Programming in Python with Full Stack Development (303105257)

Anand Jawdekar

Assistant Professor CSE Department

Unit-4

Introduction to python programing:

Introduction to Python and Basic Programming Concepts

Python is a high-level, versatile, and user-friendly programming language widely used in various fields, including web development, data analysis, machine learning, and automation. Its simple syntax and readability make it an excellent choice for beginners.

In the late 1980s, Guido van Rossum dreamed of developing Python. The first version of Python 0.9.0 was released in 1991. Since its release, Python started gaining popularity. According to reports, Python is now the most popular programming language among developers because of its high demands in the tech realm..

Features of Python

Easy to use and Read - Python's syntax is clear and easy to read, making it an ideal language for both beginners and experienced programmers. This simplicity can lead to faster development and reduce the chances of errors.
Dynamically Typed - The data types of variables are determined during run-time. We do not need to specify the data type of a variable during writing codes.
High-level - High-level language means human readable code.
Compiled and Interpreted - Python code first gets compiled into bytecode, and then interpreted line by line. When we download the Python in our system form pyhton.org we download the default implement of Python known as CPython. CPython is considered to be Complied and Interpreted both.

Features

Garbage Collected - Memory allocation and de-allocation are automatically managed. Programmers do not specifically need to manage the memory.

Purely Object-Oriented - It refers to everything as an object, including numbers and strings.

Cross-platform Compatibility - Python can be easily installed on Windows, macOS, and various Linux distributions, allowing developers to create software that runs across different operating systems.

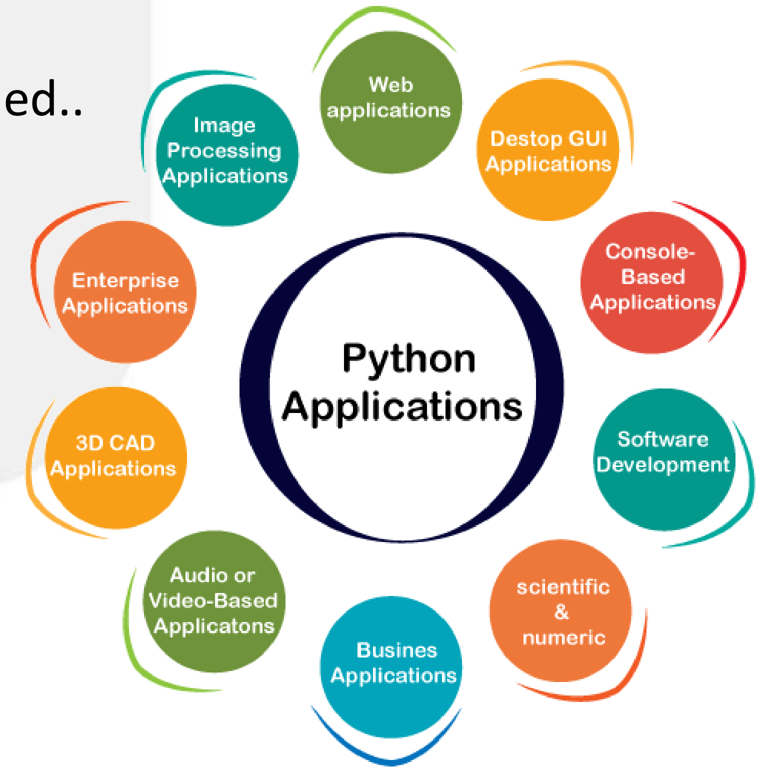
Rich Standard Library - Python comes with several standard libraries that provide ready-to-use modules and functions for various tasks, ranging from **web development** and **data manipulation** to **machine learning** and **networking**.

Open Source - Python is an open-source, cost-free programming language. It is utilized in several sectors and disciplines as a result.

Python Applications

Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application.

Here, we are specifying application areas where Python can be applied..



Python Variables

A variable is the name given to a memory location. A value-holding Python variable is also known as an identifier.

Since Python is an infer language that is smart enough to determine the type of a variable, we do not need to specify its type in Python.

Variable names must begin with a letter or an underscore, but they can be a group of both letters and digits.

The name of the variable should be written in lowercase. Both Rahul and rahul are distinct variables.

Identifier Naming

The variable's first character must be an underscore or alphabet (_).
Every one of the characters with the exception of the main person might be a letter set of lower-case(a-z), capitalized (A-Z), highlight, or digit (0-9).
White space and special characters (!, @, #, %, etc.) are not allowed in the identifier name. ^, &, *).
Identifier name should not be like any watchword characterized in the language.
Names of identifiers are case-sensitive; for instance, my name, and MyName isn't something very similar.
Examples of valid identifiers: a123, _n, n_9, etc.
Examples of invalid identifiers: 1a, n%4, n 9, etc.

Declaring Variable and Assigning Values

Python doesn't tie us to pronounce a variable prior to involving it in the application. It permits us to make a variable at the necessary time.

In Python, we don't have to explicitly declare variables. The variable is declared automatically whenever a value is added to it.

The equal (=) operator is utilized to assign worth to a variable.

Python Operators

The operator is a symbol that performs a specific operation between two operands, according to one definition. Operators serve as the foundation upon which logic is constructed in a program in a particular programming language. In every programming language, some operators perform several tasks. Same as other languages, Python also has some operators, and these are given below –

- Arithmetic operators
- Comparison operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators
- Arithmetic Operators

Basic Programming Concepts

1. VariablesDefinition: A variable is a named storage location for data in memory.Syntax

variable_name = value

Data Types

2. Python offers a variety of built-in data types to handle different kinds of data. These types are categorized as Numeric, Sequence, Mapping, Set, Boolean, and Special types. Python is dynamically typed, so you don’t need to specify the type of a variable explicitly.

Cont..

1. Numeric Types

These are used to store numerical values.

int: Integer values (e.g., 1, 42, -5)

float: Floating-point numbers (e.g., 3.14, -0.5)

complex: Complex numbers (e.g., 2 + 3j, -1j)

Example

```
x = 10          # int
y = 3.14        # float
z = 2 + 3j      # complex
```

‘cont..

2. Sequence Types

Sequence types represent collections of items.

a) str (String)

A sequence of Unicode characters.

Strings are immutable.

Example

```
greeting = "Hello, World!"
print(greeting[0])  # Output: 'H'
```

b) list

A mutable sequence of items of any data type.

Example

```
fruits = ["apple", "banana", "cherry"]
fruits.append("date")  # Adds 'date' to the list
```

Cont..

c) tuple

An immutable sequence of items.

Examples:

python

coordinates = (10, 20, 30) # coordinates[0] = 40 # This will raise an error

d) range

Represents a sequence of numbers, often used in loops.

Examples:

for i in range(1, 5): # Output: 1, 2, 3, 4 print(i)

Cont..

3. Mapping Type

dict (Dictionary)

Stores key-value pairs, where keys are unique.

Examples:

python

student = {"name": "Alice", "age": 20, "grade": "A"} print(student["name"]) # Output: Alice

Cont..

5. Boolean Type

bool

Represents logical values: True or False.

Examples:

python

x = 10 > 5 # Output: True y = bool(0) # Output: False

Cont..

4. Set Types

a) set

Unordered collection of unique items.

Examples:

python

numbers = {1, 2, 3, 3} # Output: {1, 2, 3}

b) frozenset

Immutable version of a set.

Examples:

python

frozen = frozenset([1, 2, 3]) # frozen.add(4) # This will raise an error

Steps:

6. Special Data Type

NoneType

Represents the absence of a value or a null value.

Examples:

python

value = None if value is None: print("No value assigned.")

.

Summary Table of Python Data Types

Category	Data Type	Example
Numeric	int	10
	float	3.14
	complex	2 + 3j
Sequence	str	"Hello"
	list	[1, 2, 3]
	tuple	(1, 2, 3)
Mapping	dict	{"key": "value"}
Set	set	{1, 2, 3}
	frozenset	frozenset([1, 2])
Boolean	bool	True or False
Special	NoneType	None

Python Keywords

Every scripting language has designated words or keywords, with particular definitions and usage guidelines. Python is no exception. The fundamental constituent elements of any Python program are Python keywords.

Python keywords are unique words reserved with defined meanings and functions that we can only apply for those functions. You'll never need to import any keyword into your program because they're permanently present.

Python's built-in methods and classes are not the same as the keywords. Built-in methods and classes are constantly present; however, they are not as limited in their application as keywords.

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

Python Comments

Single-Line Comments

```
# This code is to show an example of a single-line comment
print( 'This statement does not have a hashtag before it' )
```

Multi-Line Comments

```
# it is a
# comment
# extending to multiple lines
```

Python If-else statements

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
Nested if Statement	Nested if statements enable us to use if ? else statement inside an outer if statement.

Indentation in Python

For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

The syntax of the if-statement is given below.

```
if expression:
    statement
```

Example 1

```
# Simple Python program to understand the if statement
num = int(input("enter the number:"))
# Here, we are taking an integer num and taking input dynamically
if num%2 == 0:
# Here, we are checking the condition. If the condition is true, we will enter the block
    print("The Given number is an even number")
```

Output:

```
enter the number: 10 The Given number is an even number
```

Parul® University

DIGITAL LEARNING CONTENT

Python Loops

The following loops are available in Python to fulfil the looping needs. Python offers 3 choices for running the loops. The basic functionality of all the techniques is the same, although the syntax and the amount of time required for checking the condition differ.
We can run a single statement or set of statements repeatedly using a loop command.
The following sorts of loops are available in the Python programming language.

Cont..

Sr.No.	Name of the loop	Loop Type & Description
1	**While loop**	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	**For loop**	This type of loop executes a code block multiple times and abbreviates the code that manages the loop variable.
3	**Nested loops**	We can iterate a loop inside another loop.

Cont..

Sr.No.	Name of the control statement	Description
1	**Break statement**	This command terminates the loop's execution and transfers the program's control to the statement next to the loop.
2	**Continue statement**	This command skips the current iteration of the loop. The statements following the continue statement are not executed once the Python interpreter reaches the continue statement.
3	**Pass statement**	The pass statement is used when a statement is syntactically necessary, but no code is to be executed.

For loop

The for Loop

Python's for loop is designed to repeatedly execute a code block while iterating through a list, tuple, dictionary, or other iterable objects of Python. The process of traversing a sequence is known as iteration.

Syntax of the for Loop

```
for value in sequence:
    { code block }
```

In this case, the variable value is used to hold the value of every item present in the sequence before the iteration begins until this particular iteration is completed.

Loop iterates until the final item of the sequence are reached.

For loop example

```
# Python program to show how the for loop works

# Creating a sequence which is a tuple of numbers
numbers = [4, 2, 6, 7, 3, 5, 8, 10, 6, 1, 9, 2]

# variable to store the square of the number
square = 0

# Creating an empty list
squares = []

# Creating a for loop
for value in numbers:
   square = value ** 2
   squares.append(square)
print("The list of squares is", squares)
```

Output:

```
The list of squares is [16, 4, 36, 49, 9, 25, 64, 100, 36, 1, 81, 4]
```

Using else Statement with for Loop

As already said, a for loop executes the code block until the sequence element is reached. The statement is written right after the for loop is executed after the execution of the for loop is complete.
Only if the execution is complete does the else statement comes into play. It won't be executed if we exit the loop or if an error is thrown.
Here is a code to better understand if-else statements.

Example

```
# Python program to show how if-else statements work

string = "Python Loop"

# Initiating a loop
for s in a string:
    # giving a condition in if block
    if s == "o":
        print("If block")
    # if condition is not satisfied then else block will be executed
    else:
        print(s)
```

Output:

```
P y t h If block n L If block If block p
```

The range() Function

With the help of the range() function, we may produce a series of numbers. range(10) will produce values between 0 and 9. (10 numbers).

We can give specific start, stop, and step size values in the manner range(start, stop, step size). If the step size is not specified, it defaults to 1.

Since it doesn't create every value it "contains" after we construct it, the range object can be characterized as being "slow." It does provide in, len, and __getitem__ actions, but it is not an iterator.

The example that follows will make this clear.

Example

```
# Python program to show the working of range() function

print(range(15))

print(list(range(15)))

print(list(range(4, 9)))

print(list(range(5, 25, 4)))
```

Output:

```
range(0, 15)
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
 [4, 5, 6, 7, 8]
[5, 9, 13, 17, 21]
```

Python While Loops

The Python while loop iteration of a code block is executed as long as the given Condition, i.e., conditional_expression, is true.

If we don't know how many times we'll execute the iteration ahead of time, we can write an indefinite loop.

Syntax of Python While Loop

```
Statement
while Condition:
    Statement
```

The given condition, i.e., conditional_expression, is evaluated initially in the Python while loop. Then, if the conditional expression gives a boolean value True, the while loop statements are executed. The conditional expression is verified again when the complete code block is executed. This procedure repeatedly occurs until the conditional expression returns the boolean value False.

The statements of the Python while loop are dictated by indentation.

The code block begins when a statement is indented & ends with the very first unindented statement. Any non-zero number in Python is interpreted as boolean True. False is interpreted as None and 0.

Example

Program code 1:

Now we give code examples of while loops in Python for printing numbers from 1 to 10. The code is given below -

```
i=1
while i<=10:
    print(i, end=' ')
    i+=1
```

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
1 2 3 4 5 6 7 8 9 10
```

Loop Control Statements

Continue Statement
It returns the control of the Python interpreter to the beginning of the loop.
Code

```
# Python program to show how to use continue loop control

# Initiating the loop
for string in "While Loops":
    if string == "o" or string == "i" or string == "e":
        continue
    print('Current Letter:', string)
```

Output:

```
Current Letter: W
Current Letter: h
Current Letter: l
Current Letter:
Current Letter: L
Current Letter: p
Current Letter: s
```

Break Statement

It stops the execution of the loop when the break statement is reached.

Code

```
# Python program to show how to use the break statement

# Initiating the loop
for string in "Python Loops":
    if string == 'n':
        break
    print('Current Letter: ', string)
```

Output:

output is given below -

```
Current Letter: P Current Letter: y Current Letter: t Current Letter: h Current Letter: o
```

Pass Statement

Pass statements are used to create empty loops. Pass statement is also employed for classes, functions, and empty control statements.

Code

```
# Python program to show how to use the pass statement
for a string in "Python Loops":
    pass
print( 'The Last Letter of given string is:', string)
```

Output:

```
The Last Letter of given string is: s
```

Python String

Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes. The computer does not understand the characters; internally, it stores manipulated character as the combination of the 0's and 1's.

Each character is encoded in the ASCII or Unicode character. So we can say that Python strings are also called the collection of Unicode characters.

In Python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

Consider the following example in Python to create a string.
Syntax:

```
str = "Hi Python !"
```

Here, if we check the type of the variable **str** using a Python script **print**(type(str)), then it will **print** a string (str).

In Python, strings are treated as the sequence of characters, which means that Python doesn't support the character data-type; instead, a single character written as 'p' is treated as the string of length 1.

Example

Creating String in Python

We can create a string by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string or **docstrings**.

```
#Using single quotes
str1 = 'Hello Python'
print(str1)
#Using double quotes
str2 = "Hello Python"
print(str2)

#Using triple quotes
str3 = '''''Triple quotes are generally used for
   represent the multiline or
   docstring'''
print(str3)
```

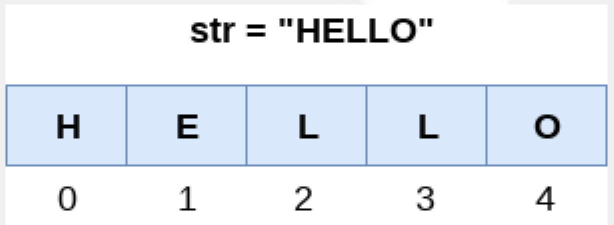
Example

Output:

```
Hello Python
Hello Python
Triple quotes are generally used for
represent the multiline or
 docstring
```

Strings indexing and splitting

Like other languages, the indexing of the Python strings starts from 0. For example, The string "HELLO" is indexed as given in the below figure.



```
str[0] = 'H'

str[1] = 'E'

str[2] = 'L'

str[3] = 'L'

str[4] = 'O'
```

Consider the following example:

```
str = "HELLO"
print(str[0])
print(str[1])
print(str[2])
print(str[3])
print(str[4])
# It returns the IndexError because 6th index doesn't exist
print(str[6])
```

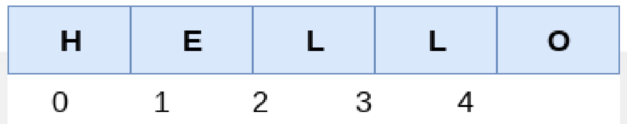
Output:

```
H E L L O IndexError: string index out of range
```

As shown in Python, the slice operator [] is used to access the individual characters of the string. However, we can use the : (colon) operator in Python to access the substring from the given string. Consider the following example.

Example

str = "HELLO"



str[0] = 'H'	str[:] = 'HELLO'
str[1] = 'E'	str[0:] = 'HELLO'
str[2] = 'L'	str[:5] = 'HELLO'
str[3] = 'L'	str[:3] = 'HEL'
str[4] = 'O'	str[0:2] = 'HE'
	str[1:4] = 'ELL'

String Operators

Operator	Description
+	It is known as concatenation operator used to join the strings given either side of the operator.
*	It is known as repetition operator. It concatenates the multiple copies of the same string.
[]	It is known as slice operator. It is used to access the sub-strings of a particular string.
[:]	It is known as range slice operator. It is used to access the characters from the specified range.
in	It is known as membership operator. It returns if a particular sub-string is present in the specified string.
not in	It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string.
r/R	It is used to specify the raw string. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string.
%	It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python.

Example

```
str = "Hello"
str1 = " world"
print(str*3) # prints HelloHelloHello
print(str+str1)# prints Hello world
print(str[4]) # prints o
print(str[2:4]); # prints ll
print('w' in str) # prints false as w is not present in str
print('wo' not in str1) # prints false as wo is present in str1.
print(r'C://python37') # prints C://python37 as it is written
print("The string str : %s"%(str)) # prints The string str : Hello
```

Output:

```
HelloHelloHello
Hello world
O
 ll False False C://python37 The string str : Hello
```

Python String functions

Method	Description
capitalize()	It capitalizes the first character of the String. This function is deprecated in python3
casefold()	It returns a version of s suitable for case-less comparisons.
center(width ,fillchar)	It returns a space padded string with the original string centred with equal number of left and right spaces.
count(string,begin,end)	It counts the number of occurrences of a substring in a String between begin and end index.
decode(encoding = 'UTF8', errors = 'strict')	Decodes the string using codec registered for encoding.
encode()	Encode S using the codec registered for encoding. Default encoding is 'utf-8'.
endswith(suffix ,begin=0,end=len(string))	It returns a Boolean value if the string terminates with given suffix between begin and end.
expandtabs(tabsize = 8)	It defines tabs in string to multiple spaces. The default space value is 8.
find(substring ,beginIndex, endIndex)	It returns the index value of the string where substring is found between begin index and end index.
format(value)	It returns a formatted version of S, using the passed value.
index(subsring, beginIndex, endIndex)	It throws an exception if string is not found. It works same as find() method.
isalnum()	It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise, it returns false.
isalpha()	It returns true if all the characters are alphabets and there is at least one character, otherwise False.
isdecimal()	It returns true if all the characters of the string are decimals.
isdigit()	It returns true if all the characters are digits and there is at least one character, otherwise False.
isidentifier()	It returns true if the string is the valid identifier.
islower()	It returns true if the characters of a string are in lower case, otherwise false.
isnumeric()	It returns true if the string contains only numeric characters.

Python List

In Python, the sequence of various data types is stored in a list. A list is a collection of different kinds of values or items. Since Python lists are mutable, we can change their elements after forming. The comma (,) and the square brackets [enclose the List's items] serve as separators.
Although six Python data types can hold sequences, the List is the most common and reliable form. A list, a type of sequence data, is used to store the collection of data.
A list is a collection of items separated by commas and denoted by the symbol [].

Example

```
# a simple list
list1 = [1, 2, "Python", "Program", 15.9]
list2 = ["Amy", "Ryan", "Henry", "Emma"]
 # printing the list
print(list1)
print(list2)
# printing the type of list
print(type(list1))
print(type(list2))
```

Output:

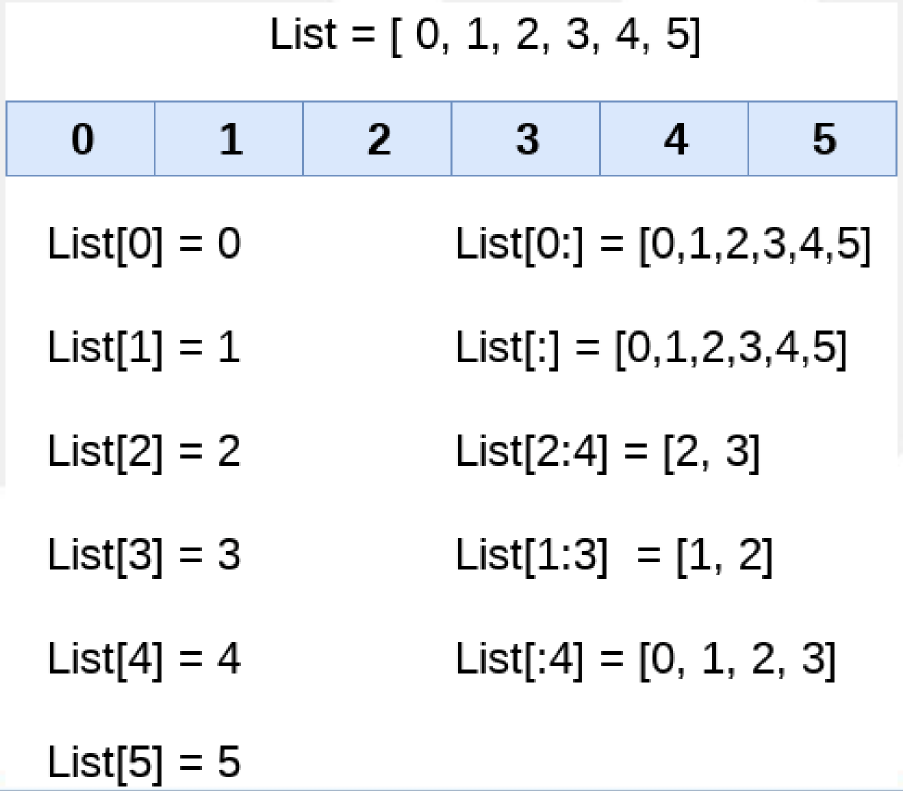
```
[1, 2, 'Python', 'Program', 15.9] ['Amy', 'Ryan', 'Henry', 'Emma'] < class ' list ' > < class ' list ' >
```

Characteristics of Lists

The characteristics of the List are as follows:
The lists are in order.
The list element can be accessed via the index.
The mutable type of List is
The rundowns are changeable sorts.
The number of various elements can be stored in a list.

List Indexing and Splitting

The indexing procedure is carried out similarly to string processing. The slice operator [] can be used to get to the List's components. The index ranges from 0 to length -1. The 0th index is where the List's first element is stored; the 1st index is where the second element is stored, and so on.



List = [0, 1, 2, 3, 4, 5]

0	1	2	3	4	5

List[0] = 0	List[0:] = [0,1,2,3,4,5]
List[1] = 1	List[:] = [0,1,2,3,4,5]
List[2] = 2	List[2:4] = [2, 3]
List[3] = 3	List[1:3] = [1, 2]
List[4] = 4	List[:4] = [0, 1, 2, 3]
List[5] = 5	

List Indexing and Splitting

We can get the sub-list of the list using the following syntax.
list_varible(start:stop:step)
The beginning indicates the beginning record position of the rundown.
The stop signifies the last record position of the rundown.
Within a start, the step is used to skip the nth element: stop.
The start parameter is the initial index, the step is the ending index, and the value of the end parameter is the number of elements that are "stepped" through. The default value for the step is one without a specific value. Inside the resultant Sub List, the same with record start would be available, yet the one with the file finish will not. The first element in a list appears to have an index of zero.

Python List Operations

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

Repetition

Concatenation

Length

Iteration

Membership

1. Repetition

The redundancy administrator empowers the rundown components to be rehashed on different occasions.

Code

```
# repetition of list
# declaring the list
list1 = [12, 14, 16, 18, 20]
# repetition operator *
l = list1 * 2
print(l)
```

Output:

```
[12, 14, 16, 18, 20, 12, 14, 16, 18, 20]
```

2. Concatenation

It concatenates the list mentioned on either side of the operator.

Code

```
# concatenation of two lists
# declaring the lists
list1 = [12, 14, 16, 18, 20]
list2 = [9, 10, 32, 54, 86]
# concatenation operator +
l = list1 + list2
print(l)
```

Output:

```
[12, 14, 16, 18, 20, 9, 10, 32, 54, 86]
```

3. Length

It is used to get the length of the list

Code

```
# size of the list
# declaring the list
list1 = [12, 14, 16, 18, 20, 23, 27, 39, 40]
# finding length of the list
len(list1)
```

Output:

```
9
```

4. Iteration

The for loop is used to iterate over the list elements.

Code

```
# iteration of the list
# declaring the list
list1 = [12, 14, 16, 39, 40]
# iterating
for i in list1:
   print(i)
```

Output:

```
12 14 16 39 40
```

5. Membership

It returns true if a particular item exists in a particular list otherwise false.

Code

```
# membership of the list
# declaring the list
list1 = [100, 200, 300, 400, 500]
# true will be printed if value exists
# and false if not
print(600 in list1)
print(700 in list1)
print(1040 in list1)
print(300 in list1)
print(100 in list1)
print(500 in list1)
```

Output:

```
False False False True True True
```

Iterating a List

A list can be iterated by using a for - in loop. A simple list containing four strings, which can be iterated as follows.

Code

```
# iterating a list
list = ["John", "David", "James", "Jonathan"]
for i in list:
    # The i variable will iterate over the elements of the List and contains each element in each iteration.

    print(i)
```

Output:

```
John David James Jonathan
```

Adding Elements to the List

The append() function in Python can add a new item to the List. In any case, the annex() capability can enhance the finish of the rundown.
Consider the accompanying model, where we take the components of the rundown from the client and print the rundown on the control center.

Code

```
#Declaring the empty list
l =[]
#Number of elements will be entered by the user
n = int(input("Enter the number of elements in the list:"))
# for loop to take the input
for i in range(0,n):
    # The input is taken from the user and added to the list as the item
    l.append(input("Enter the item:"))
print("printing the list items..")
# traversal loop to print the list items
for i in l:
    print(i, end = "  ")
```

Example

```
Enter the number of elements in the list:10
Enter the item:32
 Enter the item:56
Enter the item:81
Enter the item:2
Enter the item:34
Enter the item:65
 Enter the item:09
Enter the item:66
 Enter the item:12
Enter the item:18
printing the list items..
32 56 81 2 34 65 09 66 12 18
```

Removing Elements from the List

The remove() function in Python can remove an element from the List. To comprehend this idea, look at the example that follows.

Code

```
list = [0,1,2,3,4]
print("printing original list: ");
for i in list:
   print(i,end=" ")
list.remove(2)
print("\nprinting the list after the removal of first element...")
for i in list:
   print(i,end=" ")
```

Output:

```
printing original list: 0 1 2 3 4 printing the list after the removal of first element... 0 1 3 4
```

Python List Built-in Functions

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

len()

max()

min()

len( )

It is used to calculate the length of the list.

Code

```
# size of the list
# declaring the list
list1 = [12, 16, 18, 20, 39, 40]
# finding length of the list
len(list1)
```

Output:

```
6
```

Python List Built-in Functions

Max( )

It returns the maximum element of the list

Code

```
# maximum of the list
list1 = [103, 675, 321, 782, 200]
# large element in the list
print(max(list1))
```

Output:

```
782
```

Python List Built-in Functions

Min( )

It returns the minimum element of the list

Code

```
# minimum of the list
list1 = [103, 675, 321, 782, 200]
# smallest element in the list
print(min(list1))
```

Output:

```
103
```

Python Tuples

A comma-separated group of items is called a Python triple. The ordering, settled items, and reiterations of a tuple are to some degree like those of a rundown, but in contrast to a rundown, a tuple is unchanging.

The main difference between the two is that we cannot alter the components of a tuple once they have been assigned. On the other hand, we can edit the contents of a list.

Example

("Suzuki", "Audi", "BMW"," Skoda ") is a tuple.

Features of Python Tuple

Tuples are an immutable data type, meaning their elements cannot be changed after they are generated.

Each element in a tuple has a specific order that will never change because tuples are ordered sequences.

All the objects-also known as "elements"-must be separated by a comma, enclosed in parenthesis (). Although parentheses are not required, they are recommended.

Example

Code

```
# Python program to show how to create a tuple
# Creating an empty tuple
empty_tuple = ()
print("Empty tuple: ", empty_tuple)

# Creating tuple having integers
int_tuple = (4, 6, 8, 10, 12, 14)
print("Tuple with integers: ", int_tuple)

# Creating a tuple having objects of different data types
mixed_tuple = (4, "Python", 9.3)
print("Tuple with different data types: ", mixed_tuple)

# Creating a nested tuple
nested_tuple = ("Python", {4: 5, 6: 2, 8:2}, (5, 3, 5, 6))
print("A nested tuple: ", nested_tuple)
```

Example

Output:

```
Empty tuple: ()
Tuple with integers: (4, 6, 8, 10, 12, 14)
Tuple with different data types: (4, 'Python', 9.3)
A nested tuple: ('Python', {4: 5, 6: 2, 8: 2}, (5, 3, 5, 6))
```

Accessing Tuple Elements

A tuple's objects can be accessed in a variety of ways.

Indexing

Indexing We can use the index operator [] to access an object in a tuple, where the index starts at 0. The indices of a tuple with five items will range from 0 to 4. An Index Error will be raised assuming we attempt to get to a list from the Tuple that is outside the scope of the tuple record. An index above four will be out of range in this scenario.

Because the index in Python must be an integer, we cannot provide an index of a floating data type or any other type. If we provide a floating index, the result will be TypeError.

The method by which elements can be accessed through nested tuples can be seen in the example below.

Example

```
# Python program to show how to access tuple elements
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Collection")
print(tuple_[0])
print(tuple_[1])
# trying to access element index more than the length of a tuple
try:
   print(tuple_[5])
except Exception as e:
   print(e)
# trying to access elements through the index of floating data type
try:
   print(tuple_[1.0])
except Exception as e:
   print(e)
# Creating a nested tuple
nested_tuple = ("Tuple", [4, 6, 2, 6], (6, 2, 6, 7))

# Accessing the index of a nested tuple
print(nested_tuple[0][3])
print(nested_tuple[1][1])
```

Output:

```
Python Tuple tuple index out of range tuple indices must be integers or slices, not float l 6
```

Negative Indexing

Python's sequence objects support negative indexing.

The last thing of the assortment is addressed by - 1, the second last thing by - 2, etc.

Code

```
# Python program to show how negative indexing works in Python tuples
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Collection")
# Printing elements using negative indices
print("Element at -1 index: ", tuple_[-1])
print("Elements between -4 and -1 are: ", tuple_[-4:-1])
```

Output:

```
Element at -1 index: Collection Elements between -4 and -1 are: ('Python', 'Tuple', 'Ordered')
```

Slicing

Tuple slicing is a common practice in Python and the most common way for programmers to deal with practical issues. Look at a tuple in Python. Slice a tuple to access a variety of its elements. Using the colon as a straightforward slicing operator (:) is one strategy.
To gain access to various tuple elements, we can use the slicing operator colon (:).

Code

```
# Python program to show how slicing works in Python tuples
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")
# Using slicing to access elements of the tuple
print("Elements between indices 1 and 3: ", tuple_[1:3])
# Using negative indexing in slicing
print("Elements between indices 0 and -4: ", tuple_[:-4])
# Printing the entire tuple by using the default start and end values.
print("Entire tuple: ", tuple_[:])
```

Output:

```
Elements between indices 1 and 3: ('Tuple', 'Ordered') Elements between indices 0 and -4: ('Python', 'Tuple')
Entire tuple: ('Python', 'Tuple', 'Ordered', 'Immutable', 'Collection', 'Objects')
```

Deleting a Tuple

A tuple's parts can't be modified, as was recently said. We are unable to eliminate or remove tuple components as a result. However, the keyword del can completely delete a tuple.

Code

```
# Python program to show how to delete elements of a Python tuple
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")
# Deleting a particular element of the tuple
try:
   del tuple_[3]
   print(tuple_)
except Exception as e:
   print(e)
# Deleting the variable from the global space of the program
del tuple_
# Trying accessing the tuple after deleting it
try:
   print(tuple_)
except Exception as e:
   print(e)
```

Output:

```
'tuple' object does not support item deletion name 'tuple_' is not defined
```

Repetition Tuples in Python

```
# Python program to show repetition in tuples
tuple_ = ('Python',"Tuples")
print("Original tuple is: ", tuple_)
# Repeting the tuple elements
tuple_ = tuple_ * 3
print("New tuple is: ", tuple_)
```

Output:

```
Original tuple is: ('Python', 'Tuples')
New tuple is: ('Python', 'Tuples', 'Python', 'Tuples', 'Python', 'Tuples')
```

Tuple Methods

Like the list, Python Tuples is a collection of immutable objects. There are a few ways to work with tuples in Python. With some examples, this essay will go over these two approaches in detail.
The following are some examples of these methods.

Count () Method

The times the predetermined component happens in the Tuple is returned by the count () capability of the Tuple.

Code

```
# Creating tuples
T1 = (0, 1, 5, 6, 7, 2, 2, 4, 2, 3, 2, 3, 1, 3, 2)
T2 = ('python', 'java', 'python', 'Tpoint', 'python', 'java')
# counting the appearance of 3
res = T1.count(2)
print('Count of 2 in T1 is:', res)
# counting the appearance of java
res = T2.count('java')
print('Count of Java in T2 is:', res)
```

Output:

```
Count of 2 in T1 is: 5 Count of java in T2 is: 2
```

Index() Method:

The Index() function returns the first instance of the requested element from the Tuple.

Parameters:

The thing that must be looked for.

Start: (Optional) the index that is used to begin the final (optional) search: The most recent index from which the search is carried out

Index Method

Code

```
# Creating tuples
Tuple_data = (0, 1, 2, 3, 2, 3, 1, 3, 2)
# getting the index of 3
res = Tuple_data.index(3)
print('First occurrence of 1 is', res)
# getting the index of 3 after 4th
# index
res = Tuple_data.index(3, 4)
print('First occurrence of 1 after 4th index is:', res)
```

Output:

```
First occurrence of 1 is 3 First occurrence of 1 after 4th index is: 6
```

Tuples have the following advantages over lists:

Triples take less time than lists do.
Due to tuples, the code is protected from accidental modifications. It is desirable to store non-changing information in "tuples" instead of "records" if a program expects it.
A tuple can be used as a dictionary key if it contains immutable values like strings, numbers, or another tuple. "Lists" cannot be utilized as dictionary keys because they are mutable.

Python Set

A Python set is the collection of the unordered items. Each element in the set must be unique, immutable, and the sets remove the duplicate elements. Sets are mutable which means we can modify it after its creation.

Unlike other collections in Python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index. However, we can print them all together, or we can get the list of elements by looping through the set.

Creating a set

The set can be created by enclosing the comma-separated immutable items with the curly braces {}. Python also provides the set() method, which can be used to create the set by the passed sequence.

Example 1: Using curly braces

```
Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
print(Days)
print(type(Days))
print("looping through the set elements ... ")
for i in Days:
    print(i)
```

Output:

```
{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'} <class 'set'> looping
through the set elements ... Friday Tuesday Monday Saturday Thursday Sunday Wednesday
```

Example 2: Using set() method

```
Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
print(Days)
print(type(Days))
print("looping through the set elements ... ")
for i in Days:
    print(i)
```

Output:

```
{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'} <class 'set'> looping
through the set elements ... Friday Wednesday Thursday Saturday Monday Tuesday Sunday
```

Adding items to the set

Python provides the **add()** method and **update()** method which can be used to add some particular item to the set. The add() method is used to add a single element whereas the update() method is used to add multiple elements to the set. Consider the following example.

Example: 1 - Using add() method

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(months)
print("\nAdding other months to the set...");
Months.add("July");
Months.add ("August");
print("\nPrinting the modified set...");
print(Months)
print("\nlooping through the set elements ... ")
for i in Months:
   print(i)
```

Output:

```
printing the original set ... {'February', 'May', 'April', 'March', 'June', 'January'} Adding other months to the set... Printing the modified set...
{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'} looping through the set elements ... February July May April March August
June January
```

Example - 2 Using update() function

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nupdating the original set ... ")
Months.update(["July","August","September","October"]);
print("\nprinting the modified set ... ")
print(Months);
```

Output:

```
printing the original set ... {'January', 'February', 'April', 'May', 'June', 'March'} updating the original set ... printing the modified set ... {'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July', 'September', 'March'}
```

Removing items from the set

Python provides the **discard()** method and **remove()** method which can be used to remove the items from the set. The difference between these function, using discard() function if the item does not exist in the set then the set remain unchanged whereas remove() method will through an error.

Example-1 Using discard() method

```
months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(months)
print("\nRemoving some months from the set...");
months.discard("January");
months.discard("May");
print("\nPrinting the modified set...");
print(months)
print("\nlooping through the set elements ... ")
for i in months:
   print(i)
```

Output:

```
printing the original set ... {'February', 'January', 'March', 'April', 'June', 'May'} Removing some months from the set... Printing
the modified set... {'February', 'March', 'April', 'June'} looping through the set elements ... February March April June
```

Example-2 Using remove() function

```
months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(months)
print("\nRemoving some months from the set...");
months.remove("January");
months.remove("May");
print("\nPrinting the modified set...");
print(months)
```

Output:

```
printing the original set ... {'February', 'June', 'April', 'May', 'January', 'March'} Removing some
months from the set... Printing the modified set... {'February', 'June', 'April', 'March'}
```

Difference between discard() and remove()

Despite the fact that **discard()** and **remove()** method both perform the same task, There is one main difference between discard() and remove().

If the key to be deleted from the set using discard() doesn't exist in the set, the Python will not give the error. The program maintains its control flow.

On the other hand, if the item to be deleted from the set using remove() doesn't exist in the set, the Python will raise an error.

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving items through discard() method...");
Months.discard("Feb"); #will not give an error although the key feb is not available in the set
print("\nprinting the modified set...")
print(Months)
print("\nRemoving items through remove() method...");
Months.remove("Jan") #will give an error as the key jan is not available in the set.
print("\nPrinting the modified set...")
print(Months)
```

```
printing the original set ... {'March', 'January', 'April', 'June', 'February', 'May'} Removing items through discard() method...
printing the modified set... {'March', 'January', 'April', 'June', 'February', 'May'} Removing items through remove() method...
Traceback (most recent call last): File "set.py", line 9, in Months.remove("Jan") KeyError: 'Jan'
```

FrozenSets

In Python, a frozen set is an immutable version of the built-in set data type. It is similar to a set, but its contents cannot be changed once a frozen set is created.
Frozen set objects are unordered collections of unique elements, just like sets. They can be used the same way as sets, except they cannot be modified. Because they are immutable, frozen set objects can be used as elements of other sets or dictionary keys, while standard sets cannot.
One of the main advantages of using frozen set objects is that they are hashable, meaning they can be used as keys in dictionaries or as elements of other sets. Their contents cannot change, so their hash values remain constant. Standard sets are not hashable because they can be modified, so their hash values can change.

Example

Consider the following example to create the frozen set.

```
Frozenset = frozenset([1,2,3,4,5])
print(type(Frozenset))
print("\nprinting the content of frozen set...")
for i in Frozenset:
   print(i);
Frozenset.add(6) #gives an error since we cannot change the content of Frozenset after creation
```

Output:

```
<class 'frozenset'> printing the content of frozen set... 1 2 3 4 5 Traceback (most recent call last): File
"set.py", line 6, in <module> Frozenset.add(6) #gives an error since we can change the content of
Frozenset after creation AttributeError: 'frozenset' object has no attribute 'add'
```

Python Dictionary

Dictionaries are a useful data structure for storing data in Python because they are capable of imitating real-world data arrangements where a certain value exists for a given key.
The data is stored as key-value pairs using a Python dictionary.
This data structure is mutable
The components of dictionary were made using keys and values.
Keys must only have one component.
Values can be of any type, including integer, list, and tuple.
A dictionary is, in other words, a group of key-value pairs, where the values can be any Python object. The keys, in contrast, are immutable Python objects, such as strings, tuples, or numbers.
Dictionary entries are ordered as of Python version 3.7. In Python 3.6 and before, dictionaries are generally unordered.

Creating the Dictionary

Curly brackets are the simplest way to generate a Python dictionary, although there are other approaches as well. With many key-value pairs surrounded in curly brackets and a colon separating each key from its value, the dictionary can be built. (:). The following provides the syntax for defining the dictionary.

Syntax:

Dict = {"Name": "Gayle", "Age": 25}

In the above dictionary **Dict**, The keys **Name** and **Age** are the strings which comes under the category of an immutable object.

Let's see an example to create a dictionary and print its content.

Example

Code

```
Employee = {"Name": "Johnny", "Age": 32, "salary":26000,"Company":"^TCS"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
```

Output

```
<class 'dict'> printing Employee data .... {'Name': 'Johnny', 'Age': 32, 'salary': 26000, 'Company': TCS}
```

Python provides the built-in function **dict()** method which is also used to create the dictionary.

Example

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

# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Hcl', 2: 'WIPRO', 3:'Facebook'})
print("\nCreate Dictionary by using  dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict = dict([(4, 'Rinku'), (2, Singh)])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output

```
Empty Dictionary: {} Create Dictionary by using dict(): {1: 'Hcl', 2: 'WIPRO', 3: 'Facebook'} Dictionary with each item as a pair:
{4: 'Rinku', 2: 'Singh'}
```

Accessing the dictionary values

To access data contained in lists and tuples, indexing has been studied. The keys of the dictionary can be used to obtain the values because they are unique from one another. The following method can be used to access dictionary values.

Code

```
Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"}
print(type(Employee))
print("printing Employee data .... ")
print("Name : %s" %Employee["Name"])
print("Age : %d" %Employee["Age"])
print("Salary : %d" %Employee["salary"])
print("Company : %s" %Employee["Company"])
```

Output

```
ee["Company"]) Output <class 'dict'> printing Employee data .... Name : Dev Age : 20 Salary : 45000 Company :
WIPRO
```

Python provides us with an alternative to use the get() method to access the dictionary values. It would give the same result as given by the indexing.

Adding Dictionary Values

The dictionary is a mutable data type, and utilising the right keys allows you to change its values. Dict[key] = value and the value can both be modified. An existing value can also be updated using the update() method

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)
# Adding elements to dictionary one at a time
Dict[0] = 'Peter'
Dict[2] = 'Joseph'
Dict[3] = 'Ricky'
print("\nDictionary after adding 3 elements: ")
print(Dict)
# Adding set of values
# with a single Key
# The Emp_ages doesn't exist to dictionary
Dict['Emp_ages'] = 20, 33, 24
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Updating existing Key's Value
Dict[3] = 'JavaTpoint'
print("\nUpdated key value: ")
print(Dict)
```

Example

Output

```
Empty Dictionary: {} Dictionary after adding 3 elements: {0: 'Peter', 2: 'Joseph', 3: 'Ricky'} Dictionary after adding 3 elements: {0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)} Updated key value: {0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp_ages': (20, 33, 24)}
```

Deleting Elements using del Keyword

The items of the dictionary can be deleted by using the **del** keyword as given below.

Code

```
Employee = {"Name": "David", "Age": 30, "salary":55000,"Company":"WIPRO"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Deleting some of the employee data")
del Employee["Name"]
del Employee["Company"]
print("printing the modified information ")
print(Employee)
print("Deleting the dictionary: Employee");
del Employee
print("Lets try to print it again ");
print(Employee)
```

Cont..

Output

<class 'dict'> printing Employee data .... {'Name': 'David', 'Age': 30, 'salary': 55000, 'Company': 'WIPRO'} Deleting some of the employee data printing the modified information {'Age': 30, 'salary': 55000} Deleting the dictionary: Employee Lets try to print it again NameError: name 'Employee' is not defined.

Properties of Dictionary Keys

1. In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key. Consider the following example.

Code

```
Employee={"Name":"John","Age":29,"Salary":25000,"Company":"WIPRO","Name":
"John"}
for x,y in Employee.items():
        print(x,y)
```

Output

```
Name John Age 29 Salary 25000 Company WIPRO
```

Parul® University

DIGITAL LEARNING CONTENT

Built-in Dictionary Functions

A function is a method that can be used on a construct to yield a value. Additionally, the construct is unaltered. A few of the Python methods can be combined with a Python dictionary.
The built-in Python dictionary methods are listed below, along with a brief description.

len()

The dictionary's length is returned via the len() function in Python. The string is lengthened by one for each key-value pair.

Code

```
dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
len(dict)
```

Output

```
4
```

Cont..

any()

Like how it does with lists and tuples, the any() method returns True indeed if one dictionary key does have a Boolean expression that evaluates to True.

Code

```
dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
any({'':'','':'','3':''})
```

Output

```
True
```

all()

Unlike in any() method, all() only returns True if each of the dictionary's keys contain a True Boolean value.

Code

```
dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
all({1:'',2:'','':''})
```

Output: false

Cont..

sorted()

Like it does with lists and tuples, the sorted() method returns an ordered series of the dictionary's keys. The ascending sorting has no effect on the original Python dictionary.

Code

```
dict = {7: "Ayan", 5: "Bunny", 8: "Ram", 1: "Bheem"}
sorted(dict)
```

Output

```
[ 1, 5, 7, 8]
```

Built-in Dictionary methods

The built-in python dictionary methods along with the description and Code are given below.

clear()

It is mainly used to delete all the items of the dictionary.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# clear() method
dict.clear()
print(dict)
```

Output

```
{ }
```

Built-in Dictionary methods

copy()

It returns a shallow copy of the dictionary which is created.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# copy() method
dict_demo = dict.copy()
print(dict_demo)
```

Output

```
{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}
```

Built-in Dictionary methods

pop()

It mainly eliminates the element using the defined key.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# pop() method
dict_demo = dict.copy()
x = dict_demo.pop(1)
print(x)
```

Output

```
{2: 'WIPRO', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}
```

Built-in Dictionary methods

popitem()

removes the most recent key-value pair entered

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# popitem() method
dict_demo.popitem()
print(dict_demo)
```

Output

```
{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook'}
```

Built-in Dictionary methods

keys()

It returns all the keys of the dictionary.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# keys() method
print(dict_demo.keys())
```

Output

```
dict_keys([1, 2, 3, 4, 5])
```

Built-in Dictionary methods

tems()

It returns all the key-value pairs as a tuple.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# items() method
print(dict_demo.items())
```

Output

```
dict_items([(1, 'Hcl'), (2, 'WIPRO'), (3, 'Facebook'), (4, 'Amazon'), (5, 'Flipkart')])
```

Built-in Dictionary methods

get()

It is used to get the value specified for the passed key.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# get() method
print(dict_demo.get(3))
```

Output

```
Facebook
```

Built-in Dictionary methods

update()

It mainly updates all the dictionary by adding the key-value pair of dict2 to this dictionary.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# update() method
dict_demo.update({3: "TCS"})
print(dict_demo)
```

Output

```
{1: 'Hcl', 2: 'WIPRO', 3: 'TCS'}
```

Built-in Dictionary methods

values()

It returns all the values of the dictionary with respect to given input.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# values() method
print(dict_demo.values())
```

Output

```
dict_values(['Hcl', 'WIPRO', 'TCS'])
```