
squares = [1, 4, 9, 16]
sum = 0
for num in squares:
    sum += num
print(sum)
```

```
30
```

In [57]:

```
#Using the range Function to Generate a Sequence to Iterate Over
print("This will execute first")

for _ in range(3):
    print("This line will execute three times")
    print("This line will also execute three times")

print("Now we are outside of the for loop!")
```

```
This will execute first
This line will execute three times
This line will also execute three times
This line will execute three times
This line will also execute three times
This line will execute three times
This line will also execute three times
Now we are outside of the for loop!
```

In [58]:

```
#print even numbers from the list range
evens = []
for i in range(10):
    if i % 2 == 0:
        evens.append(i)
print(evens)
```

```
[0, 2, 4, 6, 8]
```

In [59]:

```
#A list comprehension
[print(i) for i in range(10) if i % 2 == 0]
```

```
0
2
4
6
8
```

Out[59]:

```
[None, None, None, None, None]
```

In [60]:

```
#filtering the list
li = [1, 2, 3]
[elem*2 for elem in li if elem>1]
```

Out[60]:

```
[4, 6]
```

Nested List

A list can contain any sort object, even another list (sublist), which in turn can contain sublists themselves, and so on. This is known as nested list.

In [6]:

```
L = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', 'h']
print(len(L))
L[1][1][1]
L[1][3]
```

```
4
```

Out[6]:

```
'ff'
```

In [62]:

```
#access individual items in a nested list using multiple indexes.
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
print(L[2])
print(L[2][2])
print(L[2][2][0])
```

```
['cc', 'dd', ['eee', 'fff']]
['eee', 'fff']
eee
```

You can access a nested list by negative indexing as well.

Negative indexes count backward from the end of the list. So, L[-1] refers to the last item, L[-2] is the second-last, and so on

In [63]:

```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
print(L[-3])
print(L[-3][-1])
print(L[-3][-1][-2])
```

```
['cc', 'dd', ['eee', 'fff']]
['eee', 'fff']
eee
```

In [64]:

```
#change the value of a specific item in a nested list by referring to
#its index number.
L = ['a', ['bb', 'cc'], 'd']
L[1][1] = 0
print(L)
```

```
['a', ['bb', 0], 'd']
```

In [65]:

```
#To add new values to the end of the nested list, use append() method.
L = ['a', ['bb', 'cc'], 'd']
#L.append('YY')
L[1].append('xx')
print(L)
```

```
['a', ['bb', 'cc', 'xx'], 'd']
```

In [66]:

```
#merge one list into another by using extend() method.
L = ['a', ['bb', 'cc'], 'd']
L[1].extend([1,2,3])
print(L)
```

```
['a', ['bb', 'cc', 1, 2, 3], 'd']
```

In [7]:

```
#use pop() method it modifies the list and returns the removed item.
L = ['a', ['bb', 'cc', 'dd'], 'e']
#x = L[1].pop(1)
x = L[1].pop()
print(L)
print(x)
```

```
['a', ['bb', 'cc'], 'e']
dd
```

In [68]:

```
#If you don’t need the removed value, use the del statement.
L = ['a', ['bb', 'cc', 'dd'], 'e']
del L[1][1]
print(L)
```

```
['a', ['bb', 'dd'], 'e']
```

In [9]:

```
#If you’re not sure where the item is in the list, use remove() method
#to delete it by value.
L = ['a', ['bb', 'cc', 'dd'], 'e']
L[1].remove('cc')
#L.remove('cc')
print(L)
```

```
['a', ['bb', 'dd'], 'e']
```

In [70]:

```
#To iterate over the items of a nested list, use simple for loop.
L = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
for list in L:
    for number in list:
        print(number, end=' ')
```

```
1 2 3 4 5 6 7 8 9
```

In [71]:

```
matrix = []

for i in range(5):

    matrix.append([])

    for j in range(5):
        matrix[i].append(j)

print(matrix)
```

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

In [72]:

```
#using nested list comprehension in just one line
matrix = [[j for j in range(5)] for i in range(5)]

print(matrix)
```

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

In [73]:

```
matrix = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]

flatten_matrix = []

for sublist in matrix:
    for val in sublist:
        flatten_matrix.append(val)

print(flatten_matrix)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [74]:

```
matrix = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
flatten_matrix = [val for sublist in matrix for val in sublist]
print(flatten_matrix)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [75]:

```
nested1 = [['a', 'b', 'c'],['d', 'e'],['f', 'g', 'h']]
print(nested1[0])
print(len(nested1))
nested1.append(['i'])
print("-------")
for L in nested1:
    print(L)
```

```
['a', 'b', 'c']
3
-------
['a', 'b', 'c']
['d', 'e']
['f', 'g', 'h']
['i']
```

List Example Programs

In [76]:

```
# print the numbers from 0 through 99
for i in range(100):
    print(i)
```

In [77]:

```
# Write the program to remove the duplicate element of the list.
list1 = [1,2,2,3,55,98,65,65,13,29]
list2 = []
for i in list1:
    if i not in list2:
        list2.append(i)
print(list2)
```

```
[1, 2, 3, 55, 98, 65, 13, 29]
```

In [78]:

```
# Write the program to find the lists consist of at least one common element.
list1 = [1,2,3,4,5,6]
list2 = [7,8,9,2,10]
for x in list1:
    for y in list2:
        if x == y:
            print("The common element is:",x)
```

```
The common element is: 2
```

In [79]:

```
# Write a program to find the sum of the element in the list
list1 = [3,4,5,9,10,12,24]
sum = 0
for i in list1:
    sum = sum+i
print("The sum is:",sum)
```

```
The sum is: 67
```

In [80]:

```
#first char of colors list of elements
colors = ["Red", "Orange", "Yellow", "Green", "Blue", "Indigo", "Violet"]
initials = []

for color in colors:
    initials.append(color[0])

print(initials)
```

```
['R', 'O', 'Y', 'G', 'B', 'I', 'V']
```

In [81]:

```
# first char of each word
rest = ["sleep", 'dormir', 'dormire', "slaap", 'sen', 'yuxu', 'yanam']
let = ''
for phrase in rest:
    let += phrase[0]
print("first char of each word:",let)
```

```
first char of each word: sddssyy
```

In [82]:

```
#list of planets which is the lenth<6 char
planets = [['Mercury', 'Venus', 'Earth'], ['Mars', 'Jupiter', 'Saturn'], ['Uranus', 'Nep

flatten_planets = []

for sublist in planets:
    for planet in sublist:

        if len(planet) < 6:
            flatten_planets.append(planet)

print(flatten_planets)
```

```
['Venus', 'Earth', 'Mars', 'Pluto']
```

In [83]:

```
planets = [['Mercury', 'Venus', 'Earth'], ['Mars', 'Jupiter', 'Saturn'], ['Uranus', 'Nep
flatten_planets = [planet for sublist in planets for planet in sublist if len(planet) <
print(flatten_planets)
```

```
['Venus', 'Earth', 'Mars', 'Pluto']
```

In [84]:

```
# split and join
sent = "The mall has excellent sales right now."
print(sent)
wrds = sent.split()
print(wrds)
wrds[1] = 'store'
new_sent = " ".join(wrds)
print(new_sent)
```

```
The mall has excellent sales right now.
['The', 'mall', 'has', 'excellent', 'sales', 'right', 'now.']
The store has excellent sales right now.
```

In [85]:

```
#accumulation
byzo = 'hello world!'
c = 0
for x in byzo:
    z = x + "!"
    print(z)
    c = c + 1
print("string length",c)
```

```
h!
e!
l!
l!
o!
 !
w!
o!
r!
l!
d!
!!
string length 12
```

In [86]:

```
# total chars
cawdra = ['candy', 'daisy', 'pear', 'peach', 'gem', 'crown']
t = 0
for elem in cawdra:
    t = t + len(elem)
print("total chars:",t)
```

```
total chars: 27
```

In [87]:

```
# find no of strings in the list
lst = ['plan', 'answer', 5, 9.29, 'order, items','ram', [4]]
s = 0
for item in lst:
    if type(item) == type("string"):
        s = s + 1
print("Number of strings in the list:",s)
```

```
Number of strings in the list: 4
```

In [88]:

```
# For each word in the list verbs, add an -ing ending.
verbs = ["kayak", "cry", "walk", "eat", "drink", "fly"]
new=[]
for word in verbs:
    w=word+'ing'
    new.append(w)
print(new)
```

```
['kayaking', 'crying', 'walking', 'eating', 'drinking', 'flying']
```

In [89]:

```
d = {}
for i in range(1,16):
    d.update({i:i*i})
print("The Dictionary of squares is \n", d)
```

```
The Dictionary of squares is
 {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100, 11:
121, 12: 144, 13: 169, 14: 196, 15: 225}
```

In [90]:

```
# program to count no of vowels and consonants in given string
str = "cmrabefg"
v = ['a','e','i','o','u']
ac,ac1 = 0,0
for i in range(0, len(str)):
    if str[i] in v:
        ac = ac + 1
    else:
        ac1 = ac1 + 1

print("No of Vowels in given string: ", ac)
print("No of Consonants in given string: ", ac1)
```

```
No of Vowels in given string:  2
No of Consonants in given string:  6
```

Python Tuples

In Python, tuples are part of the standard language. This is a data structure very similar to the list data structure. The main difference being that tuple manipulation are faster than list because tuples are immutable.

A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.

A tuple, like a list, is a sequence of items of any type. The printed representation of a tuple is a comma-separated sequence of values, enclosed in parentheses. In other words, the representation is just like lists, except with parentheses () instead of square brackets [].

Tuples are a lot like lists:

- Tuples are ordered

- Tuples maintains a left-to-right positional ordering among the items they contain.
- Accessed by index – Items in a tuple can be accessed using an index.
- Tuples can contain any sort of object – It can be numbers, strings, lists and even other tuples.

except:

- Tuples are immutable – you can't add, delete, or change items after the tuple is defined.

Create a Tuple

Creating a tuple is just like creating a list, except that you use regular brackets instead of square brackets

In [9]:

```
weekdays = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
print(weekdays)
```

```
('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
```

In [10]:

```
#the brackets are optional
weekdays = "Monday", "Tuesday", "Wednesday", "Thursday", "Friday"
print(weekdays)
```

```
('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
```

In [11]:

```
type1 = (1, 3, 4, 5, 'test')
print(type1)
```

```
(1, 3, 4, 5, 'test')
```

In [157]:

```
#If you want to create a tuple with a single element, you must use the comma:
weekday = ('Monday',)
print(weekday)
print(type(weekday))
```

```
('Monday',)
<class 'tuple'>
```

In [13]:

```
a = ("dog")
b = ("dog",)
print(type(a))
print(type(b))
```

```
<class 'str'>
<class 'tuple'>
```

In [14]:

```
t = (5,)
print(type(t))

x = (5)
print(type(x))
```

```
<class 'tuple'>
<class 'int'>
```

In [15]:

```
#Creating an empty Tuple
Tuple1 = ()
print("Initial empty Tuple: ")
print (Tuple1)

#Creatting a Tuple with the use of string
Tuple1 = ('CMR TC', 'CSE')
print("\nTuple with the use of String: ")
print(Tuple1)

# Creating a Tuple with the use of list
list1 = [1, 2, 4, 5, 6]
print("\nTuple using List: ")
print(tuple(list1))

#Creating a Tuple with the use of built-in function
Tuple1 = tuple('CMR TC')
print("\nTuple with the use of function: ")
print(Tuple1)
```

```
Initial empty Tuple:
()

Tuple with the use of String:
('CMR TC', 'CSE')

Tuple using List:
(1, 2, 4, 5, 6)

Tuple with the use of function:
('C', 'M', 'R', ' ', 'T', 'C')
```

In [16]:

```
#Creating a Tuple with Mixed Datatype
Tuple1 = (5, 'Welcome', 7, 'CMR TC')
print("\nTuple with Mixed Datatypes: ")
print(Tuple1)

#Creating a Tuple with nested tuples
Tuple1 = (0, 1, 2, 3)
Tuple2 = ('python', 'CSE')
Tuple3 = (Tuple1, Tuple2)
print("\nTuple with nested tuples: ")
print(Tuple3)

#Creating a Tuple with repetition
Tuple1 = ('CMR TC',) * 3
print("\nTuple with repetition: ")
print(Tuple1)

#Creating a Tuple with the use of loop
Tuple1 = ('CMR TC')
n = 5
print("\nTuple with a loop")
for i in range(int(n)):
    Tuple1 = (Tuple1,)
print(Tuple1)
```

```
Tuple with Mixed Datatypes:
(5, 'Welcome', 7, 'CMR TC')

Tuple with nested tuples:
((0, 1, 2, 3), ('python', 'CSE'))

Tuple with repetition:
('CMR TC', 'CMR TC', 'CMR TC')

Tuple with a loop
(((((('CMR TC',),),),),),)
```

Tuple Packing & Unpacking

When working with multiple values or multiple variable names, the Python interpreter does some automatic packing and unpacking to and from tuples, which allows some simplifications in the code you write.

Tuple Packing

When a tuple is created, the items in the tuple are packed together into the object.

Wherever python expects a single value, if multiple expressions are provided, separated by commas, they are automatically packed into a tuple.

For example, we can omit the parentheses when assigning a tuple of values to a single variable.

In [17]:

```
T = 'red', 'green', 'blue', 'cyan'
print(T)
```

```
('red', 'green', 'blue', 'cyan')
```

In [18]:

```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")
# or equivalently
julia = "Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia"
print(julia[4])
```

```
2009
```

Tuple Unpacking

When a packed tuple is assigned to a new tuple, the individual items are unpacked (assigned to the items of a new tuple).

Python has a very powerful tuple assignment feature that allows a tuple of variable names on the left of an assignment statement to be assigned values from a tuple on the right of the assignment. Another way to think of this is that the tuple of values is unpacked into the variable names.

In [19]:

```
T = ('red', 'green', 'blue', 'cyan')
(a, b, c, d) = T

print(a)

print(b)

print(c)

print(d)
```

```
red
green
blue
cyan
```

In [20]:

```
julia = "Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia"

name, surname, birth_year, movie, movie_year, profession, birth_place = julia
print(name)
```

```
Julia
```

In [21]:

```
person = ("Salman", '5 ft', "Actor")
(name, height, profession) = person
print(name)
print(height)
print(profession)
```

```
Salman
5 ft
Actor
```

In [22]:

```
#Naturally, the number of variables on the left and the number of values
#on the right have to be the same.
(a, b, c, d) = (1, 2, 3)
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Cell In[22], line 3
      1 #Naturally, the number of variables on the left and the number of
values
      2 #on the right have to be the same.
----> 3 (a, b, c, d) = (1, 2, 3)

ValueError: not enough values to unpack (expected 4, got 3)
```

In [23]:

```
#swapping with temp
a = 1
b = 2
temp = a
a = b
b = temp
print(a, b, temp)
```

```
2 1 1
```

In [24]:

```
#Tuple assignment solves this problem neatly
a = 1
b = 2
(a, b) = (b, a)
print(a, b)
```

```
2 1
```

In [25]:

```
#Multiple assignment with unpacking is particularly useful when you iterate
#through a list of tuples.
authors = [('Paul', 'Resnick'), ('Brad', 'Miller'), ('Lauren', 'Murphy')]
for first_name, last_name in authors:
    print("first name:", first_name, "last name:", last_name)
```

```
first name: Paul last name: Resnick
first name: Brad last name: Miller
first name: Lauren last name: Murphy
```

Accessing Items in a Tuple

One can access the elements of a tuple in multiple ways, such as indexing, negative indexing, range, etc.

In [26]:

```
access_tuple = ('a', 'b', 1, 3, [5, 'x', 'y', 'z'])
print(len(access_tuple))
print(access_tuple[0])
print(access_tuple[4][1])
```

```
5
a
x
```

In [27]:

```
#We can find the use of negative indexing on tuples.
access_tuple = ('a', 'b', 1, 3)
print(access_tuple[-1])
```

```
3
```

In [28]:

```
prices = (1.99, 2.00, 5.50, 20.95, 100.98)
print(prices[0])
print(prices[-1])
print(prices[3-5])
```

```
1.99
100.98
20.95
```

In [29]:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

```
('orange', 'kiwi', 'melon')
```

In [30]:

```
#We can find a range of tuples.
access_tuple = ('a', 'b', 1, 3, 5, 'x', 'y', 'z')
print(access_tuple[2:5])
```

```
(1, 3, 5)
```

In [31]:

```
#Access a List Item within a Tuple
t = (101, 202, ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"])
print(t[2][1])
```

```
Tuesday
```

In [32]:

```
#Accessing Tuple with Indexing
Tuple1 = tuple("CMR TC")
print("\nFirst element of Tuple: ")
print(Tuple1[1])

#Tuple unpacking
Tuple1 = ("CMR TC", "For", "CSE")

#This line unpack values of Tuple1
a, b, c = Tuple1
print("\nValues after unpacking: ")
print(a)
print(b)
print(c)
```

```
First element of Tuple:
M

Values after unpacking:
CMR TC
For
CSE
```

Tuple Concatenation & Repetition

Concatenation of tuple is the process of joining of two or more Tuples. Concatenation is done by the use of '+' operator. Concatenation of tuples is done always from the end of the original tuple. Other arithmetic operations do not apply on Tuples.

Tuples can be joined using the concatenation operator + or Replication operator *

In [33]:

```
# Concatenaton of tuples
Tuple1 = (0, 1, 2, 3)
Tuple2 = ("CMR TC", "For", "CSE")

Tuple3 = Tuple1 + Tuple2

# Printing first Tuple
print("Tuple 1: ")
print(Tuple1)

# Printing Second Tuple
print("\nTuple2: ")
print(Tuple2)

# Printing Final Tuple
print("\nTuples after Concatenaton: ")
print(Tuple3)
```

```
Tuple 1:
(0, 1, 2, 3)

Tuple2:
('CMR TC', 'For', 'CSE')

Tuples after Concatenaton:
(0, 1, 2, 3, 'CMR TC', 'For', 'CSE')
```

In [34]:

```
weekdays = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
weekenddays = ("Saturday", "Sunday")
alldays = weekdays + weekenddays
print(weekdays)
print(weekenddays)
print(alldays)
```

```
('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
('Saturday', 'Sunday')
('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sund
ay')
```

In [35]:

```
T = ('red', 'green', 'blue') + (1, 2, 3)
print(T)

T = ('red',) * 3
print(T)
```

```
('red', 'green', 'blue', 1, 2, 3)
('red', 'red', 'red')
```

Update a Tuple

tuples are immutable. This means that items defined in a tuple cannot be changed once the tuple has been created.

As with strings, if we try to use item assignment to modify one of the elements of a tuple, we get an error. In fact, that's the key difference between lists and tuples: tuples are like immutable lists. None of the operations on lists that mutate them are available for tuples. Once a tuple is created, it can't be changed.

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

You can convert the tuple into a list, change the list, and convert the list back into a tuple.

In [36]:

```
Tuple1 = (1, 3, 4, 'test', 'red')
Tuple1[1] =4
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[36], line 2
      1 Tuple1 = (1, 3, 4, 'test', 'red')
----> 2 Tuple1[1] =4

TypeError: 'tuple' object does not support item assignment
```

In [37]:

```
#if the item in tuple itself is a mutable data type like list,
# its nested items can be changed.
tuple1 = (1, 2, 3, [4, 5])
tuple1[3][0] = 7
print(tuple1)
```

```
(1, 2, 3, [7, 5])
```

In [1]:

```
#Convert the tuple into a list to be able to change it
x = ("apple", "banana", "cherry")
print(x)
y = list(x)
print(y)
y[1] = "kiwi"
print(y)
x = tuple(y)
print(x)
```

```
('apple', 'banana', 'cherry')
['apple', 'banana', 'cherry']
['apple', 'kiwi', 'cherry']
('apple', 'kiwi', 'cherry')
```

In [2]:

```
#Although you can't change tuple items, you can change
# list items within a tuple.
t = (101, 202, ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"])
print(t)
t[2][2] = "Humpday"
print(t)
```

```
(101, 202, ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'])
(101, 202, ['Monday', 'Tuesday', 'Humpday', 'Thursday', 'Friday'])
```

Deleting a Tuple

we cannot change the items in a tuple. which also suggests that we cannot remove items from the tuple.

In [3]:

```
Tuple1 = (1, 3, 4, 'test', 'red')
del (Tuple1[1])
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[3], line 2
      1 Tuple1 = (1, 3, 4, 'test', 'red')
----> 2 del (Tuple1[1])

TypeError: 'tuple' object doesn't support item deletion
```

In [4]:

```
#But one can delete a tuple by using the keyword del( ) with a tuple.
Tuple1 = (1, 3, 4, 'test', 'red')
del (Tuple1)
print (Tuple1)
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[4], line 4
      2 Tuple1 = (1, 3, 4, 'test', 'red')
      3 del (Tuple1)
----> 4 print (Tuple1)

NameError: name 'Tuple1' is not defined
```

Tuple Slicing

Slicing of a Tuple is done to fetch a specific range or slice of sub-elements from a Tuple. Slicing can also be done to lists and arrays. Indexing in a list results to fetching a single element whereas Slicing allows to fetch a set of elements.

As tuples are immutable but we can take slices of one tuple and can place those slices in another tuple.

In [5]:

```
Tuple1 = (1, 3, 4, 'test', 'red')
Sliced=(Tuple1[2:])
print (Sliced)
```

```
(4, 'test', 'red')
```

In [6]:

```
tuple = (1,2,3,4,5,6,7,8,9,10)
print(tuple[1:])
print(tuple[:4])
print(tuple[1:5])
print(tuple[0:9:2])
```

```
(2, 3, 4, 5, 6, 7, 8, 9, 10)
(1, 2, 3, 4)
(2, 3, 4, 5)
(1, 3, 5, 7, 9)
```

In [7]:

```
#The elements from left to right are traversed using the negative indexing
tuple1 = (1, 2, 3, 4, 5)
print(tuple1[-1])
print(tuple1[-4])
print(tuple1[-3:-1])
print(tuple1[:-1])
print(tuple1[-2:])
```

```
5
2
(3, 4)
(1, 2, 3, 4)
(4, 5)
```

In [8]:

```
T = ('a', 'b', 'c', 'd', 'e', 'f')

print(T[2:5])

print(T[0:2])

print(T[3:-1])
```

```
('c', 'd', 'e')
('a', 'b')
('d', 'e')
```

In [9]:

```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")
print(julia[2])
print(julia[2:6])

print(len(julia))

julia = julia[:3] + ("Eat Pray Love", 2010) + julia[5:]
print(julia)
```

```
1967
(1967, 'Duplicity', 2009, 'Actress')
7
('Julia', 'Roberts', 1967, 'Eat Pray Love', 2010, 'Actress', 'Atlanta, Geo
rgia')
```

Nesting Operation on Tuples

Nesting simply means the place or store one or more inside the other.

In [10]:

```
Tuple1 = (1, 3, 4)
Tuple2 = ('red', 'green', 'blue')
Tuple3 = (Tuple1, Tuple2)
print (Tuple3)
```

```
((1, 3, 4), ('red', 'green', 'blue'))
```

Check if Item Exists

To determine if a specified item is present in a tuple use the in keyword:

In [1]:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

```
Yes, 'apple' is in the fruits tuple
```

In [2]:

```
Tuple1 = (1, 3, 4, 'test', 'red')
print (1 in Tuple1)
print (5 in Tuple1)
```

```
True
False
```

In [3]:

```
T = ('red', 'green', 'blue')
if 'red' in T:
    print('yes')

if 'yellow' not in T:
    print('yes')
```

```
yes
yes
```

Use of Tuples as keys in Dictionaries

We know that tuples are hash-able (remains the same thorough out its life span), and the list is not. We should use tuples as the keys to create a composite key and use the same in a dictionary.

In [4]:

```
tuplekey = {}
tuplekey[('blue', 'sky')] = 'Good'
tuplekey[('red','blood')] = 'Bad'
print(tuplekey)
```

```
{('blue', 'sky'): 'Good', ('red', 'blood'): 'Bad'}
```

Iterate through a Tuple

We can form an iterating loop with tuples.

In [5]:

```
my_tuple = ("red", "Orange", "Green", "Blue")
for colour in my_tuple:
    print(colour)
```

```
red
Orange
Green
Blue
```

Tuples as Return Values

Functions can return tuples as return values.

In each case, a function (which can only return a single value), can create a single tuple holding multiple elements.

In [6]:

```
def circleInfo(r):
    c = 2 * 3.14159 * r
    a = 3.14159 * r * r
    return (c, a)

print(circleInfo(10))
```

```
(62.8318, 314.159)
```

In [7]:

```
def circleInfo(r):
    c = 2 * 3.14159 * r
    a = 3.14159 * r * r
    return c, a

print(circleInfo(10))
```

```
(62.8318, 314.159)
```

In [8]:

```
def circleInfo(r):
    c = 2 * 3.14159 * r
    a = 3.14159 * r * r
    return c, a

print(circleInfo(10))

circumference, area = circleInfo(10)
print(circumference)
print(area)

circumference_two, area_two = circleInfo(45)
print(circumference_two)
print(area_two)
```

```
(62.8318, 314.159)
62.8318
314.159
282.74309999999997
6361.719749999999
```

In [9]:

```
#Python even provides a way to pass a single tuple to a function and have
#it be unpacked for assignment to the named parameters.
def add(x, y):
    return x + y

print(add(3, 4))
z = (5, 4)
print(add(*z))
```

```
7
9
```

Tuple Methods

Python has a set of built-in methods that you can call on tuple objects.

count() Method

Use count() method to find the number of times the given item appears in the tuple. tuple.count(item)

In [10]:

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.count(5)
print(x)
```

```
2
```

In [11]:

```
T = ('red', 'green', 'blue')
print(T.count('red'))
```

1

In [12]:

```
T = (1, 9, 7, 3, 9, 1, 9, 2)
print(T.count(9))
```

3

index() Method

The index() method finds the first occurrence of the specified value.

The index() method raises an exception if the value is not found. tuple.index(item,start,end)

In [13]:

```
T = ('red', 'green', 'blue', 'yellow')
print(T.index('green'))
```

1

In [14]:

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.index(7)
print(x)
```

2

In [15]:

```
T = ('a','b','c','d','e','f','a','b','c','d','e','f')
print(T.index('c'))
```

2

In [16]:

```
# Find 'c' starting a position 5
T = ('a','b','c','d','e','f','a','b','c','d','e','f')
print(T.index('c',5))
```

8

In [17]:

```
# Find 'c' in between 5 & 10
T = ('a','b','c','d','e','f','a','b','c','d','e','f')
print(T.index('c',5,10))
```

```
8
```

In [18]:

```
#index() method raises a 'ValueError' if specified item is not found in
#the tuple.
T = ('a','b','c','d','e','f','a','b','c','d','e','f')
print(T.index('x'))
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Cell In[18], line 4
      1 #index() method raises a 'ValueError' if specified item is not fou
nd in
      2 #the tuple.
      3 T = ('a','b','c','d','e','f','a','b','c','d','e','f')
----> 4 print(T.index('x'))

ValueError: tuple.index(x): x not in tuple
```

In [19]:

```
T = ('a','b','c','d','e','f','a','b','c','d','e','f')
print(T.index('c',4,7))
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Cell In[19], line 2
      1 T = ('a','b','c','d','e','f','a','b','c','d','e','f')
----> 2 print(T.index('c',4,7))

ValueError: tuple.index(x): x not in tuple
```

In [20]:

```
#To avoid such exception, you can check if item exists in a tuple,
#using in operator inside if statement.
T = ('a','b','c','d','e','f','a','b','c','d','e','f')
if 'x' in T:
    print(T.index('x'))
```

Inbuilt functions for Tuples

Python has some built-in functions which can be performed directly on the tuples. For e.g., all(), any(),

all()

Determines whether all items in an iterable are True

The all() function returns True if all items in an iterable are True. Otherwise, it returns False. all(iterable)

In Python, all the following values are considered False:

- Constants defined to be false: None and False.
- Zero of any numeric type: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
- Empty sequences and collections: '', (), [], {}, set(), range(0)

In [21]:

```
L = (1, 1, 1)
print(all(L))
```

```
True
```

In [22]:

```
L = [0, 1, 1]
print(all(L))
```

```
False
```

In [23]:

```
L = [True, 0, 1]
print(all(L))

T = ('', 'red', 'green')
print(all(T))

S = {0j, 3+4j}
print(all(S))
```

```
False
False
False
```

In [24]:

```
#If the iterable is empty, the function returns True.
L = []
print(all(L))

L = [[], []]
print(all(L))
```

```
True
False
```

any()

Determines whether any item in an iterable is True

The any() function returns True if any item in an iterable is True. Otherwise, it returns False. any(iterable)

In Python, all the following values are considered False.

- Constants defined to be false: None and False.
- Zero of any numeric type: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
- Empty sequences and collections: '', (), [], {}, set(), range(0)

In [25]:

```
L = [0, 0, 0]
print(any(L))

L = [0, 1, 0]
print(any(L))
```

```
False
True
```

In [26]:

```
L = [False, 0, 1]
print(any(L))

T = ('', [], 'green')
print(any(T))

S = {0j, 3+4j, 0.0}
print(any(S))
```

```
True
True
True
```

In [27]:

```
#If the iterable is empty, the function returns False.
L = []
print(any(L))
```

```
False
```

enumerate()

Adds a counter to an iterable

The enumerate() function adds a counter to an iterable and returns it as an enumerate object.

By default, enumerate() starts counting at 0 but if you add a second argument start, it'll start from that number instead. enumerate(iterable,start)

In [28]:

```
#how to iterate through the indexes of a sequence, and thus enumerate
#the items and their positions in the sequence.
fruits = ['apple', 'pear', 'apricot', 'cherry', 'peach']
for n in range(len(fruits)):
    print(n, fruits[n])
```

```
0 apple
1 pear
2 apricot
3 cherry
4 peach
```

In [29]:

```
# with enumerate() function
fruits = ['apple', 'pear', 'apricot', 'cherry', 'peach']
for item in enumerate(fruits):
    print(item[0], item[1])
```

```
0 apple
1 pear
2 apricot
3 cherry
4 peach
```

In [30]:

```
fruits = ['apple', 'pear', 'apricot', 'cherry', 'peach']
for idx, fruit in enumerate(fruits):
    print(idx, fruit)
```

```
0 apple
1 pear
2 apricot
3 cherry
4 peach
```

In [31]:

```
L = ['red', 'green', 'blue']
x = list(enumerate(L))
print(x)
```

```
[(0, 'red'), (1, 'green'), (2, 'blue')]
```

In [32]:

```
L = ['red', 'green', 'blue']
x = list(enumerate(L, 10))
print(x)
```

```
[(10, 'red'), (11, 'green'), (12, 'blue')]
```

In [33]:

```
#When you iterate an enumerate object, you get a tuple
#containing (counter, item)
L = ['red', 'green', 'blue']
for pair in enumerate(L):
    print(pair)
```

```
(0, 'red')
(1, 'green')
(2, 'blue')
```

len()

Returns the number of items of an object

The len() function returns the number of items of an object.

The object may be a sequence (such as a string, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

```
len(object)
```

In [35]:

```
T = ('red', 'green', 'blue')
x = len(T)
print(x)
```

```
3
```

max(tuple)

It gives the maximum value from the tuple, here the condition is tuple should not contain string values.

The max() function can find -the largest of two or more values (such as numbers, strings etc.) -the largest item in an iterable (such as list, tuple etc.)

With optional key parameter, you can specify custom comparison criteria to find maximum value.

```
max(iterable,key,default)
```

In [36]:

```
x = max(10, 20, 30)
print(x)
```

```
30
```

In [37]:

```
# Specify default value '0'
L = []
x = max(L, default='0')
print(x)
```

```
0
```

In [38]:

```
#You can also pass in your own custom function as the key function.
# Find out who is the oldest student
def myFunc(e):
  return e[1]

L = [('Sam', 35),
    ('Tom', 25),
    ('Bob', 30)]

x = max(L, key=myFunc)
print(x)
```

```
('Sam', 35)
```

min(tuple)

It gives the minimum value from the tuple, here the condition is tuple should not contain string values.

The min() function can find -the smallest of two or more values (such as numbers, strings etc.) -the smallest item in an iterable (such as list, tuple etc.)

With optional key parameter, you can specify custom comparison criteria to find minimum value.
min(iterable,key,default)

In [39]:

```
tuple1 = (1, 2, 3, 6)
print(min(tuple1))
```

```
1
```

In [40]:

```
# Specify default value '0'
L = []
x = min(L, default='0')
print(x)
```

```
0
```

In [41]:

```
# Find out who is the youngest student
def myFunc(e):
  return e[1]

L = [('Sam', 35),
    ('Tom', 25),
    ('Bob', 30)]

x = min(L, key=myFunc)
print(x)
```

```
('Tom', 25)
```

sum(tuple):

The elements in a tuple should be integers only for this operation. The sum will provide a summation of all the elements in the tuple.

The sum() function sums the items of an iterable and returns the total.

If you specify an optional parameter start, it will be added to the final sum. sum(iterable,start)

In [42]:

```
tuple1 = (1, 2, 3, 6)
print(sum(tuple1))
```

```
12
```

In [43]:

```
# Start with '10' and add all items in a list
L = [1, 2, 3, 4, 5]
x = sum(L, 10)
print(x)
```

```
25
```

sorted(tuple):

If the elements are not arranged in order we can use the sorted inbuilt function.

The sorted() method sorts the items of any iterable

You can optionally specify parameters for sort customization like sorting order and sorting criteria. sorted(iterable,key,reverse)

In [44]:

```
tuple2= (1, 4, 3, 2, 1, 7, 6)
print(sorted(tuple2))
```

```
[1, 1, 2, 3, 4, 6, 7]
```

In [45]:

```
T = ('cc', 'aa', 'dd', 'bb')
print(sorted(T))
```

```
['aa', 'bb', 'cc', 'dd']
```

In [46]:

```
T = ('cc', 'aa', 'dd', 'bb')
tmp = list(T)
tmp.sort()
print(tmp)
```

```
['aa', 'bb', 'cc', 'dd']
```

In [47]:

```
# strings are sorted alphabetically
L = ['red', 'green', 'blue', 'orange']
x = sorted(L)
print(x)

# numbers are sorted numerically
L = [42, 99, 1, 12]
x = sorted(L)
print(x)
```

```
['blue', 'green', 'orange', 'red']
[1, 12, 42, 99]
```

In [48]:

```
# Sort a tuple
L = ('cc', 'aa', 'dd', 'bb')
x = sorted(L)
print(x)

D = {'Bob':30, 'Sam':25, 'Max':35, 'Tom':20}
y = sorted(D)
print(y)
```

```
['aa', 'bb', 'cc', 'dd']
['Bob', 'Max', 'Sam', 'Tom']
```

In [49]:

```
#You can also sort an iterable in reverse order by setting reverse to true.
L = ['cc', 'aa', 'dd', 'bb']
x = sorted(L, reverse=True)
print(x)
```

```
['dd', 'cc', 'bb', 'aa']
```

In [50]:

```
#key=len (the built-in len() function) sorts the strings by length,
#from shortest to longest.
L = ['orange', 'red','flo', 'green', 'blue']
x = sorted(L, key=len)
print(x)
```

```
['red', 'flo', 'blue', 'green', 'orange']
```

Example

In [51]:

```
# Split an email address into a user name and a domain
addr = 'bob@python.org'
user, domain = addr.split('@')

print(user)

print(domain)
```

```
bob
python.org
```

Python Dictionary

Dictionary is an indexed collection of items where items are stored in form of key value pairs. And keys are unique whereas values can be changed.

Dictionaries are optimized for faster retrieval when the key is known, values can be of any type but keys are immutable and can be only string, numbers or tuples and dictionaries are written inside curly braces.

A dictionary is a sequence of items. Each item is a pair made of a key and a value. Dictionaries are not sorted. You can access to the list of keys or values independently.

e.g. dict = {key1:value1, key2:value2, ... }.

The "empty dict" is just an empty pair of curly braces {}.

Create a Dictionary

In [52]:

```
d = {"Key1": "Value1", "Key2": "Value2"}
d
```

Out[52]:

```
{'Key1': 'Value1', 'Key2': 'Value2'}
```

In [53]:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
print(type(thisdict))
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
<class 'dict'>
```

In [54]:

```
Dict = {'Name': 'CMR TC', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

```
Dictionary with the use of Mixed Keys:
{'Name': 'CMR TC', 1: [1, 2, 3, 4]}
```

Accessing of Dictionary

Dictionaries are not sorted. You can access to the list of keys or values independently.

In [55]:

```
d = {'first':'string value', 'second':[1,2]}
d.keys()
```

Out[55]:

```
dict_keys(['first', 'second'])
```

In [56]:

```
d.values()
```

Out[56]:

```
dict_values(['string value', [1, 2]])
```

In [57]:

```
# You can access to the value of a given key as follows:
d['first']
```

Out[57]:

```
'string value'
```

In [58]:

```
planet_size = {"Earth": 40075, "Saturn": 378675, "Jupiter": 439264}
print(planet_size["Earth"])
print(planet_size.get("Saturn"))
```

```
40075
378675
```

Adding Dictionary values

The dictionary is a mutable data type, and its values can be updated by using the specific keys. The value can be updated along with key Dict[key] = value. The update() method is also used to update an existing value.

Note: If the key-value already present in the dictionary, the value gets updated. Otherwise, the new keys added in the dictionary.

In [59]:

```
dict = {}
dict['one'] = 'uno'
dict['two'] = 'dos'
dict['three'] = 'tres'
print(dict)
```

```
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

In [60]:

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

Dict[0] = 'CMR TC'
Dict[2] = 'CSE'
Dict[3] = 1
print("\nDictionary after adding 3 elements: ")
print(Dict)
```

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'CMR TC', 2: 'CSE', 3: 1}
```

In [61]:

```
# Adding set of values to a single Key
Dict['Value_set'] = 2, 3, 4
print("\nDictionary after adding 3 elements: ")
print(Dict)
```

```
Dictionary after adding 3 elements:
{0: 'CMR TC', 2: 'CSE', 3: 1, 'Value_set': (2, 3, 4)}
```

In [62]:

```
# Updating existing Key's Value
Dict[2] = 'Welcome'
print("\nUpdated key value: ")
print(Dict)
```

```
Updated key value:
{0: 'CMR TC', 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4)}
```

In [63]:

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
inventory['pears'] = 0
inventory
```

Out[63]:

```
{'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 0}
```

In [64]:

```
# Adding Nested Key value to Dictionary
Dict[5] = {'Nested' :{'1' : 'Life', '2' : 'CSE'}}
print("\nAdding a Nested Key: ")
print(Dict)
```

```
Adding a Nested Key:
{0: 'CMR TC', 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4), 5: {'Nested':
{'1': 'Life', '2': 'CSE'}}}
```

In [65]:

```
mydict = {"cat":12, "dog":6, "elephant":23}
mydict["mouse"] = mydict["cat"] + mydict["dog"]
print(mydict["mouse"])
mydict
```

```
18
```

Out[65]:

```
{'cat': 12, 'dog': 6, 'elephant': 23, 'mouse': 18}
```

Removing Elements from Dictionary

In Python Dictionary, deletion of keys can be done by using the del keyword. Using del keyword, specific values from a dictionary as well as whole dictionary can be deleted.

Items in a Nested dictionary can also be deleted by using del keyword and providing specific nested key and particular key to be deleted from that nested Dictionary.

In [66]:

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
del inventory['pears']
inventory
```

Out[66]:

```
{'apples': 430, 'bananas': 312, 'oranges': 525}
```

In [67]:

```
planet_size = {"Earth": 40075, "Saturn": 378675, "Jupiter": 439264}
print(planet_size)
del planet_size["Jupiter"]
print(planet_size)
```

```
{'Earth': 40075, 'Saturn': 378675, 'Jupiter': 439264}
{'Earth': 40075, 'Saturn': 378675}
```

In [68]:

```
# You can use the del keyword to delete the whole dictionary
user = {"Name": "Christine", "Age": 23}
del user
user
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[68], line 4
      2 user = {"Name": "Christine", "Age": 23}
      3 del user
----> 4 user

NameError: name 'user' is not defined
```

In [69]:

```
# Initial Dictionary
Dict = { 5 : 'Welcome', 6 : 'To', 7 : 'CMR TC',
        'A' : {1 : 'CMR TC', 2 : 'For', 3 : 'CMR TC'},
        'B' : {1 : 'CMR TC', 2 : 'Life'}}
print("Initial Dictionary: ")
print(Dict)

# Deleting a Key value
del Dict[6]
print("\nDeleting a specific key: ")
print(Dict)

# Deleting a Key from Nested Dictionary
del Dict['A'][2]
print("\nDeleting a key from Nested Dictionary: ")
print(Dict)
```

```
Initial Dictionary:
{5: 'Welcome', 6: 'To', 7: 'CMR TC', 'A': {1: 'CMR TC', 2: 'For', 3: 'CMR
TC'}, 'B': {1: 'CMR TC', 2: 'Life'}}

Deleting a specific key:
{5: 'Welcome', 7: 'CMR TC', 'A': {1: 'CMR TC', 2: 'For', 3: 'CMR TC'},
'B': {1: 'CMR TC', 2: 'Life'}}

Deleting a key from Nested Dictionary:
{5: 'Welcome', 7: 'CMR TC', 'A': {1: 'CMR TC', 3: 'CMR TC'}, 'B': {1: 'CMR
TC', 2: 'Life'}}
```

Dictionary Methods

copy()

To copy one dictionary to another the copy method is used, So the key-value pairs of one dictionary will be copied to the other one. Using this process to a dictionary with existing contents makes all pairs of that active dictionary to be put back with the new pairs.

so this all the items will be copied and become a component of the newly declared dictionary item.

The copy() method doesn't take any parameters.

In [70]:

```
Depts = {'Dep#1':'CSE','Dep#2':'IT', 'Dep#3':'AI & ML' }
Vehicles = Depts.copy()
print("All Top Depts in CMR TC : ", Vehicles)
```

```
All Top Depts in CMR TC :  {'Dep#1': 'CSE', 'Dep#2': 'IT', 'Dep#3': 'AI &
ML'}
```

In [71]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.copy()
print(x)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

clear()

The clear() method removes all items from the dictionary. clear() method doesn't return any value.

In [72]:

```
text = {1: "CMR TC", 2: "CSE"}

text.clear()
print('text =', text)
```

```
text = {}
```

In [73]:

```
original = {1:'CMR TC', 2:'CSE'}

new = original.copy()

new.clear()

print('new: ', new)
print('original: ', original)
```

```
new:  {}
original:  {1: 'CMR TC', 2: 'CSE'}
```

In [74]:

```
#Unlike copy(), the assignment operator does deep copy.
original = {1:'CMR TC', 2:'CSE'}

new = original.copy()

new.clear()
print('new: ', new)
print('original: ', original)

original = {1:'one', 2:'two'}

new = original

new.clear()
print('new: ', new)
print('original: ', original)
```

```
new:  {}
original:  {1: 'CMR TC', 2: 'CSE'}
new:  {}
original:  {}
```

update()

The process of update defines two means, one is adding a new element to an existing dictionary or updating a key-value pair of an existing dictionary.

So when a new item is added it gets appended to the end of the dictionary. Similarly, when an existing dictionary component is updated the there will not be any positional change to a component only the update will be applied for the impacted item.

In [75]:

```
Bikes={'Bike#1':'Bajaj','Bike#2':'Hero Honda','Bike#3':'Yamaha' }
Bikes.update({'Bike#4' : 'Bullet'})
print("All Top Bikes in market List1 : ", Bikes)
print("!-----------------------------------------------!")
Bikes.update( {'Bike#3' : 'Hero-Honda'})
print("All Top Bikes in market List2 : ", Bikes)
```

```
All Top Bikes in market List1 :  {'Bike#1': 'Bajaj', 'Bike#2': 'Hero Hond
a', 'Bike#3': 'Yamaha', 'Bike#4': 'Bullet'}
!-----------------------------------------------!
All Top Bikes in market List2 :  {'Bike#1': 'Bajaj', 'Bike#2': 'Hero Hond
a', 'Bike#3': 'Hero-Honda', 'Bike#4': 'Bullet'}
```

In [76]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

car.update({"color": "White"})
print(car)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'White'}
```

In [77]:

```
#Update with another Dictionary.
Dictionary1 = { 'A': 'CMR TC', 'B': 'CSE', }
Dictionary2 = { 'B': 'CMR TC'}

print("Original Dictionary:")
print(Dictionary1)

Dictionary1.update(Dictionary2)
print("Dictionary after updation:")
print(Dictionary1)
```

```
Original Dictionary:
{'A': 'CMR TC', 'B': 'CSE'}
Dictionary after updation:
{'A': 'CMR TC', 'B': 'CMR TC'}
```

In [78]:

```
#Update with an iterable.
Dictionary1 = { 'A': 'CMR TC'}

print("Original Dictionary:")
print(Dictionary1)

Dictionary1.update(B = 'For', C = 'CMR TC')
print("Dictionary after updation:")
print(Dictionary1)
```

```
Original Dictionary:
{'A': 'CMR TC'}
Dictionary after updation:
{'A': 'CMR TC', 'B': 'For', 'C': 'CMR TC'}
```

In [79]:

```
#The update() method also allows iterable key/value pairs as parameter.
einventory = {'Fan': 200, 'Bulb':150, 'Led':1000}
print("Inventory:",einventory)
einventory.update(cooler=50,switches=1000)
print("Updated inventory:",einventory)
```

```
Inventory: {'Fan': 200, 'Bulb': 150, 'Led': 1000}
Updated inventory: {'Fan': 200, 'Bulb': 150, 'Led': 1000, 'cooler': 50, 's
witches': 1000}
```

keys()

The keys() method returns a view object. The view object contains the keys of the dictionary, as a list.

In [80]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.keys()
print(x)
```

```
dict_keys(['brand', 'model', 'year'])
```

In [81]:

```
product = {'name':'laptop','brand':'hp','price':80000}
p = product.keys()
print(p)
```

```
dict_keys(['name', 'brand', 'price'])
```

In [82]:

```
Dictionary1 = {'A': 'CMR TC', 'B': 'For', 'C': 'CSE'}
print(Dictionary1.keys())
empty_Dict1 = {}
print(empty_Dict1.keys())
```

```
dict_keys(['A', 'B', 'C'])
dict_keys([])
```

In [83]:

```
Dictionary1 = {'A': 'CMR TC', 'B': 'For', 'C': 'CSE'}
print("Keys before Dictionary Updation:")
keys = Dictionary1.keys()
print(keys)
Dictionary1.update({'C':'CMR TC'})
print('\nAfter dictionary is updated:')
print(keys)
```

```
Keys before Dictionary Updation:
dict_keys(['A', 'B', 'C'])

After dictionary is updated:
dict_keys(['A', 'B', 'C'])
```

In [84]:

```
#When an item is added in the dictionary, the view object also gets updated:
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.keys()
print(x)
car["color"] = "white"
print(car)
print(x)
```

```
dict_keys(['brand', 'model', 'year'])
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'white'}
dict_keys(['brand', 'model', 'year', 'color'])
```

In [85]:

```
product = {'name':'laptop','brand':'hp','price':80000}

for p in product.keys():
    if p == 'price' and product[p] > 50000:
        print("product price is too high",)
```

```
product price is too high
```

In [86]:

```
inventory = {'apples': 25, 'bananas': 220, 'oranges': 525, 'pears': 217}

for akey in inventory.keys():
    if akey == 'bananas' and inventory[akey] > 200:
        print("We have sufficient inventory for the ", akey)
```

```
We have sufficient inventory for the  bananas
```

In [87]:

```
# keys in Python3 does not support indexing.
test_dict = { "CMR TC" : 7, "for" : 1, "CMR TC" : 2 }
j = 0
for i in test_dict:
    if (j == 1):
        print ('2nd key using loop : ' + i)
        print ('2nd value using keys() : ' + test_dict[i])
    j = j + 1
```

```
2nd key using loop : for

---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[87], line 7
      5 if (j == 1):
      6     print ('2nd key using loop : ' + i)
----> 7     print ('2nd value using keys() : ' + test_dict[i])
      8 j = j + 1

TypeError: can only concatenate str (not "int") to str
```

items()

The items method is used for displaying all the elements (tuples) present in the python dictionary.

so when a dictionary item is applied to an items method all keys and values associated with that respective dictionary will be displayed.

In [88]:

```
student = {}
items = student.items()
print(items)
```

```
dict_items([])
```

In [89]:

```
Depts = {'Dep#1':'CSE','Dep#2':'IT', 'Dep#3':'AI & ML' }
print('All Top Depts in the CMR TC',Depts.items())
```

```
All Top Depts in the CMR TC dict_items([('Dep#1', 'CSE'), ('Dep#2', 'IT'),
('Dep#3', 'AI & ML')])
```

In [90]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.items()
print(x)
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
```

In [91]:

```
#To show working of items() after modification of Dictionary.
Dictionary1 = { 'A': 'CMR TC', 'B': 4, 'C': 'CMR TC' }

print("Original Dictionary items:")

items = Dictionary1.items()

print(items)

del[Dictionary1['C']]
print('Updated Dictionary:')
print(items)
```

```
Original Dictionary items:
dict_items([('A', 'CMR TC'), ('B', 4), ('C', 'CMR TC')])
Updated Dictionary:
dict_items([('A', 'CMR TC'), ('B', 4)])
```

In [92]:

```
#When an item in the dictionary changes value, the view object also gets updated:
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.items()
print(x)
car["year"] = 2018
print(x)
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2018)])
```

In [93]:

```
student = {'name':'rohan', 'course':'B.Tech', 'email':'rohan@abc.com'}
for st in student:
    print("(",st, ":", student[st], end="), ")

items = student.items()
print("\n", items)
```

```
( name : rohan), ( course : B.Tech), ( email : rohan@abc.com),
 dict_items([('name', 'rohan'), ('course', 'B.Tech'), ('email', 'rohan@ab
c.com')])
```

len()

The len() method is used to determine the count elements in a given dictionary component. so the overall count of the total number of key-value pairs in the corresponding dictionary will be displayed.

This acts moreover as a wrapper method so it means the dictionary item will be wrapped into the length method.

To determine how many items (key-value pairs) a dictionary has, use the len() function.

In [94]:

```
Bikes={'Bike#1':'Bajaj','Bike#2':'Hero Honda','Bike#3':'Yamaha' }
print('Total bikes in the market',len(Bikes))
```

```
Total bikes in the market 3
```

str()

The str() method is used for making a dictionary into a string format. this is more of a typecasting method. So typecasting means conversion of a component in one data type to a different data type value.

again this implies a wrapper process where the impacted dictionary component will be wrapped into the str() method.

In [95]:

```
Bikes = {'Bike#1' : 'Bajaj','Bike#2' : 'Hero Honda','Bike#3' : 'Yamaha' }
print(type(Bikes))
print(Bikes)
Bikes_str = str(Bikes)
print(type(Bikes_str))
print(Bikes_str)
```

```
<class 'dict'>
{'Bike#1': 'Bajaj', 'Bike#2': 'Hero Honda', 'Bike#3': 'Yamaha'}
<class 'str'>
{'Bike#1': 'Bajaj', 'Bike#2': 'Hero Honda', 'Bike#3': 'Yamaha'}
```

fromkeys()

The fromkeys() method returns a dictionary with the specified keys and the specified value.

In [96]:

```
x = ('key1', 'key2', 'key3')
y = 0
thisdict = dict.fromkeys(x, y)
print(thisdict)
```

```
{'key1': 0, 'key2': 0, 'key3': 0}
```

In [97]:

```
x = ('key1', 'key2', 'key3')
thisdict = dict.fromkeys(x)
print(thisdict)
```

```
{'key1': None, 'key2': None, 'key3': None}
```

In [98]:

```
seq = { 'a', 'b', 'c', 'd', 'e' }
res_dict = dict.fromkeys(seq)
print ("The newly created dict with None values : " + str(res_dict))

val=(1,'gh',3,4,4)
res_dict2 = dict.fromkeys(seq, val)
print ("The newly created dict with 1 as value : " + str(res_dict2))
```

```
The newly created dict with None values : {'a': None, 'd': None, 'e': Non
e, 'b': None, 'c': None}
The newly created dict with 1 as value : {'a': (1, 'gh', 3, 4, 4), 'd':
(1, 'gh', 3, 4, 4), 'e': (1, 'gh', 3, 4, 4), 'b': (1, 'gh', 3, 4, 4), 'c':
(1, 'gh', 3, 4, 4)}
```

get()

The get() method returns the value of the item with the specified key.

In [99]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.get("model")
print(x)
```

```
Mustang
```

In [100]:

```
planet_size = {"Earth": 40075, "Saturn": 378675, "Jupiter": 439264}
print(planet_size["Earth"])
print(planet_size.get("Saturn"))
```

```
40075
378675
```

In [101]:

```
#Try to return the value of an item that do not exist:
planet_size = {"Earth": 40075, "Saturn": 378675, "Jupiter": 439264}
print(planet_size.get("Arrakis"))
print(planet_size.get("Arrakis", "Huh?"))
```

```
None
Huh?
```

In [102]:

```
dic = {"A":1, "B":2}
print(dic.get("A"))
print(dic.get("C"))
print(dic.get("C","Not Found ! "))
```

```
1
None
Not Found !
```

In [103]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.get("price", 15000)
print(x)
print(car)
```

```
15000
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

pop()

The pop() method removes the specified item from the dictionary.

In [104]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
car.pop("model")
print(car)
```

```
{'brand': 'Ford', 'year': 1964}
```

In [105]:

```
# The value of the removed item is the return value of the pop() method:
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.pop("model")
print(x)
```

```
Mustang
```

In [106]:

```
# The pop() method accepts the key as an argument and remove the
#associated value.
Dict = {1: 'CMR TC', 2: 'CSE', 3: 'Department'}

pop_ele = Dict.pop(3)
print(Dict)
```

```
{1: 'CMR TC', 2: 'CSE'}
```

In [107]:

```
inventory = {'shirts': 25, 'paints': 220, 'shock': 525, 'tshirts': 217}
element = inventory.pop('shoes',100)
print(element)
```

```
100
```

In [108]:

```
inventory = {'shirts': 25, 'paints': 220, 'shocks': 525, 'tshirts': 217}
print(inventory)
p = inventory.pop('shirts')
print("Removed",p,"shirts")
print(inventory)
```

```
{'shirts': 25, 'paints': 220, 'shocks': 525, 'tshirts': 217}
Removed 25 shirts
{'paints': 220, 'shocks': 525, 'tshirts': 217}
```

In [109]:

```
test_dict = { "Nikhil" : 7, "Akshat" : 1, "Akash" : 2 }
print ("The dictionary before deletion : " + str(test_dict))
pop_ele = test_dict.pop('Akash')
print ("Value associated to poppped key is : " + str(pop_ele))
print ("Dictionary after deletion is : " + str(test_dict))
```

```
The dictionary before deletion : {'Nikhil': 7, 'Akshat': 1, 'Akash': 2}
Value associated to poppped key is : 2
Dictionary after deletion is : {'Nikhil': 7, 'Akshat': 1}
```

popitem()

The popitem() method removes the item that was last inserted into the dictionary. The removed item is the return value of the popitem() method, as a tuple.

In [110]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
car.popitem()
print(car)
```

```
{'brand': 'Ford', 'model': 'Mustang'}
```

In [111]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.popitem()
print(x)
```

```
('year', 1964)
```

In [112]:

```
Dict = {1: 'CMR TC', 'name': 'For', 3: 'CSE'}

pop_ele = Dict.popitem()
print("\nDictionary after deletion: " + str(Dict))
print("The arbitrary pair returned is: " + str(pop_ele))
```

```
Dictionary after deletion: {1: 'CMR TC', 'name': 'For'}
The arbitrary pair returned is: (3, 'CSE')
```

In [113]:

```
#If the dictionary is empty, it returns an error KeyError.
inventory = {}
print(inventory)
p = inventory.popitem()
print("Removed",p)
print(inventory)
```

```
{}

---------------------------------------------------------------------------
-
KeyError                                  Traceback (most recent call las
t)
Cell In[113], line 4
      2 inventory = {}
      3 print(inventory)
----> 4 p = inventory.popitem()
      5 print("Removed",p)
      6 print(inventory)

KeyError: 'popitem(): dictionary is empty'
```

In [114]:

```
inventory = {'shirts': 25, 'paints': 220, 'shocks': 525, 'tshirts': 217}

print(inventory)
p = inventory.popitem()
print("Removed",p)
print(inventory)
inventory.update({'pajama':117})
print(inventory)
```

```
{'shirts': 25, 'paints': 220, 'shocks': 525, 'tshirts': 217}
Removed ('tshirts', 217)
{'shirts': 25, 'paints': 220, 'shocks': 525}
{'shirts': 25, 'paints': 220, 'shocks': 525, 'pajama': 117}
```

In [115]:

```
test_dict = { "Nikhil" : 7, "Akshat" : 1, "Akash" : 2 }

print ("The dictionary before deletion : " + str(test_dict))
n = len(test_dict)

for i in range(0, n) :
    print ("Rank " + str(i + 1) + " " + str(test_dict.popitem()))
print ("The dictionary after deletion : " + str(test_dict))
```

```
The dictionary before deletion : {'Nikhil': 7, 'Akshat': 1, 'Akash': 2}
Rank 1 ('Akash', 2)
Rank 2 ('Akshat', 1)
Rank 3 ('Nikhil', 7)
The dictionary after deletion : {}
```

setdefault()

The setdefault() method returns the value of the item with the specified key.If the key does not exist, insert the key, with the specified value

In [116]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.setdefault("model", "Bronco")
#x=car["model"]
print(x)
car
```

```
Mustang
```

Out[116]:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

In [117]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.setdefault("color", "white")
print(x)
car
```

```
white
```

Out[117]:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'white'}
```

In [118]:

```
coursefee = {'B,Tech': 400000, 'BA':2500, 'B.COM':50000}
p = coursefee.setdefault('BA')
print("default",p)
print(coursefee)
```

```
default 2500
{'B,Tech': 400000, 'BA': 2500, 'B.COM': 50000}
```

In [119]:

```
coursefee = {'B,Tech': 400000, 'BA':2500, 'B.COM':50000}
p = coursefee.setdefault('BCA')
print("default",p)
print(coursefee)
```

```
default None
{'B,Tech': 400000, 'BA': 2500, 'B.COM': 50000, 'BCA': None}
```

In [120]:

```
coursefee = {'B,Tech': 400000, 'BA':2500, 'B.COM':50000}
p = coursefee.setdefault('BCA',100000)
print("default",p)
print(coursefee)
```

```
default 100000
{'B,Tech': 400000, 'BA': 2500, 'B.COM': 50000, 'BCA': 100000}
```

In [121]:

```
Dictionary1 = { 'A': 'CMR TC', 'B': 'For', 'C': 'CSE'}
Third_value = Dictionary1.setdefault('C')
print("Dictionary:", Dictionary1)
print("Third_value:", Third_value)
```

```
Dictionary: {'A': 'CMR TC', 'B': 'For', 'C': 'CSE'}
Third_value: CSE
```

values()

The values() method returns a view object. The view object contains the values of the dictionary, as a list.

The view object will reflect any changes done to the dictionary.

In [122]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.values()
print(x)
```

```
dict_values(['Ford', 'Mustang', 1964])
```

In [123]:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.values()
car["year"] = 2018
print(x)
```

```
dict_values(['Ford', 'Mustang', 2018])
```

In [124]:

```
dictionary = {"raj": 2, "striver": 3, "vikram": 4}
print(dictionary.values())

dictionary = {"CMR TC": "5", "for": "3", "CSE": "5"}
print(dictionary.values())
```

```
dict_values([2, 3, 4])
dict_values(['5', '3', '5'])
```

In [125]:

```
einventory = {'Fan': 200, 'Bulb':150, 'Led':1000}
length = len(einventory)
print("Total number of values:",length)
keys = einventory.keys()
print("All the Keys:",keys)
item = einventory.items()
print("Items:",einventory)
p = einventory.popitem()
print("Deleted items:",p)
stock = einventory.values()
print("Stock available",einventory)
```

```
Total number of values: 3
All the Keys: dict_keys(['Fan', 'Bulb', 'Led'])
Items: {'Fan': 200, 'Bulb': 150, 'Led': 1000}
Deleted items: ('Led', 1000)
Stock available {'Fan': 200, 'Bulb': 150}
```

In [126]:

```
salary = {"raj" : 50000, "striver" : 60000, "vikram" : 5000}
list1 = salary.values()
print(list1)
print(sum(list1))
```

```
dict_values([50000, 60000, 5000])
115000
```

Loop Through a Dictionary

You can loop through a dictionary by using a for loop. When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

In [127]:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(x)
```

```
brand
model
year
```

In [128]:

```
for x in thisdict:
  print(thisdict[x])
```

```
Ford
Mustang
1964
```

In [129]:

```
#You can also use the values() method to return values of a dictionary:
for x in thisdict.values():
  print(x)
```

```
Ford
Mustang
1964
```

In [130]:

```
#Loop through both keys and values, by using the items() method:
for x, y in thisdict.items():
  print(x, y)
```

```
brand Ford
model Mustang
year 1964
```

Nested Dictionary

In [131]:

```
Dict = {1: 'CMR TC', 2: 'For',
        3:{'A' : 'Welcome', 'B' : 'To', 'C' : 'CSE'}}
print(Dict)
```

```
{1: 'CMR TC', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'CSE'}}
```

In [132]:

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}
print(people)
```

```
{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie', 'ag
e': '22', 'sex': 'Female'}}
```

In [133]:

```
#To access element of a nested dictionary, we use indexing [] syntax in Python.
print(people[1]['name'])
print(people[1]['age'])
print(people[1]['sex'])
```

```
John
27
Male
```

In [134]:

```
#change or add elements in a nested dictionary
people[3] = {}
people[3]['name'] = 'Luna'
people[3]['age'] = '24'
people[3]['sex'] = 'Female'
people[3]['married'] = 'No'
print(people[3])
print(people)
```

```
{'name': 'Luna', 'age': '24', 'sex': 'Female', 'married': 'No'}
{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie', 'ag
e': '22', 'sex': 'Female'}, 3: {'name': 'Luna', 'age': '24', 'sex': 'Femal
e', 'married': 'No'}}
```

In [135]:

```
#delete elements from a nested dictionary
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},
          3: {'name': 'Luna', 'age': '24', 'sex': 'Female', 'married': 'No'},
          4: {'name': 'Peter', 'age': '29', 'sex': 'Male', 'married': 'Yes'}}

del people[3]['married']
del people[4]['married']

print(people[3])
print(people[4])
print(people)
```

```
{'name': 'Luna', 'age': '24', 'sex': 'Female'}
{'name': 'Peter', 'age': '29', 'sex': 'Male'}
{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie', 'ag
e': '22', 'sex': 'Female'}, 3: {'name': 'Luna', 'age': '24', 'sex': 'Femal
e'}, 4: {'name': 'Peter', 'age': '29', 'sex': 'Male'}}
```

In [136]:

```
#delete dictionary from a nested dictionary
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},
          3: {'name': 'Luna', 'age': '24', 'sex': 'Female'},
          4: {'name': 'Peter', 'age': '29', 'sex': 'Male'}}

del people[3], people[4]
print(people)
```

```
{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie', 'ag
e': '22', 'sex': 'Female'}}
```

In [137]:

```
#iterate through a Nested dictionary
people = {1: {'Name': 'John', 'Age': '27', 'Sex': 'Male'},
          2: {'Name': 'Marie', 'Age': '22', 'Sex': 'Female'}}

for p_id, p_info in people.items():
    print("\nPerson ID:", p_id)

    for key in p_info:
        print(key + ':', p_info[key])
```

```
Person ID: 1
Name: John
Age: 27
Sex: Male

Person ID: 2
Name: Marie
Age: 22
Sex: Female
```

Example Programs

In [138]:

```
#What is printed by the following statements?
mydict = {"cat":12, "dog":6, "elephant":23}
mydict["mouse"] = mydict["cat"] + mydict["dog"]
print(mydict["mouse"])
```

```
18
```

In [139]:

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Enter the details of the new employee....");
Employee["Name"] = input("Name: ");
Employee["Age"] = int(input("Age: "));
Employee["salary"] = int(input("Salary: "));
Employee["Company"] = input("Company:");
print("printing the new data");
print(Employee)
```

```
<class 'dict'>
printing Employee data ....
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
Enter the details of the new employee....
Name: john
Age: 23
Salary: 12345
Company:cmr
printing the new data
{'Name': 'john', 'Age': 23, 'salary': 12345, 'Company': 'cmr'}
```

In [140]:

```
#You can access a list item within a dictionary by referring to
#the list's key within the dictionary
food = {"Fruit": ["Apple", "Orange", "Banana"], "Vegetables": ("Eat", "Your", "Greens")}
print(food["Fruit"][1])
```

```
Orange
```

In [141]:

```
#If your dictionary contains lists, you can update list items within
#those lists by referring to the list's key within the dictionary
food = {"Fruit": ["Apple", "Orange", "Banana"]}
print(food)

food["Fruit"][2] = "Watermelon"
print(food)
```

```
{'Fruit': ['Apple', 'Orange', 'Banana']}
{'Fruit': ['Apple', 'Orange', 'Watermelon']}
```

In [142]:

```
#Extract the value associated with the key color and assign it to the variable color.
info = {'personal_data':
         {'name': 'Lauren',
          'age': 20,
          'major': 'Information Science',
          'physical_features':
             {'color': {'eye': 'blue',
                        'hair': 'brown'},
              'height': "5'8"}
         },
        'other':
         {'favorite_colors': ['purple', 'green', 'blue'],
          'interested_in': ['social media', 'intellectual property', 'copyright', 'music
         }
       }
color1=info['personal_data']['physical_features']['color']
print(color1)
```

```
{'eye': 'blue', 'hair': 'brown'}
```

In [143]:

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

for k in inventory:
    print("Got key", k)
```

```
Got key apples
Got key bananas
Got key oranges
Got key pears
```

In [144]:

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
print('apples' in inventory)
print('cherries' in inventory)

if 'bananas' in inventory:
    print(inventory['bananas'])
else:
    print("We have no bananas")
```

```
True
False
312
```

In [145]:

```
#What is printed by the following statements
mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
answer = mydict.get("cat")//mydict.get("dog")
print(answer)
```

```
2
```

In [146]:

```
#What is printed by the following statements
total = 0
mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
for akey in mydict:
    if len(akey) > 3:
        total = total + mydict[akey]
print(total)
```

```
43
```

In [147]:

```
def myFunc(e):
  return e['year']

cars = [
  {'car': 'Ford', 'year': 2005},
  {'car': 'Mitsubishi', 'year': 2000},
  {'car': 'BMW', 'year': 2019},
  {'car': 'VW', 'year': 2011}
]

cars.sort(key=myFunc)
print(cars)
```

```
[{'car': 'Mitsubishi', 'year': 2000}, {'car': 'Ford', 'year': 2005}, {'ca
r': 'VW', 'year': 2011}, {'car': 'BMW', 'year': 2019}]
```

In [148]:

```
#Write a program that finds the key in a dictionary that has the maximum value.
#If two keys have the same maximum value, it's OK to print out either one.
d = {'a': 194, 'b': 54, 'c':34, 'd': 44, 'e': 312, 'full':31}
ks = d.keys()
best_key_so_far = list(ks)[0]
for k in ks:
    if d[k] > d[best_key_so_far]:
        best_key_so_far = k

print("key " + best_key_so_far + " has the highest value, " + str(d[best_key_so_far]))
```

```
key e has the highest value, 312
```

In [151]:

```
#sentence translation
pirate = {}
pirate['sir'] = 'matey'
pirate['hotel'] = 'fleabag inn'
pirate['student'] = 'swabbie'
pirate['boy'] = 'matey'
pirate['restaurant'] = 'galley'
#and so on

sentence = input("Please enter a sentence in English: ")

psentence = []
words = sentence.split()
for aword in words:
    if aword in pirate:
        psentence.append(pirate[aword])
    else:
        psentence.append(aword)

print(" ".join(psentence))
```

```
Please enter a sentence in English: cmrtc
cmrtc
```

In [152]:

```
#code counts the frequencies of different numbers in the list.
L = ['E', 'F', 'B', 'A', 'D', 'I', 'I', 'C', 'B', 'A', 'D', 'D', 'E', 'D']

d = {}
for x in L:
    if x in d:
        d[x] = d[x] + 1
    else:
        d[x] = 1

for x in d.keys():
    print("{} appears {} times".format(x, d[x]))
```

```
E appears 2 times
F appears 1 times
B appears 2 times
A appears 2 times
D appears 4 times
I appears 2 times
C appears 1 times
```

In [153]:

```
#We can force the results to be displayed in some fixed ordering, by sorting the keys.
L = ['E', 'F', 'B', 'A', 'D', 'I', 'I', 'C', 'B', 'A', 'D', 'D', 'E', 'D']
d = {}
for x in L:
    if x in d:
        d[x] = d[x] + 1
    else:
        d[x] = 1
y = sorted(d.keys(), reverse=True)
for k in y:
    print("{} appears {} times".format(k, d[k]))
```

```
I appears 2 times
F appears 1 times
E appears 2 times
D appears 4 times
C appears 1 times
B appears 2 times
A appears 2 times
```

In [154]:

```
#Write a program that allows the user to enter a string.
#It then prints a table of the letters of the alphabet in alphabetical
#order which occur in the string together with the number of times each letter occurs.
#Case should be ignored.
x = input("Enter a sentence: ")
x = x.lower()

alphabet = 'abcdefghijklmnopqrstuvwxyz'
letter_count = {}

for char in x:
    if char in alphabet:
        if char in letter_count:
            letter_count[char] = letter_count[char] + 1
        else:
            letter_count[char] = 1

keys = letter_count.keys()
for char in sorted(keys):
    print(char, letter_count[char])
```

```
Enter a sentence: cmrtc
c 2
m 1
r 1
t 1
```

In [ ]: