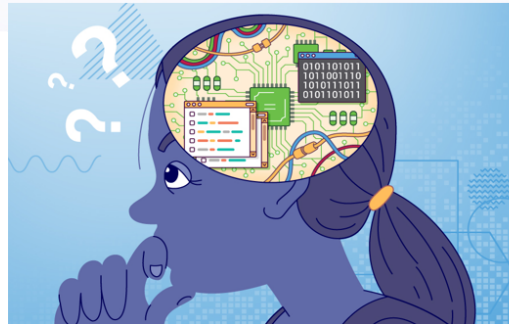


# WELCOME TO

## Day-2

# THINKING PROGRAMMING

*First solve, then code !*



# ABOUT ME

- I'm Asha Rani K P, working as an Assistant Professor in the Department of Computer Science & Engineering at Dr. Ambedkar Institute of Technology, Bengaluru-560056.



- 13+ years of Teaching Experience.
- Research interests include Data Science with emphasis on Machine Learning and Deep Learning.

# *AGENDA FOR DAY-2*

- Functions
- Strings
- Lists
- Modules
- Arrays
- Hands-On Session

**FUNCTIONS IN PYTHON**

- A function is a block of code which only runs when it is called.
- A function is Reusable piece of code.
- Functions are created for solving a specific problem.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

Built – in functions

`print(), range(), min(), max() etc.`

User defined functions

`Area_of_circle()`

# BUILT-IN FUNCTIONS IN PYTHON

1.	abs()	<i>abs(x)</i> where x is an integer or floating-point number.	The abs() function returns the absolute value of a number.
2.	min()	min(arg_1, arg_2, arg_3,...,arg_n) where arg_1, arg_2, arg_3 are the arguments.	The min() function returns the smallest of two or more arguments
3.	max()	max(arg_1, arg_2, arg_3,...,arg_n) where arg_1, arg_2, arg_3 are the arguments.	The max() function returns the largest of two or more
4.	divmod()	<u>divmod(a, b)</u> where a and b are numbers representing numerator and denominator.	The divmod() function takes two numbers as arguments and return a pair of numbers consisting of their quotient and remainder. For example, if a and b are integer values, then the result is the same as (a // b, a % b). If either a or b is a floating-point number, then the result is (q, a% b), where q is the whole number of the quotient
5.	pow()	pow(x, y)  where x and y are numbers	The pow(x, y) function returns x to the power y which is equivalent to using the power operator: x**y
6.	<u>len()</u>	len(s) where s may be a string, byte, list, tuple, range, dictionary or a set.	The <u>len()</u> function returns the length or the number of items in an object.

```
In [1]: 1 # Built-in Functions Examples
        2 abs(-3)
```

```
Out[1]: 3
```

```
In [2]: 1 # Built-in Functions Examples
        2 min(1, 2, 3, 4, 5)
```

Out[2]: 1

```
In [3]: 1 # Built-in Functions Examples
        2 max(4, 5, 6, 7, 8)
```

Out[3]: 8

```
In [4]: 1 # Built-in Functions Examples
        2 divmod(5, 2)
```

Out[4]: (2, 1)

```
In [5]: 1 # Built-in Functions Examples
        2 divmod(8.5, 3)
```

Out[5]: (2.0, 2.5)

```
In [1]: 1 # Built-in Functions Examples
        2 pow(3, 2)
```

Out[1]: 9

```
In [2]: 1 # Built-in Functions Examples
        2 len("Japan")
```

Out[2]: 5

# USERDEFINED FUNCTIONS

- Functions that we define ourselves to do certain specific task are referred as user-defined functions.

## CREATING A FUNCTION

## Syntax:

```
def function_name(argument1,argument2,...):  
    Statement 1  
    Statement 2  
    .....  
    .....  
    Statement n
```

### Example

```
#FUNCTION DEFINITION  
def my_function():  
    print("WELCOME TO FUNCTIONS IN PYTHON")
```

## Syntax for defining a function

```
def function_name(arguments):  
    statement 1  
    statement 2  
    .....  
    .....  
    .....  
    return statement
```

## Example

```
def area_circle(radius):  
    area = 3.14 * r * r  
    return area
```



Radius = 1 cm

area\_circle(1)



Radius = 3 cm

area\_circle(3)

# CALLING A FUNCTION



## Example

#FUNCTION DEFINITION

```
def my_function():  
    print("WELCOME TO FUNCTIONS IN PYTHON ")
```

#CALLING FUNCTION

```
my_function()
```

#OUTPUT

```
WELCOME TO FUNCTIONS IN PYTHON
```

```
In [8]: 1 #Example  
2 #CREATING A FUNCTION----FUNCTION DEFINITION--- WILL NOT BE EXECUTED UNTIL ITS CALLED  
3  
4 def my_function():  
5     print("WELCOME TO FUNCTIONS IN PYTHON")  
6     print("Inside Function Definition")
```

```
In [9]: 1 #Example  
2 #FUNCTION DEFINITION----- FUNCTION CALL----- To call a function use the function name.  
3  
4 #FUNCTION DEFINITION  
5 def my_function():  
6     print("WELCOME TO FUNCTIONS IN PYTHON ")  
7     print("Inside Function Definition")  
8  
9 #CALLING FUNCTION  
10 my_function()
```

```
WELCOME TO FUNCTIONS IN PYTHON  
Inside Function Definition
```

# main() FUNCTION IN PYTHON

## Python Main Function

- Main function is like the entry point of a program.
- However, Python interpreter runs the code right from the first line.
- The execution of the code starts from the starting line and goes line by line.
- It does not matter where the main function is present or it is present or not.
- Since there is no `main()` function in Python, when the command to run a Python program is given to the interpreter, the code that is at level 0 indentation is to be executed.
- However, before doing that, **it will define a few special variables.**  
`__name__` **is one such special variable.**
- If the source file is executed as the main program, the interpreter sets the `__name__` **variable to have a value** `__main__`.
- If this file is being imported from another module, `__name__` will be set to the module's name.
- `__name__` is a built-in variable which evaluates to the **name of the current module.**

In [10]:

```
1 #Example
2 print("Outside main() Function")
3 #FUNCTION DEFINITION
4 def main():
5     print("Inside main() Function")
6
7 if __name__ == "__main__":
8     main()
9
```

Outside main() Function

Inside main() Function

## A Basic Python main()

```
def main():  
    print("Hello World!")  
  
if __name__ == "__main__":  
    main()
```

- In this code, there is a function called `main()` that prints the phrase `Hello World!` when the Python interpreter executes it.
- There is also a conditional (or `if`) statement that checks the value of `__name__` and compares it to the string `"__main__"`.
- When the `if` statement evaluates to `True`, the Python interpreter executes `main()`.
- When Python interpreter reads the `if` statement and sees that `__name__` does equal to `"__main__"`, it will execute the block of statements present there.

In [11]:

```
1  #Let's see an example without the main function.
2
3  #Example:
4
5  print("Good Morning")
6
7  def main():
8      print("Hello Python")
9
10 print("Good Evening")
```

Good Morning

Good Evening

In [12]:

```
1  #Now let's see the program with function call if __name__ == "__main__".
2
3  #Example 2:
4
5  print("Good Morning")
6
7  def main():
8      print("Hello Python")
9
10 print("Good Evening")
11
12 if __name__ == "__main__":
13     main()
```

Good Morning

Good Evening

Hello Python

```
In [13]: 1 #Program to Demonstrate a Function with and without Arguments
2 def function_definition_with_no_argument():
3     print("This is a function definition with NO Argument")
4 def function_definition_with_one_argument(message):
5     print(f"This is a function definition with {message}")
6 def main():
7     function_definition_with_no_argument()
8     function_definition_with_one_argument("One Argument")
9 if __name__ == "__main__":
10     main()
```

This is a function definition with NO Argument  
This is a function definition with One Argument

## return STATEMENT IN PYTHON

- The python return statement is used in a function to return something to the caller program.
- We can use the return statement inside a function only.
- In Python, every function returns something. If there are no return statements, then it returns None.
- If the return statement contains an expression, it's evaluated first and then the value is returned.
- The return statement terminates the function execution.
- A function can have multiple return statements. When any of them is executed, the function terminates.
- A function can return multiple types of values.

- Python function can return multiple values in a single return statement.

## return STATEMENT EXAMPLE

- Let's look at a simple example to add two numbers and return the total to the caller.

```
def add(x, y):  
    total = x + y  
    return total
```

## return STATEMENT EXAMPLE

- We can optimize the function by having the expression in the return statement.

```
def add(x, y):  
    return x + y
```

<br>

# Python Functions can have multiple return statements

- Single function can have multiple return statements. Execution of the function ends when one of these return statements is reached:

```
def type_of_int(i):  
    if i % 2 == 0:  
        return 'even'  
    else:  
        return 'odd'
```

In [14]:

```
1 def add(x, y):  
2     total = x + y  
3     return total  
4  
5 ans=add(2,5)  
6 print("SUM OF 2 NUMBERS =",ans)
```

SUM OF 2 NUMBERS = 7

In [15]:

```
1 def add(x, y):  
2     return x+y  
3  
4 ans=add(2,5)  
5 print("SUM OF 2 NUMBERS =",ans)
```

SUM OF 2 NUMBERS = 7



```
In [16]: 1 i=15
          2 def type_of_int(i):
          3     if i % 2 == 0:
          4         return 'even'
          5     else:
          6         return 'odd'
          7 type_of_int(i)
          8
```

```
Out[16]: 'odd'
```

## Python Function can return multiple types of values

- Unlike other programming languages, python functions are not restricted to return a single type of value.
- If you look at the function definition, it doesn't have any information about what it can return.

```
In [17]: 1 def various_return_types(n):
2         if(n==1):
3             return "Hello World."    # Return a string
4         elif(n==2):
5             return 42                 # Return a value
6         else:
7             return True               # Return a boolean
8     print(various_return_types(1))
9                                     #OUTPUT-Hello World.
10    print(various_return_types(2))
11                                     #OUTPUT-42
12    print(various_return_types(3))
13                                     #OUTPUT-True
14
```

```
Hello World.
42
True
```

## Returning Multiple Values in a single return Statement

- We can return multiple values from a single return statement. These values are separated by a comma and returned as a tuple to the caller program.

```
def return_multiple_values():  
    return 1, 2, 3  
  
print(return_multiple_values())  
print(type(return_multiple_values()))
```

Output:

```
(1, 2, 3)  
<class 'tuple'>
```

```
In [18]: 1 def return_multiple_values():  
2         return 1, 2, 3  
3  
4 print(return_multiple_values())  
5 print(type(return_multiple_values()))
```

```
(1, 2, 3)  
<class 'tuple'>
```

```
In [19]: 1 #Program to Demonstrate the Return of Multiple Values from a Function Definition  
2 def world_war():  
3     alliance_world_war = input("Which alliance won World War 2?")  
4     world_war_end_year = input("When did World War 2 end?")  
5     return alliance_world_war, world_war_end_year  
6 def main():  
7     alliance, war_end_year = world_war()  
8     print(f"The war was won by {alliance} and the war ended in {war_end_year}")  
9     if __name__ == "__main__":  
10        main()
```

The war was won by    and the war ended in

```
In [12]: 1 #Program to Check If a 3 Digit Number Is Armstrong Number or Not
2 user_number = int(input("Enter a 3 digit positive number to check for Armstrong number"))
3 def check_armstrong_number(number):
4     result = 0
5     temp = number
6     while temp != 0:
7         last_digit = temp % 10
8         result += pow(last_digit, 3)
9         temp = int(temp / 10)
10    if number == result:
11        print(f"Entered number {number} is a Armstrong number")
12    else:
13        print(f"Entered number {number} is not a Armstrong number")
14
15 def main():
16     check_armstrong_number(user_number)
17
18 if __name__ == "__main__":
19     main()
```

Entered number 123 is not a Armstrong number

## Void Functions

- In Python, it is possible to define functions without a return statement.
- Functions like this are called **void** functions, and they return None.

## SCOPE & LIFETIME OF VARIABLES IN PYTHON

- Python programs have two scopes: **global and local**.
- **A variable is a global variable if its value is accessible and modifiable throughout your program. Global variables have a global scope.**
- **A variable that is defined inside a function definition is a local variable.**
- **The lifetime of a variable refers to the duration of its existence.**
- **The local variable is created and destroyed every time the function is executed, and it cannot be accessed by any code outside the function definition.**

```
In [ ]: 1 def greet(name, msg="Good Morning!"):
        2     """
        3     This function greets to
        4     the person with the
        5     provided message.
        6
        7     If the message is not provided,
        8     it defaults to "Good
        9     morning!"
        10    """
        11
        12    print("Hello", name + ', ' + msg)
        13
        14
        15    greet("KARTHIK")
        16    greet("MANJUNATH", "How do you do?")
```

- In this function, the parameter name does not have a default value and is required (mandatory) during a call.

- On the other hand, the parameter msg has a default value of "Good morning!". So, it is optional during a call.
- If a value is provided, it will overwrite the default value.
- Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

```
In [ ]: 1 #Program to Demonstrate the Use of Default Parameters
2
3 def work_area(prompt, domain="Data Analytics"):
4     print(f"{prompt} {domain}")
5 def main():
6     work_area("Sam works in")
7     work_area("Alice has interest in", "Internet of Things")
8 if __name__ == "__main__":
9     main()
```

```
In [ ]: 1 #Program to Demonstrate the Use of Keyword Arguments
2 def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
3     print(f"This parrot wouldn't {action}, if you put {voltage}, volts through it.")
4     print(f"Lovely plumage, the {type}")
5     print(f"It's {state} !!!")
6
7 parrot(1000)
8 parrot(voltage=1000)
9 parrot(voltage=1000000, action='VOOOOOM')
10 parrot('a thousand', state='pushing up the daisies')
```

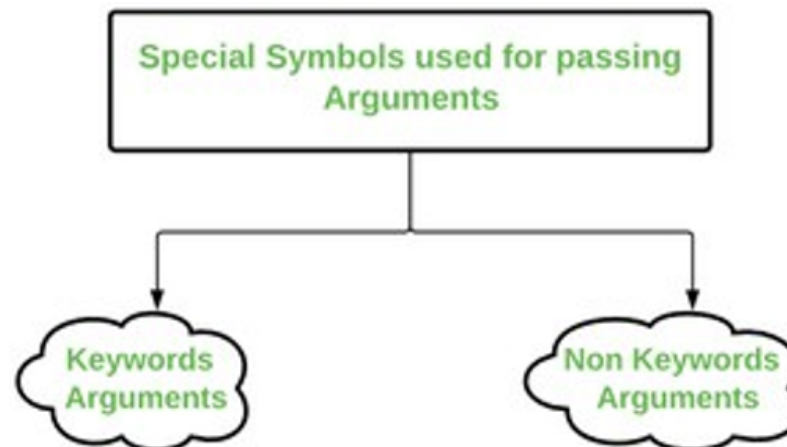
**Also, the following functional calls are invalid.**

- `parrot()` # required argument missing
- `parrot(voltage=5.0, 'dead')` # non-keyword argument after a keyword argument
- `parrot(110, voltage=220)` # duplicate value for the same argument
- `parrot(actor='John Cleese')` # unknown keyword argument

```
In [ ]: 1 #NUMBER OF ARGUMENTS IN FUNCTIONS---By default, a function must be called with the correct number of arguments, not
        2
        3 def my_function(fname, lname):
        4     print(fname + " " + lname)
        5
        6 my_function("PYTHON", "PROGRAMMING")
```

## **\*args and \*\*kwargs IN PYTHON**

- In Python, we can pass a variable number of arguments to a function using special symbols. There are two special symbols:



# 1.) \*args (Non-Keyword Arguments)

# 2.) \*\*kwargs (Keyword Arguments)

## 1.) \*args (Non-Keyword Arguments)

- **\*args allows you to take in more arguments than the number of formal arguments that you previously defined.**

```
In [ ]: 1 # Python program to illustrate *args for variable number of arguments
        2 def myFun(*argv):
        3     for arg in argv: print (arg)
        4
        5 myFun('Hello', 'Welcome', 'To', 'THINKING PROGRAMMING')
```

```
In [ ]: 1 # Python program to illustrate *args with first extra argument
        2 def myFun(arg1, *argv):
        3     print ("First argument :", arg1)
        4     for arg in argv:
        5         print("Next argument through *argv :", arg)
        6
        7
        8 myFun('Hello', 'Welcome', 'To', 'THINKING PROGRAMMING')
```

## 2.) \*\*kwargs (Keyword Arguments)



- Double star allows us to pass through keyword arguments (and any number of them).
- One can think of the kwargs as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the kwargs there doesn't seem to be any order in which they were printed out.

In [1]:

```
1 # Python program to illustrate *kwargs for variable number of keyword arguments
2 # items() method is used to return the list with all dictionary keys with values.
3
4 def myFun(**kwargs):
5     for key, value in kwargs.items():
6         print ("%s == %s" %(key, value))
7
8 myFun(first ='INTRODUCTION', mid ='TO', last='PYTHON PROGRAMMING')
```

```
first == INTRODUCTION
mid == TO
last == PYTHON PROGRAMMING
```

In [3]:

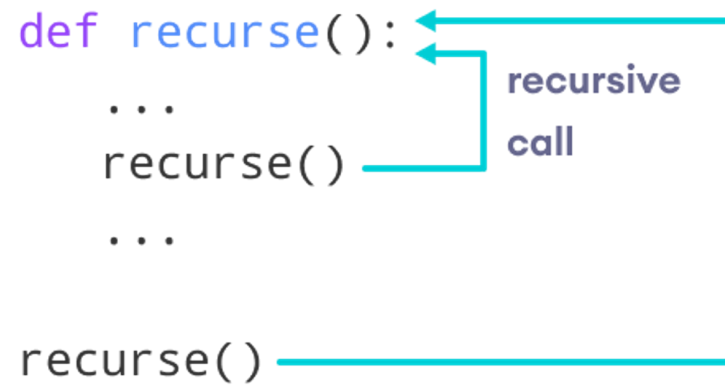
```
1 # Python program to illustrate *kwargs for # variable number of keyword arguments with # one extra argument.
2
3 def myFun(arg1, **kwargs):
4     for key, value in kwargs.items():
5         print ("%s == %s" %(key, value))
6
7 myFun("Hi", first ='INTRODUCTION', mid ='TO', last='PYTHON PROGRAMMING')
```

```
first == INTRODUCTION
mid == TO
last == PYTHON PROGRAMMING
```

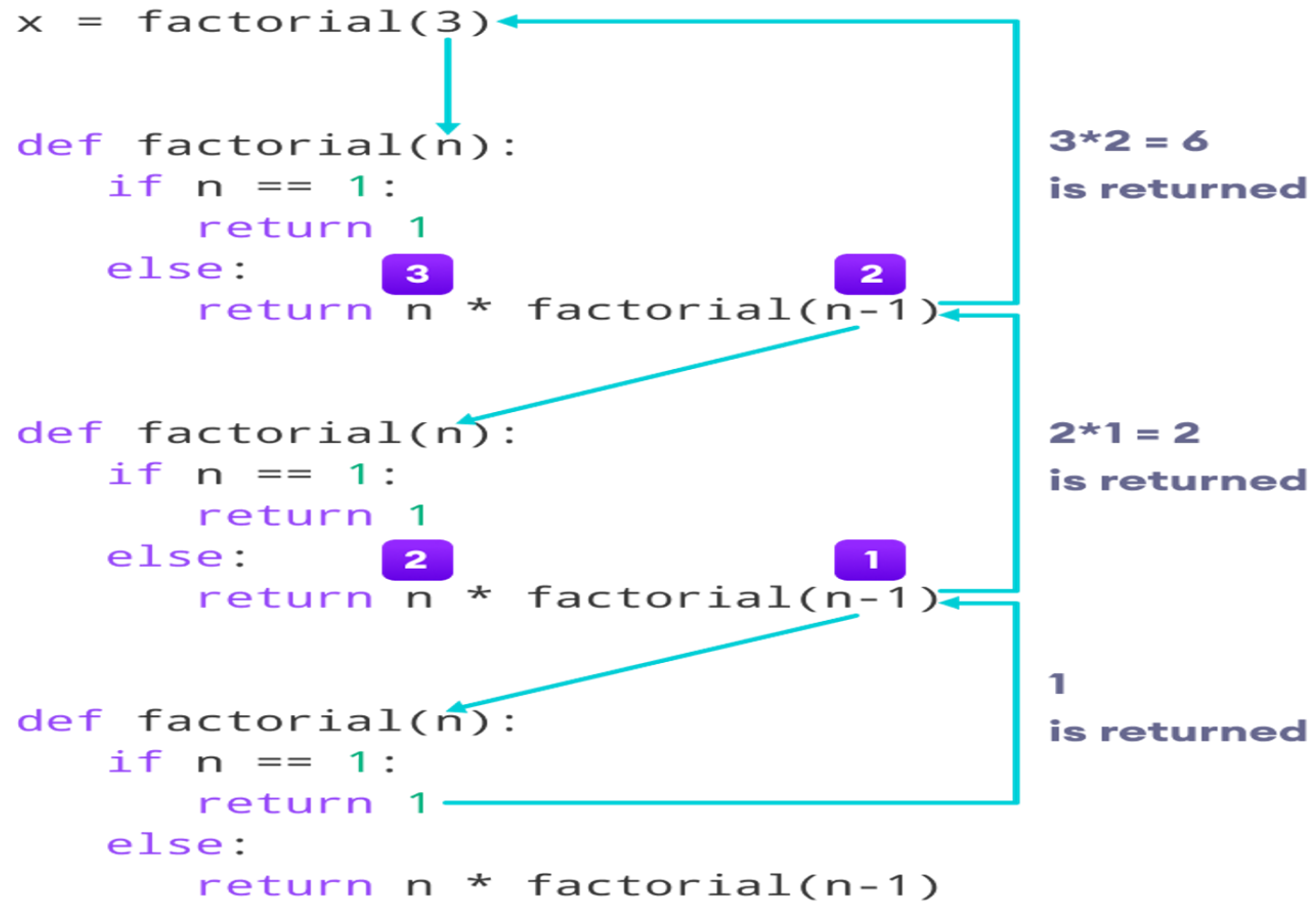
# RECURSIVE FUNCTION IN PYTHON

- In Python, we know that 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.

```
def recurse():  
    ...  
    recurse()  
    ...  
  
recurse()
```



recursive call



```
In [11]: 1 def factorial(x):
          2     """This is a recursive function
          3     to find the factorial of an integer"""
          4
          5     if x == 1:
          6         return 1
          7     else:
          8         return (x * factorial(x-1))
          9
         10
         11 num = 3
         12 print("The factorial of", num, "is", factorial(num))
```

The factorial of 3 is 6

# STRINGS IN PYTHON

- In Python, Strings are arrays of bytes representing Unicode characters.
- However, Python does not have a character data type, a single character is simply a string with a length of 1.
- Square brackets can be used to access elements of the string.

## Creating a String

- Strings in Python can be created using single quotes ' ' or double quotes " " or even triple quotes ''' '.

In [ ]:

```
1  # Python Program for Creation of String
2
3  # Creating a String with single Quotes
4  String1 = 'Welcome to PYTHON World'
5  print("String with the use of Single Quotes: ")
6  print(String1)
7
8  # Creating a String with double Quotes
9  String1 = "I'm a LEARNER"
10 print("\nString with the use of Double Quotes: ")
11 print(String1)
12
13 # Creating a String with triple Quotes
14 String1 = '''Thinking Programming'''
15 print("\nString with the use of Triple Quotes: ")
16 print(String1)
17
18 # Creating String with triple Quotes allows multiple lines
19 String1 = '''Strings
20             In
21             Python'''
22 print("\nCreating a multiline String: ")
23 print(String1)
```

## Accessing characters in Python

- In Python, individual characters of a String can be accessed by using the method of Indexing.
- Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.
- While accessing an index out of the range will cause an IndexError.

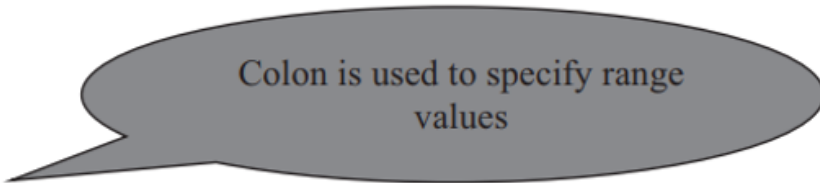
- Only Integers are allowed to be passed as an index, float or other types that will cause a `TypeError`.

P	Y	T	H	O	N	S	T	R	I	N	G	S
0	1	2	3	4	5	6	7	8	9	10	11	12
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
In [ ]: 1
        2 # Python Program to Access characters of String
        3
        4 String1 = "Access Characters of String"
        5 print("Initial String: ")
        6 print(String1)
        7
        8 # Printing First character
        9 print("\nFirst character of String is: ")
       10 print(String1[0])
       11
       12 # Printing Last character
       13 print("\nLast character of String is: ")
       14 print(String1[-1])
```

## String Slicing

- To access a range of characters in the String, the method of slicing is used.
- Slicing in a String is done by using a Slicing operator (colon :).



Colon is used to specify range values

*string\_name[start:end[:step]]*

```
In [ ]: 1 healthy_drink = "green tea"
        2 healthy_drink[0:3]
```

```
In [ ]: 1 healthy_drink[:5]
```

```
In [ ]: 1 healthy_drink[:]
```

```
In [ ]: 1 healthy_drink[2:2]
```

```
In [ ]: 1 healthy_drink[-3:-1]
```

```
In [ ]: 1 healthy_drink[6:-1]
```

```
In [ ]: 1 newspaper = "new york times"
        2 newspaper[0:12:4]
```

```
In [ ]: 1 newspaper[::4]
```

```
In [ ]: 1 # Python Program to demonstrate String slicing
2
3 # Creating a String
4 String1 = "String Slicing"
5 print("Initial String: ")
6 print(String1)
7
8 # Printing 3rd to 12th character
9 print("\nSlicing characters from 3-12: [INDEX VALUE FROM 3 TO 11 will be considered]")
10 print(String1[3:12])
11
12 # Printing characters between
13 # 3rd and 2nd last character
14 print("\nSlicing characters between " + "3rd and 2nd last character: [INDEX VALUE FROM 3 TO -3 will be considered] ")
15 print(String1[3:-2])
```

```
In [ ]: 1 #Accessing string characters in Python
2 str = 'programiz'
3 print('str = ', str)
4
5 #first character
6 print('str[0] = ', str[0])
7
8 #Last character
9 print('str[-1] = ', str[-1])
10
11 #slicing 2nd to 5th character
12 print('str[1:5] = ', str[1:5])
13
14 #slicing 6th to 2nd last character
15 print('str[5:-2] = ', str[5:-2])
```

# Strings are IMMUTABLE

- This means that **elements of a string cannot be changed once they have been assigned** . We can simply reassign different strings to the same name.



- We cannot delete or remove characters from a string.
- But deleting the string entirely is possible using the **del** keyword.

## Deleting/Updating from a String

- In Python, Updation or Deletion of characters from a String is not allowed.
- This will cause an error because item assignment or item deletion from a String is not supported.
- Although deletion of the entire String is possible with the use of a built-in del keyword. This is because Strings are immutable.

In [ ]:

```
1
2 # Python Program to Update character of a String----- Throws an ERROR
3
4 String1 = "Hello, I'm a Geek"
5 print("Initial String: ")
6 print(String1)
7
8 # Updating a character of the String----- Throws an ERROR
9 String1[2] = 'p'
10 print("\nUpdating character at 2nd Index: ")
11 print(String1)
```

In [ ]:

```
1 # Python Program to Update entire String
2
3 String1 = "Update entire String"
4 print("Initial String: ")
5 print(String1)
6
7 # Updating a String
8 String1 = "STRINGS IN PYTHON"
9 print("\nUpdated String: ")
10 print(String1)
```

In [ ]:

```
1 # Python Program to Delete characters from a String----- Throws an ERROR
2
3 String1 = "Delete characters from a String"
4 print("Initial String: ")
5 print(String1)
6
7 # Deleting a character of the String----- Throws an ERROR
8 del String1[2]
9 print("\nDeleting character at 2nd Index: ")
10 print(String1)
```

In [ ]:

```
1 # Python Program to Delete entire String
2
3 String1 = "Delete entire String"
4 print("Initial String: ")
5 print(String1)
6
7 # Deleting a String with the use of del
8 del String1
9 print("\nDeleting entire String: ")
10 print(String1)          #----- Throws an ERROR--- SINCE THE STRING GOT DELETED
```

# Escape Sequencing in Python

- While printing Strings with single and double quotes in it causes `SyntaxError` because String already contains Single and Double Quotes and hence cannot be printed with the use of either of these.
- Example1:-----**String1 = "'I'm a \"LEARNER\"'"**
- Example2:-----**String2 = "\"I'm a \"LEARNER\""**
- Hence, to print such a String either **Triple Quotes** are used or **Escape sequences** are used to print such Strings.
- Escape sequences start with a **backslash** and can be interpreted differently.
- If single quotes are used to represent a string, then all the single quotes present in the string must be escaped and same is done for Double Quotes.

```
In [ ]: 1 # Python Program for Escape Sequencing of String
2
3 # Initial String
4 String1 = '''I'm a "LEARNER"'''
5 print("Initial String with use of Triple Quotes: ")
6 print(String1)
7
8 # Escaping Single Quote
9 String1 = 'I\'m a "LEARNER"'
10 print("\nEscaping Single Quote: ")
11 print(String1)
12
13 # Escaping Double Quotes
14 String1 = "I'm a \"LEARNER\""
15 print("\nEscaping Double Quotes: ")
16 print(String1)
17
18 # Printing Paths with the
19 # use of Escape Sequences
20 String1 = "C:\\Python\\LEARNER\\"
21 print("\nEscaping Backslashes: ")
22 print(String1)
```

## Formatting of Strings

- Strings in Python can be formatted with the use of format() method which is a very versatile and powerful tool for formatting Strings.
- Format method in String contains curly braces {} as placeholders which can hold arguments according to position or keyword to specify the order.

In [ ]:

```
1 # Python Program for Formatting of Strings
2
3 # Default order
4 String1 = "{} {} {}".format('Formatting', 'of', 'Strings')
5 print("Print String in default order: ")
6 print(String1)
7
8 # Positional Formatting
9 String1 = "{1} {0} {2}".format('Formatting', 'of', 'Strings')
10 print("\nPrint String in Positional order: ")
11 print(String1)
12
13 # Keyword Formatting
14 String1 = "{f} {o} {s}".format(f='Formatting', o='of', s='Strings')
15 print("\nPrint String in order of Keywords: ")
16 print(String1)
```

- Integers such as Binary, hexadecimal, etc., and floats can be rounded or displayed in the exponent form with the use of format specifiers.

In [ ]:

```
1 # Formatting of Integers
2 String1 = "{0:b}".format(16)
3 print("\nBinary representation of 16 is ")
4 print(String1)
5
6 # Formatting of Floats
7 String1 = "{0:e}".format(165.6458)
8 print("\nExponent representation of 165.6458 is ")
9 print(String1)
10
11 # Rounding off Integers
12 String1 = "{0:.2f}".format(1/6)
13 print("\none-sixth is : ")
14 print(String1)
```

- A string can be left() or center(^) justified with the use of format specifiers, separated by a colon(:).

In [ ]:

```
1 # String alignment
2 String1 = "|{:<10}|{: ^10}|{:>10}|".format('Python', 'String', 'Alignment')
3 print("\nLeft, center and right alignment with Formatting: ")
4 print(String1)
5
6 # To demonstrate aligning of spaces
7 String1 = "\n{0: ^16} was founded in {1: <4}!".format("PYTHON STRINGS", 2021)
8 print(String1)
```

- Old style formatting was done without the use of format method by **using % operator**.

In [ ]:

```
1 # Python Program for # Old Style Formatting of Integers
2
3 Integer1 = 12.3456789
4 print("Formatting in 3.2f format: ")
5 print('The value of Integer1 is %3.2f' % Integer1)
6 print("\nFormatting in 3.4f format: ")
7 print('The value of Integer1 is %3.4f' % Integer1)
```

- Python provides many functions that are built into the interpreter and always available.

Function	Description
chr()	Converts an integer to a character
ord()	Converts a character to an integer
len()	Returns the length of a string
str()	Returns a string representation of an object

```
In [ ]: 1 #CONVERTING CHARACTER TO AN INTEGER
        2 ord('a')
        3
```

```
In [ ]: 1 #CONVERTING CHARACTER TO AN INTEGER
        2 ord(' ')
```

```
In [ ]: 1 #CONVERTING AN INTEGER TO CHARACTER
        2 chr(97)
```

```
In [ ]: 1 #CONVERTING AN INTEGER TO CHARACTER
        2 chr(65)
```

```
In [ ]: 1 #FINDING THE LENGTH OF A STRING
        2 s = 'BUILT-IN FUNCTIONS IN STRINGS'
        3 len(s)
```

```
In [ ]: 1 #Write Python Code to Determine Whether the Given String Is a Palindrome or Not Using Slicing
        2 def main():
        3     user_string = input("Enter string: ")
        4     if user_string == user_string[::-1]:
        5         print(f"User entered string {user_string} is palindrome")
        6     else:
        7         print(f"User entered string {user_string} is not a palindrome")
        8 if __name__ == "__main__":
        9     main()
```

## Basic String Operations

- In Python, strings can also be concatenated using + sign and \* operator is used to create a repeated sequence of strings.
- Python cannot concatenate string value with integer value since they are of different data types. You need to convert integer type to string type before concatenating integer and string values.

```
In [ ]: 1 string_1 = "face"
        2 string_2 = "book"
        3 concatenated_string = string_1 + string_2
        4 concatenated_string
```

```
In [ ]: 1 repetition_of_string = "wow" * 5
        2 repetition_of_string
```



# check for the presence of a string in another string

- using **in** and **not in** membership operators.
- It returns either a Boolean True or False.
- The **in** operator evaluates to True if the string value in the left operand **appears** in the sequence of characters of string value in right operand.
- The **not in** operator evaluates to True if the string value in the left operand **does not appear** in the sequence of characters of string value in right operand.

```
In [ ]: 1 fruit_string = "apple is a fruit"
        2 fruit_sub_string = "apple"
        3 fruit_sub_string in fruit_string
```

```
In [ ]: 1 another_fruit_string = "orange"
        2 another_fruit_string not in fruit_string
```

## String Comparison

- You can use

(>, <, <=, >=, ==, !=)

to compare two strings resulting in either Boolean True or False value.

- Python compares strings using ASCII value of the characters.

```
In [ ]: 1 "january" == "jane"
```

```
In [ ]: 1 "january" != "jane"
```

```
In [ ]: 1 "january" < "jane"
```

```
In [ ]: 1 "january" > "jane"
```

```
In [ ]: 1 "january" <= "jane"
```

```
In [ ]: 1 "january" >= "jane"
```

## Joining Strings Using join() Method

- The join() method provides a flexible way to concatenate strings
- SYNTAX: **string\_name.join(sequence)**

```
In [ ]: 1 numbers = "123"  
2 characters = "amy"  
3 password = numbers.join(characters)  
4 password
```

# Split Strings Using split() Method

- The split() method returns a list of string items by breaking up the string using the delimiter string.
- SYNTAX: **string\_name.split(separator,maxsplit)**

```
In [ ]: 1 inventors = "edison, tesla, marconi, newton"
        2 inventors.split(",")           #OUTPUT : ['edison', ' tesla', ' marconi', ' newton']
```

```
In [ ]: 1 watches = "rolex hublot cartier omega"
        2 watches.split()               #OUTPUT : ['rolex', 'hublot', 'cartier', 'omega']
```

```
In [ ]: 1 txt = "apple#banana#cherry#orange"
        2 # setting the maxsplit parameter to 1, will return a list with 2 elements!
        3 x = txt.split("#", 1)
        4 print(x)
```

```
In [ ]: 1 #Program to Demonstrate String Traversing Using the for Loop
        2 def main():
        3     alphabet = "google"
        4     index = 0
        5     print(f"In the string '{alphabet}'")
        6     for each_character in alphabet:
        7         print(f"Character '{each_character}' has an index value of {index}")
        8         index += 1
        9     if __name__ == "__main__":
       10     main()
```

In [ ]:

```
1 #Write Python Program to Count the Total Number of Vowels,Consonants and Blanks in a String
2 def main():
3     user_string = input("Enter a string: ")
4     vowels = 0
5     consonants = 0
6     blanks = 0
7     for each_character in user_string:
8         if(each_character == 'a' or each_character == 'e' or each_character == 'i' or each_character == 'o' or each_
9             vowels += 1
10        elif "a" < each_character < "z":
11            consonants += 1
12        elif each_character == " ":
13            blanks += 1
14        print(f"Total number of Vowels in user entered string is {vowels}")
15        print(f"Total number of Consonants in user entered string is {consonants}")
16        print(f"Total number of Blanks in user entered string is {blanks}")
17 if __name__ == "__main__":
18     main()
```

In [ ]:

```
1 #Write Python Program to Calculate the Length of a String Without Using Built-In Len() Function
2 def main():
3     user_string = input("Enter a string: ")
4     count_character = 0
5     for each_character in user_string:
6         count_character += 1
7     print(f"The length of user entered string: {user_string} is {count_character} ")
8 if __name__ == "__main__":
9     main()
```

# LISTS IN PYTHON

- Lists are used to store **multiple items** in a single variable.

- Lists are one of 4 **built-in data types** in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
- A list can have any number of items and they may be of **different types (integer, float, string, etc.)**.
- Lists are created using **square brackets**.
- Lists are **mutable** means, any data item of a List can be replaced by any other data item.
- For example:
  - `list1 = ['physics', 'chemistry', 1997, 2000];`
  - `list2 = [1, 2, 3, 4, 5];`
  - `list3 = ["a", "b", "c", "d"]`

## Creating a List

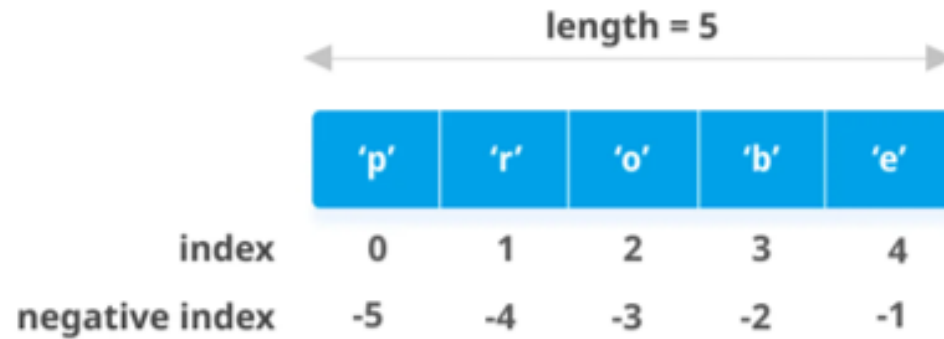
- Lists in Python can be created by just placing the sequence inside the square brackets[].
- A list doesn't need a built-in function for the creation of a list.

In [ ]:

```
1
2 # Python program to demonstrate Creation of List
3
4 # Creating a List
5 List = []
6 print("Blank List: ")
7 print(List)
8
9 # Creating a List of numbers
10 List = [10, 20, 14]
11 print("\nList of numbers: ")
12 print(List)
13
14 # List with mixed data types
15 List = [1, "Hello", 3.4]
16 print("\nList of mixed data types")
17 print(List)
18
19 # Creating a List of strings and accessing using index
20 List = ["Creation", "of", "List"]
21 print("\nList Items: ")
22 print(List[0])
23 print(List[2])
24
25 # Creating a Multi-Dimensional List (By Nesting a List inside a List)
26 List = [['Creation', 'of'], ['List']]
27 print("\nMulti-Dimensional List OR Nested List: ")
28 print(List)
```

# Access List Elements

- Elements of a list can be accessed using INDEXING



List indexing in Python

```
In [ ]: 1 my_list = ['p', 'r', 'o', 'b', 'e']
        2
        3 # first item
        4 print(my_list[0]) # p
        5
        6 # third item
        7 print(my_list[2]) # o
        8
        9 # fifth item
       10 print(my_list[4]) # e
       11
       12 # Nested List
       13 n_list = ["Happy", [2, 0, 1, 5]]
       14
       15 # Nested indexing
       16 print(n_list[0][1])
       17
       18 print(n_list[1][3])
       19
       20 # Error! Only integer can be used for indexing
       21 print(my_list[4.0])
```

```
In [ ]: 1 # Negative indexing in lists
2 my_list = ['p','r','o','b','e']
3
4 # last item
5 print(my_list[-1])
6
7 # fifth last item
8 print(my_list[-5])
```

## List Slicing using the slicing operator :

- We can access a range of items in a list by using the slicing operator :.

```
In [ ]: 1 # List slicing in Python
2
3 my_list = ['p','r','o','g','r','a','m','i','z']
4
5 # elements from index 2 to index 4
6 print(my_list[2:5])
7
8 # elements from index 5 to end
9 print(my_list[5:])
10
11 # elements beginning to end
12 print(my_list[:])
```

## List Slicing using assignment operator

- Lists are mutable, meaning their elements can be changed.  
We can use the assignment operator = to change an item or a range of items.



```
In [ ]: 1 # Correcting mistake values in a list
        2 odd = [2, 4, 6, 8]
        3
        4 # change the 1st item
        5 odd[0] = 1
        6
        7 print(odd)
        8
        9 # change 2nd to 4th items
       10 odd[1:4] = [3, 5, 7]
       11
       12 print(odd)
```

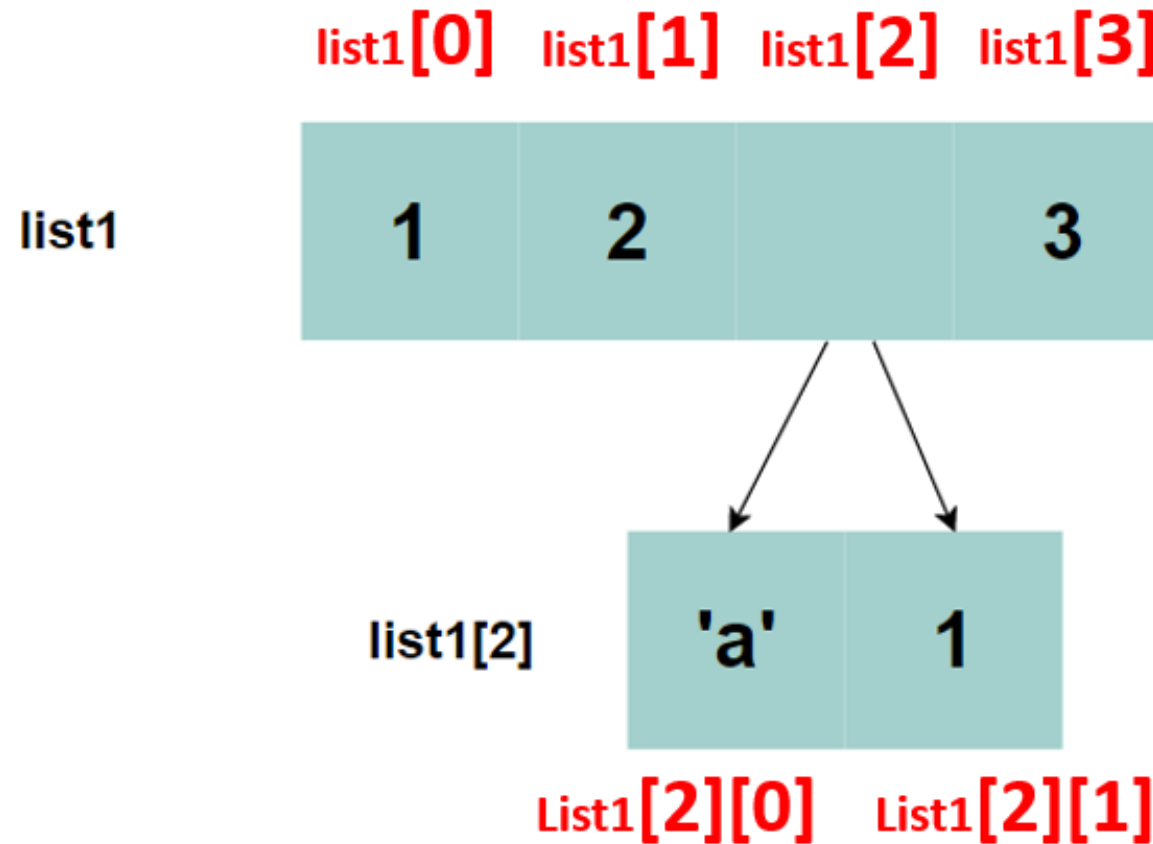
## Appending and Extending lists

- We can add one item to a list using the **append( )** method or add several items using the **extend( )** method.

```
In [ ]: 1 # Appending and Extending Lists in Python
        2 odd = [1, 3, 5]
        3
        4 odd.append(7)
        5
        6 print(odd)
        7
        8 odd.extend([9, 11, 13])
        9
       10 print(odd)
```

## Nested Lists in Python

- A nested list is a list of lists, or any list that has another list as an element (a sublist).



- They can be helpful if you want to create a matrix or need to store a sublist along with other data types.

## Creating Nested Lists

- nestedList = [1, 2, ['a', 1], 3]
- L = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', 'h']

In [1]:

```
1 # creating list
2 nestedList = [1, 2, ['a', 1], 3]
3
4 # indexing list: the sublist has now been accessed
5 subList = nestedList[2]
6
7 # access the first element inside the inner list:
8 element = nestedList[2][0]
9
10 print("List inside the nested list: ", subList)
11 print("First element of the sublist: ", element)
```

List inside the nested list: ['a', 1]

First element of the sublist: a

In [23]:

```
1 L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
2
3 print(L[2])
4 # Prints ['cc', 'dd', ['eee', 'fff']]
5
6 print(L[2][2])
7 # Prints ['eee', 'fff']
8
9 print(L[2][2][0])
```

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

['eee', 'fff']

eee

- Find Nested List Length.

```
In [21]: 1 # List of lists
2 listOfElems2D = [ [1,2,3,45,6,7],
3                  [22,33,44,55],
4                  [11,13,14,15] ]
5 length = len(listOfElems2D)
6 print('Number of lists in list = ', length)
```

Number of lists in list = 3

```
In [24]: 1 L = ['a', ['bb', 'cc'], 'd']
2
3 print(len(L))
4 # Prints 3
5
6 print(len(L[1]))
7 # Prints 2
```

3  
2

## Creating a matrix using Nested List

- Creating a matrix is one of the most useful applications of nested lists.

**list1 =**

```
list1 = [[0, 1, 2],  
         [3, 4, 5],  
         [6, 7, 8]]
```

	col 1	col 2	col 3	
row 1	list1[0][0] 0	list1[0][1] 1	list1[0][2] 2	= list1[0]
row 2	list1[1][0] 3	list1[1][1] 4	list1[1][2] 5	= list1[1]
row 3	list1[2][0] 6	list1[2][1] 7	list1[2][2] 8	= list1[2]

- They can be helpful if you want to create a matrix or need to store a sublist along with other data types.
- The number of elements in the nested lists is equal to the number of rows of the matrix.
- Thus, each element of the nested list (matrix) will have two indices: the row and the column.

In [22]:

```
1 # create matrix of size 4 x 3
2 matrix = [[0, 1, 2],
3           [3, 4, 5],
4           [6, 7, 8],
5           [9, 10, 11]]
6
7 rows = len(matrix) # no of rows is no of sublists i.e. len of list
8 cols = len(matrix[0]) # no of cols is len of sublist
9
10 # printing matrix
11 print("matrix: ")
12 for i in range(0, rows):
13     print(matrix[i])
14
15 # accessing the element on row 2 and column 1 i.e. 3
16 print("element on row 2 and column 1: ", matrix[1][0])
17
18 # accessing the element on row 3 and column 2 i.e. 7
19 print("element on row 3 and column 2: ", matrix[2][1])
```

matrix:

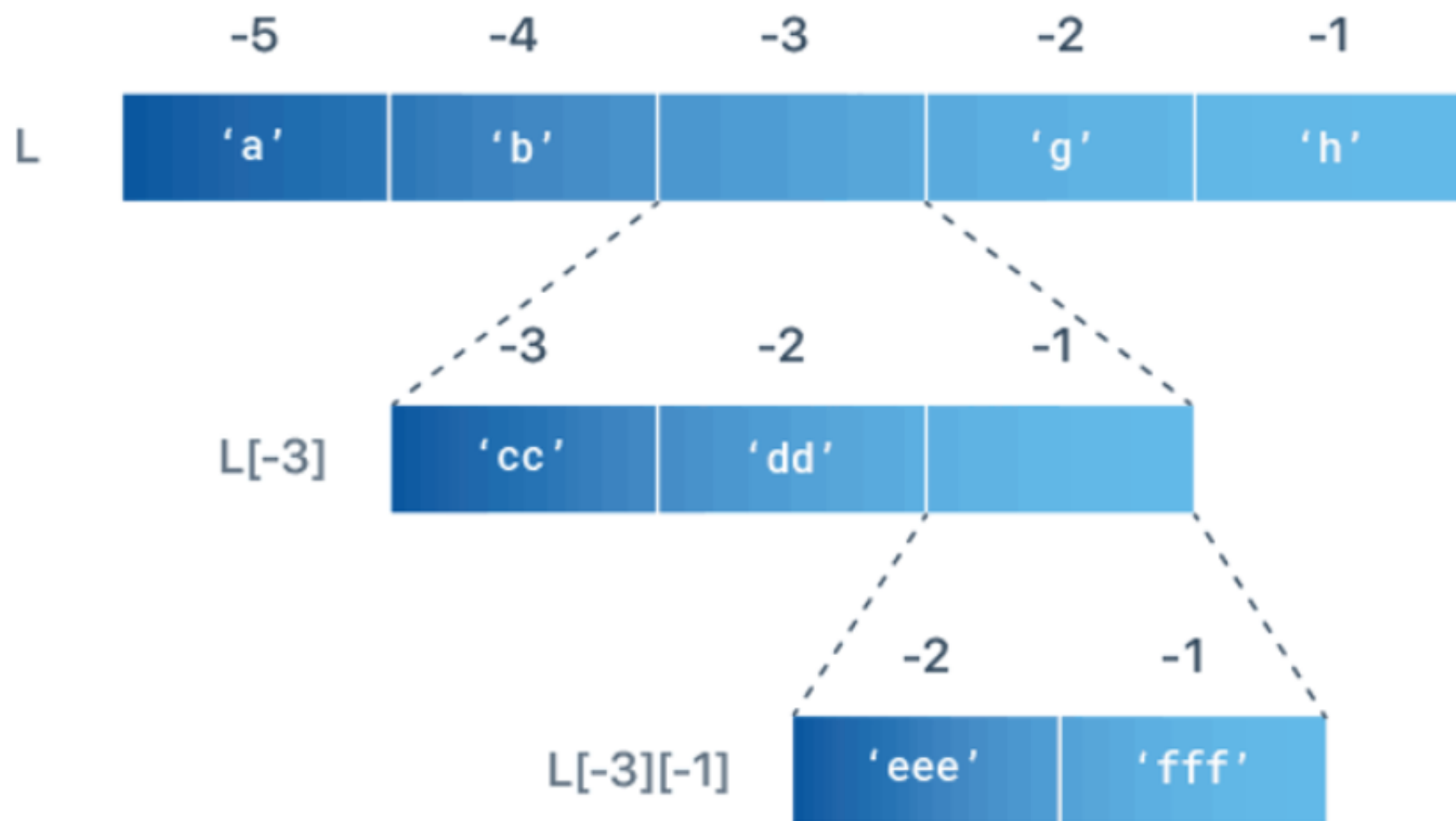
```
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
[9, 10, 11]
```

element on row 2 and column 1: 3

element on row 3 and column 2: 7

## Negative List Indexing In a Nested List

- 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.
- The negative indexes for the items in a nested list are illustrated as below:



```
In [ ]: 1 L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
2
3 print(L[-3])
4 # Prints ['cc', 'dd', ['eee', 'fff']]
5
6 print(L[-3][-1])
7 # Prints ['eee', 'fff']
8
9 print(L[-3][-1][-2])
10 # Prints eee
```

## • Add items to a Nested list---Using `append()` method

- To add new values to the end of the nested list, use **`append()`** method.

```
In [25]: 1 L = ['a', ['bb', 'cc'], 'd']
2 L[1].append('xx')
3 print(L)
4 # Prints ['a', ['bb', 'cc', 'xx'], 'd']
```

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

## • Add items to a Nested list---Using `insert()` method

- When you want to insert an item at a specific position in a nested list, use **`insert()`** method.



```
In [ ]: 1 L = ['a', ['bb', 'cc'], 'd']
        2 L[1].insert(0, 'xx')
        3 print(L)
        4 # Prints ['a', ['xx', 'bb', 'cc'], 'd']
```

## • Add items to a Nested list--- Using `extend()` method

- You can merge one list into another by using **`extend()`** method.

```
In [26]: 1 L = ['a', ['bb', 'cc'], 'd']
        2 L[1].extend([1,2,3])
        3 print(L)
        4 # Prints ['a', ['bb', 'cc', 1, 2, 3], 'd']
```

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

## • Remove items from a Nested List--- Using `pop()` method

- If you know the index of the item you want, you can use **`pop()`** method.
- It modifies the list and returns the removed item.

In [27]:

```
1 L = ['a', ['bb', 'cc', 'dd'], 'e']
2 x = L[1].pop(1)
3 print(L)
4 # Prints ['a', ['bb', 'dd'], 'e']
5
6 # removed item
7 print(x)
8 # Prints cc
```

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

## • Remove items from a Nested List--- Using del() method

- If you don't need the removed value, use the **del** statement.

In [2]:

```
1 L = ['a', ['bb', 'cc', 'dd'], 'e']
2 del L[1][1]
3 print(L)
4 # Prints ['a', ['bb', 'dd'], 'e']
```

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

## • Remove items from a Nested List--- Using remove() method

- If you're not sure where the item is in the list, use **remove()** method to delete it by value.

In [28]:

```
1 L = ['a', ['bb', 'cc', 'dd'], 'e']  
2 L[1].remove('cc')  
3 print(L)  
4 # Prints ['a', ['bb', 'dd'], 'e']
```

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

I want to create a matrix which looks like below:

```
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 [33]:

```
1 matrix = []
2
3
4 for i in range(5):
5
6     # Append an empty sublist inside the list
7     matrix.append([])
8
9     for j in range(5):
10         matrix[i].append(j)
11     print(matrix[i])
12 #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]
```

## MODULES IN PYTHON

- Modules are simply files with the “.py” extension containing Python code.
- Modules can be imported inside another Python Program.

### • Python import statement

- We can import a module using the import statement and access the definitions inside it using the dot operator as described above.

### • import math module

## import math module

```
In [4]: 1 # import statement example to import standard module math
        2
        3 import math
        4 print("The value of pi is", math.pi)
```

The value of pi is 3.141592653589793

```
In [5]: 1 # importing sqrt() and factorial from the module math
        2 from math import sqrt, factorial
        3
        4 # if we simply do "import math", then math.sqrt(16) and math.factorial() are required.
        5 print(sqrt(16))
        6 print(factorial(6))
```

4.0  
720

- **import random module**

- **random.randint()** method is used to generate random integers between the given range.








In [6]:

```
1  # Python3 program explaining work
2  # of randint() function
3
4  # import random module
5  import random
6
7  # Generates a random number between
8  # a given positive range
9  r1 = random.randint(5, 15)
10 print("Random number between 5 and 15 is % s" % (r1))
11
12 # Generates a random number between
13 # two given negative range
14 r2 = random.randint(-10, -2)
15 print("Random number between -10 and -2 is % d" % (r2))
```

Random number between 5 and 15 is 14

Random number between -10 and -2 is -9

- # Implementing Python Emoji module
- There are two ways of implementing emojis in our text, data, or applications.
- **1. Using unicode as the format**
- **2.Emoji module functions to have emojis in the data.**

face-smiling			
<u>No</u>	<u>Code</u>	<u>Browser</u>	<u>CLDR Short Name</u>
1	U+1F600		grinning face
2	U+1F603		grinning face with big eyes
3	U+1F604		grinning face with smiling eyes
4	U+1F601		beaming face with smiling eyes
5	U+1F606		grinning squinting face
6	U+1F605		grinning face with sweat
7	U+1F923		rolling on the floor laughing

- Implementing Python Emoji---  
Using unicode as the format

In [10]:

```
1 print("\U0001F600 \U0001F603")
2 print("\U0001F601 \U0001F923")
3 print("\U0001F680 \U0001F60E")
```



# • Implementing Python Emoji --- Using Emoji module functions

- **import emoji module**
- Emojis can be implemented by using the emoji module provided in Python.
- **Install Emoji Package Using pip install emoji**
- **EXAMPLE :**
- **from emoji import emojize**
- **print(emojize(":thumbs\_up:"))**.



In [5]: 1 pip install emoji

```
Collecting emoji
  Downloading emoji-1.6.1.tar.gz (170 kB)
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Building wheels for collected packages: emoji
  Building wheel for emoji (setup.py): started
  Building wheel for emoji (setup.py): finished with status 'done'
  Created wheel for emoji: filename=emoji-1.6.1-py3-none-any.whl size=169314 sha256=82d30ab88894fd3a25ce9cd53d577f499b41406f39a09edbe6500b268bcad9af
  Stored in directory: c:\users\asha rani k p\appdata\local\pip\cache\wheels\3d\c9\af\02caa5725634f27f4e2e43852f67fc9069d014038b236a827e
Successfully built emoji
Installing collected packages: emoji
Successfully installed emoji-1.6.1
Note: you may need to restart the kernel to use updated packages.
```

In [6]: 1 *# Install Emoji Package Using*  
2 *#pip install emoji*  
3  
4 **from** emoji **import** emojiize  
5 **print**(emojiize(":thumbs\_up:"))



In [7]: 1 *# import emoji module*  
2 **import** emoji  
3  
4  
5 **print**(emoji.emojiize(":grinning\_face\_with\_big\_eyes:"))  
6 **print**(emoji.emojiize(":winking\_face\_with\_tongue:"))  
7 **print**(emoji.emojiize(":zipper-mouth\_face:"))



-----

# ARRAYS IN PYTHON

- A Python Array is a collection of common type of data structures having elements with same data type.
- It is used to store collections of data. In Python programming, an arrays are handled by the “**array**” **module**.
- If you create arrays using the array module, elements of the array must be of the same numeric type.
- Python arrays are used when you need to use many variables which are of the same type.

## Syntax to Create an Array in Python

- **SYNTAX:**
- STEP1: IMPORT ARRAY MODULE AS----**import array as arr**
- STEP2: CREATING AN ARRAY a-----**a=arr.array(data type,value list)**
- **EXAMPLE:**

- `import array as myarray`
- `abc = myarray.array('d', [2.5, 4.9, 6.7])`

## • Creating Python Arrays

- To create an array of numeric values, we need to import the array module.

```
In [1]: 1 import array as arr
        2 a = arr.array('d', [1.1, 3.5, 4.5])
        3 print(a)
```

```
array('d', [1.1, 3.5, 4.5])
```

## • Accessing Python Array Elements

- We use indices to access elements of an array:

```
In [2]: 1 import array as arr
        2 a = arr.array('i', [2, 4, 6, 8])
        3
        4 print("First element:", a[0])
        5 print("Second element:", a[1])
        6 print("Last element:", a[-1])
```

```
First element: 2
Second element: 4
Last element: 8
```

# • Slicing Python Arrays

- We can access a range of items in an array by using the slicing operator :

In [3]:

```
1 import array as arr
2
3 numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]
4 numbers_array = arr.array('i', numbers_list)
5
6 print(numbers_array[2:5]) # 3rd to 5th
7 print(numbers_array[:-5]) # beginning to 4th
8 print(numbers_array[5:]) # 6th to end
9 print(numbers_array[:]) # beginning to end
```

```
array('i', [62, 5, 42])
array('i', [2, 5, 62])
array('i', [52, 48, 5])
array('i', [2, 5, 62, 5, 42, 52, 48, 5])
```

## CASE STUDY

- **TITLE:** CELEBRITY PROBLEM
- **AIM:** The celebrity problem is the problem of finding the celebrity among n people. A celebrity is someone who does not know anyone (including themselves) but is known by everyone.
- **LEARNING OBJECTIVES:**
- Implement built-in methods, user-defined functions

- Implement Python Basics such as variables, functions, keywords, control flow statements
- Importing modules and using the methods.

## Celebrity Identification Steps

Step 1. The program uses a matrix such that `matrix[i][j]` is `True` if and only if `i` knows `j`.

Step 2. Create two functions `eliminate_non_celebrities()` and `check_if_celebrity()`.

Step 3. The function `eliminate_non_celebrities()` returns a candidate who maybe a celebrity. It takes the *matrix* and *n* as argument.

Step 4. The function `check_if_celebrity()` determines whether a person is a celebrity. It takes the *matrix*, *possible\_celebrity* and *n* as argument.

Step 5. The function `eliminate_non_celebrities()` works on the principle that if `matrix[i][j] = True`, that is `i` knows `j`, then `i` cannot be the celebrity and if `matrix[i][j] = False`, that is `i` does not know `j`, then `j` cannot be the celebrity.

Step 6. The function `check_if_celebrity()` whether *possible\_celebrity* is known by everyone else and whether *possible\_celebrity* does not know anyone. If so, it returns `True`.

