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:

```
1. 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.

Code

```
1. Employee = {"Name": "Johnny", "Age": 32, "salary":26000,"Company":"^TCS"}
2. 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.

The empty curly braces {} is used to create empty dictionary.

Code

```
# 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

```
1. Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print("Name : %s" %Employee["Name"])
5. print("Age : %d" %Employee["Age"])
6. print("Salary : %d" %Employee["salary"])
7. 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.

> Note: The value is updated if the key-value pair is already present in the dictionary. Otherwise, the dictionary's newly added keys.

Let's see an example to update the dictionary values.

Example - 1:

Code

```
# 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)
```

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)}
```

Example - 2:

Code

```
1. Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print(Employee)
5. print("Enter the details of the new employee....");
6. Employee["Name"] = input("Name: ");
7. Employee["Age"] = int(input("Age: "));
8. Employee["salary"] = int(input("Salary: "));
9. Employee["Company"] = input("Company:");
10. print("printing the new data");
11. print(Employee)
```

Output

```
<class 'dict'>
printing Employee data ....
Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"} Enter
the details of the new employee....
Name: Sunny
Age: 38
Salary: 39000
Company:Hcl
printing the new data
{'Name': 'Sunny', 'Age': 38, 'salary': 39000, 'Company': 'Hcl'}
```

Deleting Elements using del Keyword

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

Code

```
1. Employee = {"Name": "David", "Age": 30, "salary":55000,"Company":"WIPRO"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. 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)
```

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.
```

The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.

Deleting Elements using pop() Method

A dictionary is a group of key-value pairs in Python. You can retrieve, insert, and remove items using this unordered, mutable data type by using their keys. The pop() method is one of the ways to get rid of elements from a dictionary. In this post, we'll talk about how to remove items from a Python dictionary using the pop() method.

The value connected to a specific key in a dictionary is removed using the pop() method, which then returns the value. The key of the element to be removed is the only argument needed. The pop() method can be used in the following ways:

Code

```
# Creating a Dictionary
Dict1 = {1: 'JavaTpoint', 2: 'Educational', 3: 'Website'}
# Deleting a key
```

```
# using pop() method
pop_key = Dict1.pop(2)
print(Dict1)
```

Output

```
{1: 'JavaTpoint', 3: 'Website'}
```

Additionally, Python offers built-in functions popitem() and clear() for removing dictionary items. In contrast to the clear() method, which removes all of the elements from the entire dictionary, popitem() removes any element from a dictionary.

Iterating Dictionary

A dictionary can be iterated using for loop as given below.

Example 1

Code

```
# for loop to print all the keys of a dictionary
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
for x in Employee:
    print(x)
```

Output

```
Name
Age
salary
Company
```

Example 2

Code

```
#for loop to print all the values of the dictionary
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"} for x in Em
ployee:
 print(Employee[x])
```

Output

```
John
29
25000
WIPRO
```

Example - 3

Code

```
1. #for loop to print the values of the dictionary by using values() method.
2. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
3. for x in Employee.values():
4.     print(x)
```

Output

```
John
29
25000
WIPRO
```

Example 4

Code

```
1. #for loop to print the items of the dictionary by using items() method
2. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
3. for x in Employee.items():
4.     print(x)
```

Output

```
 ('Name', 'John')
('Age', 29)
('salary', 25000)
('Company', 'WIPRO')
```

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
```

2. The key cannot belong to any mutable object in Python. Numbers, strings, or tuples can be used as the key, however mutable objects like lists cannot be used as the key in a dictionary.

Consider the following example.

Code

```
Employee = {"Name": "John", "Age": 29, "salary":26000,"Company":"WIPRO",[100,201,30
1]:"Department ID"}
for x,y in Employee.items():
    print(x,y)
```

Output

```
Traceback (most recent call last):
  File "dictionary.py", line 1, in
    Employee        =        {"Name":        "John",        "Age":        29,
"salary":26000,"Company":"WIPRO",[100,201,301]:"Department ID"}
TypeError: unhashable type: 'list'
```

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

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

Output

```
4
```

- **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

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

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

Output

```
False
```

- **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

```
1. dict = {7: "Ayan", 5: "Bunny", 8: "Ram", 1: "Bheem"}
2. 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

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

Output

```
{ }
```

- **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'}
```

- **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'}
```

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'}
```

- **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])
```

- **items()**

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')])
```

- **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
```

- **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'}
```

- **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'])
```

Python Functions

This tutorial will go over the fundamentals of Python functions, including what they are, their syntax, their primary parts, return keywords, and significant types. We'll also look at some examples of Python function definitions.

What are Python Functions?

A collection of related assertions that carry out a mathematical, analytical, or evaluative operation is known as a function. An assortment of proclamations called Python Capabilities returns the specific errand. Python functions are necessary for intermediate-level programming and are easy to define. Function names meet the same standards as variable names do. The objective is to define a function and group-specific frequently performed actions. Instead of repeatedly creating the same code block for various input variables, we can call the function and reuse the code it contains with different variables.

Client-characterized and worked-in capabilities are the two primary classes of capabilities in Python. It aids in maintaining the program's uniqueness, conciseness, and structure.

Advantages of Python Functions

Pause We can stop a program from repeatedly using the same code block by including functions.

- Once defined, Python functions can be called multiple times and from any location in a program.
- Our Python program can be broken up into numerous, easy-to-follow functions if it is significant.
- The ability to return as many outputs as we want using a variety of arguments is one of Python's most significant achievements.
- However, Python programs have always incurred overhead when calling functions.

However, calling functions has always been overhead in a Python program.

Syntax

```
# An example Python Function
def function_name( parameters ):
```

```
3.      # code block
```

The accompanying components make up to characterize a capability, as seen previously.

- The start of a capability header is shown by a catchphrase called def.
- function_name is the function's name, which we can use to distinguish it from other functions. We will utilize this name to call the capability later in the program. Name functions in Python must adhere to the same guidelines as naming variables.
- Using parameters, we provide the defined function with arguments. Notwithstanding, they are discretionary.
- A colon (:) marks the function header's end.
- We can utilize a documentation string called docstring in the short structure to make sense of the reason for the capability.
- Several valid Python statements make up the function's body. The entire code block's indentation depth-typically four spaces-must be the same.
- A return expression can get a value from a defined function.

Illustration of a User-Defined Function

We will define a function that returns the argument number's square when called.

```
1. # Example Python Code for User-Defined function
2. def square( num ):
3.     """
4.     This function computes the square of the number.
5.     """
6.     return num**2
7. object_ = square(6)
8. print( "The square of the given number is: ", object_ )
```

Output:

```
The square of the given number is:  36
```

Calling a Function

Calling a Function To define a function, use the def keyword to give it a name, specify the arguments it must receive, and organize the code block.

When the fundamental framework for a function is finished, we can call it from anywhere in the program. An illustration of how to use the a_function function can be found below.

```
# Example Python Code for calling a function
# Defining a function
def a_function( string ):
    "This prints the value of length of string"
    return len(string)

# Calling the function we defined
print( "Length of the string Functions is: ", a_function( "Functions" ) )
print( "Length of the string Python is: ", a_function( "Python" ) )
```

Output:

```
Length of the string Functions is:  9
Length of the string Python is:  6
```

Pass by Reference vs. Pass by Value

In the Python programming language, all parameters are passed by reference. It shows that if we modify the worth of contention within a capability, the calling capability will similarly mirror the change. For instance,

Code

```
# Example Python Code for Pass by Reference vs. Value
# defining the function
def square( item_list ):
    '''''''This function will find the square of items in the list'''
    squares = [ ]
    for l in item_list:
        squares.append( l**2 )
    return squares
```

```
9.
10. # calling the defined function
11. my_list = [17, 52, 8];
12. my_result = square( my_list )
13. print( "Squares of the list are: ", my_result )
```

Output:

```
Squares of the list are:  [289, 2704, 64]
```

Function Arguments

The following are the types of arguments that we can use to call a function:

1. Default arguments
2. Keyword arguments
3. Required arguments
4. Variable-length arguments

1) Default Arguments

A default contention is a boundary that takes as information a default esteem, assuming that no worth is provided for the contention when the capability is called. The following example demonstrates default arguments.

Code

```
1. # Python code to demonstrate the use of default arguments
2. # defining a function
3. def function( n1, n2 = 20 ):
4.     print("number 1 is: ", n1)
5.     print("number 2 is: ", n2)
6.
7.
8. # Calling the function and passing only one argument
9. print( "Passing only one argument" )
10. function(30)
```

```
11.
12. # Now giving two arguments to the function
13. print( "Passing two arguments" )
14. function(50,30)
```

Output:

```
Passing only one argument
number 1 is:  30
number 2 is:  20
Passing two arguments
number 1 is:  50
number 2 is:  30
```

2) Keyword Arguments

Keyword arguments are linked to the arguments of a called function. While summoning a capability with watchword contentions, the client might tell whose boundary esteem it is by looking at the boundary name.

We can eliminate or orchestrate specific contentions in an alternate request since the Python translator will interface the furnished watchwords to connect the qualities with its boundaries. One more method for utilizing watchwords to summon the capability() strategy is as per the following:

Code

```
1. # Python code to demonstrate the use of keyword arguments
2.   # Defining a function
3. def function( n1, n2 ):
4.     print("number 1 is: ", n1)
5.     print("number 2 is: ", n2)
6.
7. # Calling function and passing arguments without using keyword
8. print( "Without using keyword" )
9. function( 50, 30)
10.
```

```
# Calling function and passing arguments using keyword
print( "With using keyword" )
function( n2 = 50, n1 = 30)
```

Output:

```
Without using keyword
number 1 is:  50
number 2 is:  30
With using keyword
number 1 is:  30
number 2 is:  50
```

3) Required Arguments

Required arguments are those supplied to a function during its call in a predetermined positional sequence. The number of arguments required in the method call must be the same as those provided in the function's definition.

We should send two contentions to the capability() all put together; it will return a language structure blunder, as seen beneath.

Code

```
# Python code to demonstrate the use of default arguments
# Defining a function
def function( n1, n2 ):
    print("number 1 is: ", n1)
    print("number 2 is: ", n2)

# Calling function and passing two arguments out of order, we need num1 to be 20 and
 num2 to be 30
print( "Passing out of order arguments" )
function( 30, 20 )

# Calling function and passing only one argument
print( "Passing only one argument" )
try:
```

```
    function( 30 )
except:
    print( "Function needs two positional arguments" )
```

Output:

```
Passing out of order arguments
number 1 is:  30
number 2 is:  20
Passing only one argument
Function needs two positional arguments
```

4) Variable-Length Arguments

We can involve unique characters in Python capabilities to pass many contentions. However, we need a capability. This can be accomplished with one of two types of characters:

"args" and "kwargs" refer to arguments not based on keywords.

To help you understand arguments of variable length, here's an example.

Code

```
# Python code to demonstrate the use of variable-length arguments
# Defining a function
def function( *args_list ):
    ans = []
    for l in args_list:
        ans.append( l.upper() )
    return ans
# Passing args arguments
object = function('Python', 'Functions', 'tutorial')
print( object )

# defining a function
def function( **kargs_list ):
```

```
    ans = []
    for key, value in kargs_list.items():
        ans.append([key, value])
    return ans
# Paasing kwargs arguments
object = function(First = "Python", Second = "Functions", Third = "Tutorial")
print(object)
```

Output:

```
['PYTHON', 'FUNCTIONS', 'TUTORIAL']
[['First', 'Python'], ['Second', 'Functions'], ['Third', 'Tutorial']]
```

return Statement

When a defined function is called, a return statement is written to exit the function and return the calculated value.

Syntax:

```
return < expression to be returned as output >
```

The return statement can be an argument, a statement, or a value, and it is provided as output when a particular job or function is finished. A declared function will return an empty string if no return statement is written.

A return statement in Python functions is depicted in the following example.

Code

```
# Python code to demonstrate the use of return statements
# Defining a function with return statement
def square( num ):
    return num**2

# Calling function and passing arguments.
print( "With return statement" )
```

```
print( square( 52 ) )

# Defining a function without return statement
def square( num ):
    num**2

# Calling function and passing arguments.
print( "Without return statement" )
print( square( 52 ) )
```

Output:

```
With return statement
2704
Without return statement
None
```

The Anonymous Functions

Since we do not use the def keyword to declare these kinds of Python functions, they are unknown. The lambda keyword can define anonymous, short, single-output functions.

Arguments can be accepted in any number by lambda expressions; However, the function only produces a single value from them. They cannot contain multiple instructions or expressions. Since lambda needs articulation, a mysterious capability can't be straightforwardly called to print.

Lambda functions can only refer to variables in their argument list and the global domain name because they contain their distinct local domain.

In contrast to inline expressions in C and C++, which pass function stack allocations at execution for efficiency reasons, lambda expressions appear to be one-line representations of functions.

Syntax

Lambda functions have exactly one line in their syntax:

```
lambda [argument1 [,argument2... .argumentn]] : expression
```

Below is an illustration of how to use the lambda function:

Code

```
# Python code to demonstrate ananymous functions
# Defining a function
lambda_ = lambda argument1, argument2: argument1 + argument2;

# Calling the function and passing values
print( "Value of the function is : ", lambda_( 20, 30 ) )
print( "Value of the function is : ", lambda_( 40, 50 ) )
```

Output:

```
Value of the function is :  50
Value of the function is :  90
```

Scope and Lifetime of Variables

A variable's scope refers to the program's domain wherever it is declared. A capability's contentions and factors are not external to the characterized capability. They only have a local domain as a result.

The length of time a variable remains in RAM is its lifespan. The lifespan of a function is the same as the lifespan of its internal variables. When we exit the function, they are taken away from us. As a result, the value of a variable in a function does not persist from previous executions.

An easy illustration of a function's scope for a variable can be found here.

Code

```
# Python code to demonstrate scope and lifetime of variables
#defining a function to print a number.
def number( ):
    num = 50
    print( "Value of num inside the function: ", num)

num = 10
```

```
8. number()
9. print( "Value of num outside the function:", num)
```

Output:

```
Value of num inside the function:  50
Value of num outside the function: 10
```

Here, we can see that the initial value of num is 10. Even though the function number() changed the value of num to 50, the value of num outside of the function remained unchanged.

This is because the capability's interior variable num is not quite the same as the outer variable (nearby to the capability). Despite having a similar variable name, they are separate factors with discrete extensions.

Factors past the capability are available inside the capability. The impact of these variables is global. We can retrieve their values within the function, but we cannot alter or change them. The value of a variable can be changed outside of the function if it is declared global with the keyword global.

Python Capability inside Another Capability

Capabilities are viewed as top-of-the-line objects in Python. First-class objects are treated the same everywhere they are used in a programming language. They can be stored in built-in data structures, used as arguments, and in conditional expressions. If a programming language treats functions like first-class objects, it is considered to implement first-class functions. Python lends its support to the concept of First-Class functions.

A function defined within another is called an "inner" or "nested" function. The parameters of the outer scope are accessible to inner functions. Internal capabilities are developed to cover them from the progressions outside the capability. Numerous designers see this interaction as an embodiment.

Code

```
1. # Python code to show how to access variables of a nested functions
2. # defining a nested function
```

```
def word():
    string = 'Python functions tutorial'
    x = 5
    def number():
        print( string )
        print( x )

    number()
word()
```

Output:

```
Python functions tutorial
5
```

Python Built-in Functions

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. There are several built-in functions in Python which are listed below:

Python abs() Function

The python **abs()** function is used to return the absolute value of a number. It takes only one argument, a number whose absolute value is to be returned. The argument can be an integer and floating-point number. If the argument is a complex number, then, abs() returns its magnitude.

Python abs() Function Example

```
#  integer number
integer = -20
print('Absolute value of -40 is:', abs(integer))

#  floating number
floating = -20.83
```

```
7. print('Absolute value of -40.83 is:', abs(floating))
```

Output:


```
Absolute value of -20 is: 20
Absolute value of -20.83 is: 20.83
```

Python all() Function

The python **all()** function accepts an iterable object (such as list, dictionary, etc.). It returns true if all items in passed iterable are true. Otherwise, it returns False. If the iterable object is empty, the all() function returns True.

Python all() Function Example

```
1. # all values true
2. k = [1, 3, 4, 6]
3. print(all(k))
4.
5. # all values false
6. k = [0, False]
7. print(all(k))
8.
9. # one false value
10. k = [1, 3, 7, 0]
11. print(all(k))
12.
13. # one true value
14. k = [0, False, 5]
15. print(all(k))
16.
17. # empty iterable
18. k = []
19. print(all(k))
```

Output:

```
True
False
False
False
True
```

Python bin() Function

The python **bin()** function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

Python bin() Function Example

```
x =  10
y =  bin(x)
print (y)
```

Output:

```
0b1010
```

Python bool()

The python **bool()** converts a value to boolean(True or False) using the standard truth testing procedure.

Python bool() Example

```
test1 = []
print(test1,'is',bool(test1))
test1 = [0]
print(test1,'is',bool(test1))
test1 = 0.0
print(test1,'is',bool(test1))
test1 = None
print(test1,'is',bool(test1))
test1 = True
print(test1,'is',bool(test1))
```

```
test1 = 'Easy string'
print(test1,'is',bool(test1))
```

Output:

```
[] is False
[0] is True
0.0 is False
None is False
True is True
Easy string is True
```


Python bytes()

The python **bytes()** in Python is used for returning a **bytes** object. It is an immutable version of the bytearray() function.

It can create empty bytes object of the specified size.

Python bytes() Example

```
string = "Hello World."
array = bytes(string, 'utf-8')
print(array)
```

Output:

```
b ' Hello World.'
```

Python callable() Function

A python **callable()** function in Python is something that can be called. This built-in function checks and returns true if the object passed appears to be callable, otherwise false.

Python callable() Function Example

```
x = 8
print(callable(x))
```

Output:


```
False
```

Python compile() Function

The python **compile()** function takes source code as input and returns a code object which can later be executed by exec() function.

Python compile() Function Example

```
# compile string source to code
code_str = 'x=5\ny=10\nprint("sum =",x+y)'
code = compile(code_str, 'sum.py', 'exec')
print(type(code))
exec(code)
exec(x)
```

Output:

```
<class 'code'>
sum = 15
```

Python exec() Function

The python **exec()** function is used for the dynamic execution of Python program which can either be a string or object code and it accepts large blocks of code, unlike the eval() function which only accepts a single expression.

Python exec() Function Example

```
x = 8
exec('print(x==8)')
exec('print(x+4)')
```

Output:

```
True
12
```

Python sum() Function

As the name says, python **sum()** function is used to get the sum of numbers of an iterable, i.e., list.

Python sum() Function Example

```
s = sum([1, 2,4 ])
print(s)

s = sum([1, 2, 4], 10)
print(s)
```


Output:

```
7
17
```

Python any() Function

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

Python any() Function Example

```
l = [4, 3, 2, 0]
print(any(l))

l = [0, False]
print(any(l))

l = [0, False, 5]
print(any(l))
```

```
9.
10. l = []
11. print(any(l))
```

Output:

```
True
False
True
False
```

Python ascii() Function

The python **ascii()** function returns a string containing a printable representation of an object and escapes the non-ASCII characters in the string using \x, \u or \U escapes.

Python ascii() Function Example

```
1. normalText = 'Python is interesting'
2. print(ascii(normalText))
3.
4. otherText = 'Pythön is interesting'
5. print(ascii(otherText))
6.
7. print('Pyth\xf6n is interesting')
```

Output:


```
'Python is interesting'
'Pyth\xf6n is interesting'
Pythön is interesting
```

Python bytearray()

The python **bytearray()** returns a bytearray object and can convert objects into bytearray objects, or create an empty bytearray object of the specified size.

Python bytearray() Example

```
string = "Python is a programming language."

# string with encoding 'utf-8'
arr = bytearray(string, 'utf-8')
print(arr)
```

Output:

```
bytearray(b'Python is a programming language.')
```

Python eval() Function

The python **eval()** function parses the expression passed to it and runs python expression(code) within the program.

Python eval() Function Example

```
x = 8
print(eval('x + 1'))
```

Output:

```
9
```

Python float()

The python **float()** function returns a floating-point number from a number or string.

Python float() Example

```
# for integers
print(float(9))

# for floats
print(float(8.19))

# for string floats
```

```
print(float("-24.27"))

# for string floats with whitespaces
print(float("     -17.19\n"))

# string float error
print(float("xyz"))
```

Output:

```
9.0
8.19
-24.27
-17.19
ValueError: could not convert string to float: 'xyz'
```

Python format() Function

The python **format()** function returns a formatted representation of the given value.

Python format() Function Example

```
# d, f and b are a type

# integer
print(format(123, "d"))

# float arguments
print(format(123.4567898, "f"))

# binary format
print(format(12, "b"))
```

Output:

```
123
123.456790
1100
```

Python frozenset()

The python **frozenset()** function returns an immutable frozenset object initialized with elements from the given iterable.

Python frozenset() Example

```
# tuple of letters
letters = ('m', 'r', 'o', 't', 's')

fSet = frozenset(letters)
print('Frozen set is:', fSet)
print('Empty frozen set is:', frozenset())
```

Output:

```
Frozen set is: frozenset({'o', 'm', 's', 'r', 't'})
Empty frozen set is: frozenset()
```

Python getattr() Function

The python **getattr()** function returns the value of a named attribute of an object. If it is not found, it returns the default value.

Python getattr() Function Example

```
class Details:
    age = 22
    name = "Phill"

details = Details()
print('The age is:', getattr(details, "age"))
print('The age is:', details.age)
```

Output:

```
The age is: 22
The age is: 22
```

Python globals() Function

The python **globals()** function returns the dictionary of the current global symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

Python globals() Function Example

```
age = 22

globals()['age'] = 22
print('The age is:', age)
```

Output:

```
The age is: 22
```

Python hasattr() Function

The python **any()** function returns true if any item in an iterable is true, otherwise it returns False.

Python hasattr() Function Example

```
l = [4, 3, 2, 0]
print(any(l))

l = [0, False]
print(any(l))

l = [0, False, 5]
print(any(l))

l = []
print(any(l))
```

Output:

```
True
False
True
False
```

Python iter() Function

The python **iter()** function is used to return an iterator object. It creates an object which can be iterated one element at a time.

Python iter() Function Example

```
# list of numbers
list = [1,2,3,4,5]

listIter = iter(list)

# prints '1'
print(next(listIter))

# prints '2'
print(next(listIter))

# prints '3'
print(next(listIter))

# prints '4'
print(next(listIter))

# prints '5'
print(next(listIter))
```

Output:

```
1
2
```

```
3
4
5
```

Python len() Function

The python **len()** function is used to return the length (the number of items) of an object.

Python len() Function Example

```
1. strA = 'Python'
2. print(len(strA))
```

Output:

```
6
```

Python list()

The python **list()** creates a list in python.

Python list() Example

```
1. # empty list
2. print(list())
3.
4. # string
5. String = 'abcde'
6. print(list(String))
7.
8. # tuple
9. Tuple = (1,2,3,4,5)
10. print(list(Tuple))
11. # list
12. List = [1,2,3,4,5]
13. print(list(List))
```

Output:

```
[]
['a', 'b', 'c', 'd', 'e']
[1,2,3,4,5]
[1,2,3,4,5]
```

Python locals() Function

The python **locals()** method updates and returns the dictionary of the current local symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

Python locals() Function Example

```
def localsAbsent():
    return locals()

def localsPresent():
    present = True
    return locals()

print('localsNotPresent:', localsAbsent())
print('localsPresent:', localsPresent())
```

Output:

```
localsAbsent: {}
localsPresent: {'present': True}
```

Python map() Function

The python **map()** function is used to return a list of results after applying a given function to each item of an iterable(list, tuple etc.).

Python map() Function Example

```
def calculateAddition(n):
  return n+n

numbers = (1, 2, 3, 4)
result = map(calculateAddition, numbers)
print(result)

# converting map object to set
numbersAddition = set(result)
print(numbersAddition)
```

Output:

```
<map object at 0x7fb04a6bec18>
{8, 2, 4, 6}
```

Python memoryview() Function

The python **memoryview()** function returns a memoryview object of the given argument.

Python memoryview () Function Example

```
#A random bytearray
randomByteArray = bytearray('ABC', 'utf-8')

mv = memoryview(randomByteArray)

# access the memory view's zeroth index
print(mv[0])

# It create byte from memory view
print(bytes(mv[0:2]))

# It create list from memory view
print(list(mv[0:3]))
```

Output:

```
65
b'AB'
[65, 66, 67]
```

Python object()

The python **object()** returns an empty object. It is a base for all the classes and holds the built-in properties and methods which are default for all the classes.

Python object() Example

```
python = object()

print(type(python))
print(dir(python))
```

Output:

```
<class 'object'>
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
'__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__',
'__str__', '__subclasshook__']
```

Python open() Function

The python **open()** function opens the file and returns a corresponding file object.

Python open() Function Example

```
# opens python.text file of the current directory
f = open("python.txt")
# specifying full path
f = open("C:/Python33/README.txt")
```

Output:

```
Since the mode is omitted, the file is opened in 'r' mode; opens for reading.
```

Python chr() Function

Python **chr()** function is used to get a string representing a character which points to a Unicode code integer. For example, chr(97) returns the string 'a'. This function takes an integer argument and throws an error if it exceeds the specified range. The standard range of the argument is from 0 to 1,114,111.

Python chr() Function Example

```
# Calling function
result = chr(102) # It returns string representation of a char
result2 = chr(112)
# Displaying result
print(result)
print(result2)
# Verify, is it string type?
print("is it string type:", type(result) is str)
```

Output:

```
ValueError: chr() arg not in range(0x110000)
```

Python complex()

Python **complex()** function is used to convert numbers or string into a complex number. This method takes two optional parameters and returns a complex number. The first parameter is called a real and second as imaginary parts.

Python complex() Example

```
# Python complex() function example
# Calling function
a = complex(1) # Passing single parameter
b = complex(1,2) # Passing both parameters
# Displaying result
```

```
print(a)
print(b)
```

Output:

```
(1.5+0j)
(1.5+2.2j)
```

Python delattr() Function

Python **delattr()** function is used to delete an attribute from a class. It takes two parameters, first is an object of the class and second is an attribute which we want to delete. After deleting the attribute, it no longer available in the class and throws an error if try to call it using the class object.

Python delattr() Function Example

```
class Student:
    id = 101
    name = "Pranshu"
    email = "pranshu@abc.com"
# Declaring function
    def getinfo(self):
        print(self.id, self.name, self.email)
s = Student()
s.getinfo()
delattr(Student,'course') # Removing attribute which is not available
s.getinfo() # error: throws an error
```

Output:

```
101 Pranshu pranshu@abc.com
AttributeError: course
```

Python dir() Function

Python **dir()** function returns the list of names in the current local scope. If the object on which method is called has a method named __dir__(), this method will be called and must return the list of attributes. It takes a single object type argument.

Python dir() Function Example

```
# Calling function
att = dir()
# Displaying result
print(att)
```

Output:

```
['__annotations__',  '__builtins__',  '__cached__',  '__doc__',  '__file__',
'__loader__',
'__name__', '__package__', '__spec__']
```

Python divmod() Function

Python **divmod()** function is used to get remainder and quotient of two numbers. This function takes two numeric arguments and returns a tuple. Both arguments are required and numeric

Python divmod() Function Example

```
# Python divmod() function example
# Calling function
result = divmod(10,2)
# Displaying result
print(result)
```

Output:

```
(5, 0)
```

Python enumerate() Function

Python **enumerate()** function returns an enumerated object. It takes two parameters, first is a sequence of elements and the second is the start index of the sequence. We can get the elements in sequence either through a loop or next() method.

Python enumerate() Function Example

```
# Calling function
result = enumerate([1,2,3])
# Displaying result
print(result)
print(list(result))
```

Output:

```
<enumerate object at 0x7ff641093d80>
[(0, 1), (1, 2), (2, 3)]
```

Python dict()

Python **dict()** function is a constructor which creates a dictionary. Python dictionary provides three different constructors to create a dictionary:

- If no argument is passed, it creates an empty dictionary.
- If a positional argument is given, a dictionary is created with the same key-value pairs. Otherwise, pass an iterable object.
- If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument.

Python dict() Example

```
# Calling function
result = dict() # returns an empty dictionary
result2 = dict(a=1,b=2)
# Displaying result
print(result)
print(result2)
```

Output:

```
{}
{'a': 1, 'b': 2}
```

Python filter() Function

Python **filter()** function is used to get filtered elements. This function takes two arguments, first is a function and the second is iterable. The filter function returns a sequence of those elements of iterable object for which function returns **true value**.

The first argument can be **none**, if the function is not available and returns only elements that are **true**.

Python filter() Function Example

```
# Python filter() function example
def filterdata(x):
    if x>5:
        return x
# Calling function
result = filter(filterdata,(1,2,6))
# Displaying result
print(list(result))
```

Output:

```
[6]
```

Python hash() Function

Python **hash()** function is used to get the hash value of an object. Python calculates the hash value by using the hash algorithm. The hash values are integers and used to compare dictionary keys during a dictionary lookup. We can hash only the types which are given below:

Hashable types: * bool * int * long * float * string * Unicode * tuple * code object.

Python hash() Function Example

```
# Calling function
result = hash(21) # integer value
result2 = hash(22.2) # decimal value
# Displaying result
print(result)
print(result2)
```

Output:

```
21
461168601842737174
```

Python help() Function

Python **help()** function is used to get help related to the object passed during the call. It takes an optional parameter and returns help information. If no argument is given, it shows the Python help console. It internally calls python's help function.

Python help() Function Example

```
# Calling function
info = help() # No argument
# Displaying result
print(info)
```

Output:

```
Welcome to Python 3.5's help utility!
```

Python min() Function

Python **min()** function is used to get the smallest element from the collection. This function takes two arguments, first is a collection of elements and second is key, and returns the smallest element from the collection.

Python min() Function Example

```
# Calling function
```

```
small = min(2225,325,2025) # returns smallest element
small2 = min(1000.25,2025.35,5625.36,10052.50)
# Displaying result
print(small)
print(small2)
```

Output:

```
325
1000.25
```

Python set() Function

In python, a set is a built-in class, and this function is a constructor of this class. It is used to create a new set using elements passed during the call. It takes an iterable object as an argument and returns a new set object.

Python set() Function Example

```
# Calling function
result = set() # empty set
result2 = set('12')
result3 = set('javatpoint')
# Displaying result
print(result)
print(result2)
print(result3)
```

Output:

```
set()
{'1', '2'}
{'a', 'n', 'v', 't', 'j', 'p', 'i', 'o'}
```

Python hex() Function

Python **hex()** function is used to generate hex value of an integer argument. It takes an integer argument and returns an integer converted into a hexadecimal string. In case, we want to get a hexadecimal value of a float, then use float.hex() function.

Python hex() Function Example

```
# Calling function
result = hex(1)
# integer value
result2 = hex(342)
# Displaying result
print(result)
print(result2)
```

Output:

```
0x1
0x156
```

Python id() Function

Python **id()** function returns the identity of an object. This is an integer which is guaranteed to be unique. This function takes an argument as an object and returns a unique integer number which represents identity. Two objects with non-overlapping lifetimes may have the same id() value.

Python id() Function Example

```
# Calling function
val = id("Javatpoint") # string object
val2 = id(1200) # integer object
val3 = id([25,336,95,236,92,3225]) # List object
# Displaying result
print(val)
print(val2)
print(val3)
```

Output:

```
139963782059696
139963805666864
139963781994504
```

Python setattr() Function

Python **setattr()** function is used to set a value to the object's attribute. It takes three arguments, i.e., an object, a string, and an arbitrary value, and returns none. It is helpful when we want to add a new attribute to an object and set a value to it.

Python setattr() Function Example

```
1. class Student:
2.     id = 0
3.     name = ""
4.
5.     def __init__(self, id, name):
6.         self.id = id
7.         self.name = name
8.
9.  student = Student(102,"Sohan")
10. print(student.id)
11. print(student.name)
12. #print(student.email) product error
13. setattr(student, 'email','sohan@abc.com') # adding new attribute
14. print(student.email)
```

Output:

```
102
Sohan
sohan@abc.com
```

Python slice() Function

Python **slice()** function is used to get a slice of elements from the collection of elements. Python provides two overloaded slice functions. The first function takes a single

argument while the second function takes three arguments and returns a slice object. This slice object can be used to get a subsection of the collection.

Python slice() Function Example

```
# Calling function
result = slice(5) # returns slice object
result2 = slice(0,5,3) # returns slice object
# Displaying result
print(result)
print(result2)
```

Output:

```
slice(None, 5, None)
slice(0, 5, 3)
```

Python sorted() Function

Python **sorted()** function is used to sort elements. By default, it sorts elements in an ascending order but can be sorted in descending also. It takes four arguments and returns a collection in sorted order. In the case of a dictionary, it sorts only keys, not values.

Python sorted() Function Example

```
str = "javatpoint" # declaring string
# Calling function
sorted1 = sorted(str) # sorting string
# Displaying result
print(sorted1)
```

Output:

```
['a', 'a', 'i', 'j', 'n', 'o', 'p', 't', 't', 'v']
```

Python next() Function

Python **next()** function is used to fetch next item from the collection. It takes two arguments, i.e., an iterator and a default value, and returns an element.

This method calls on iterator and throws an error if no item is present. To avoid the error, we can set a default value.

Python next() Function Example

```
number = iter([256, 32, 82]) # Creating iterator
# Calling function
item = next(number)
# Displaying result
print(item)
# second item
item = next(number)
print(item)
# third item
item = next(number)
print(item)
```

Output:

```
256
32
82
```

Python input() Function

Python **input()** function is used to get an input from the user. It prompts for the user input and reads a line. After reading data, it converts it into a string and returns it. It throws an error **EOFError** if EOF is read.

Python input() Function Example

```
# Calling function
val = input("Enter a value: ")
# Displaying result
print("You entered:",val)
```

Output:

```
Enter a value: 45
You entered: 45
```

Python int() Function

Python **int()** function is used to get an integer value. It returns an expression converted into an integer number. If the argument is a floating-point, the conversion truncates the number. If the argument is outside the integer range, then it converts the number into a long type.

If the number is not a number or if a base is given, the number must be a string.

Python int() Function Example

```
# Calling function
val = int(10) # integer value
val2 = int(10.52) # float value
val3 = int('10') # string value
# Displaying result
print("integer values :",val, val2, val3)
```

Output:

```
integer values : 10 10 10
```

Python isinstance() Function

Python **isinstance()** function is used to check whether the given object is an instance of that class. If the object belongs to the class, it returns true. Otherwise returns False. It also returns true if the class is a subclass.

The **isinstance()** function takes two arguments, i.e., object and classinfo, and then it returns either True or False.

Python isinstance() function Example

```
class Student:
```

```
    id = 101
    name = "John"
    def __init__(self, id, name):
        self.id=id
        self.name=name

student = Student(1010,"John")
lst = [12,34,5,6,767]
# Calling function
print(isinstance(student, Student)) # isinstance of Student class
print(isinstance(lst, Student))
```

Output:

```
True
False
```

Python oct() Function

Python **oct()** function is used to get an octal value of an integer number. This method takes an argument and returns an integer converted into an octal string. It throws an error **TypeError**, if argument type is other than an integer.

Python oct() function Example

```
# Calling function
val = oct(10)
# Displaying result
print("Octal value of 10:",val)
```

Output:

```
Octal value of 10: 0o12
```

Python ord() Function

The python **ord()** function returns an integer representing Unicode code point for the given Unicode character.

Python ord() function Example

```
# Code point of an integer
print(ord('8'))

# Code point of an alphabet
print(ord('R'))

# Code point of a character
print(ord('&'))
```

Output:

```
56
82
38
```

Python pow() Function

The python **pow()** function is used to compute the power of a number. It returns x to the power of y. If the third argument(z) is given, it returns x to the power of y modulus z, i.e. (x, y) % z.

Python pow() function Example

```
# positive x, positive y (x**y)
print(pow(4, 2))

# negative x, positive y
print(pow(-4, 2))

# positive x, negative y (x**-y)
print(pow(4, -2))

# negative x, negative y
```

11. `print(pow(-4, -2))`

Output:

```
16
16
0.0625
0.0625
```

Python print() Function

The python **print()** function prints the given object to the screen or other standard output devices.

Python print() function Example

```
print("Python is programming language.")

x = 7
# Two objects passed
print("x =", x)

y = x
# Three objects passed
print('x =', x, '= y')
```

Output:

```
Python is programming language.
x = 7
x = 7 = y
```

Python range() Function

The python **range()** function returns an immutable sequence of numbers starting from 0 by default, increments by 1 (by default) and ends at a specified number.

Python range() function Example

```
# empty range
print(list(range(0)))

# using the range(stop)
print(list(range(4)))

# using the range(start, stop)
print(list(range(1,7 )))
```

Output:

```
[]
[0, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

Python reversed() Function

The python **reversed()** function returns the reversed iterator of the given sequence.

Python reversed() function Example

```
# for string
String = 'Java'
print(list(reversed(String)))

# for tuple
Tuple = ('J', 'a', 'v', 'a')
print(list(reversed(Tuple)))

# for range
Range = range(8, 12)
print(list(reversed(Range)))

# for list
List = [1, 2, 7, 5]
print(list(reversed(List)))
```

Output:

```
['a', 'v', 'a', 'J']
['a', 'v', 'a', 'J']
[11, 10, 9, 8]
[5, 7, 2, 1]
```

Python round() Function

The python **round()** function rounds off the digits of a number and returns the floating point number.

Python round() Function Example

```
# for integers
print(round(10))

# for floating point
print(round(10.8))

# even choice
print(round(6.6))
```

Output:

```
10
11
7
```

Python issubclass() Function

The python **issubclass()** function returns true if object argument(first argument) is a subclass of second class(second argument).

Python issubclass() Function Example

```
class Rectangle:
  def __init__(rectangleType):
    print('Rectangle is a ', rectangleType)
```

```
4.
5.  class Square(Rectangle):
6.    def __init__(self):
7.      Rectangle.__init__('square')
8.
9.  print(issubclass(Square, Rectangle))
10. print(issubclass(Square, list))
11. print(issubclass(Square, (list, Rectangle)))
12. print(issubclass(Rectangle, (list, Rectangle)))
```

Output:

```
True
False
True
True
```

Python str

The python **str()** converts a specified value into a string.

Python str() Function Example

```
1.  str('4')
```

Output:

```
'4'
```

Python tuple() Function

The python **tuple()** function is used to create a tuple object.

Python tuple() Function Example

```
1.  t1 = tuple()
2.  print('t1=', t1)
3.
4.  # creating a tuple from a list
```

```
t2 = tuple([1, 6, 9])
print('t2=', t2)

# creating a tuple from a string
t1 = tuple('Java')
print('t1=',t1)

# creating a tuple from a dictionary
t1 = tuple({4: 'four', 5: 'five'})
print('t1=',t1)
```

Output:

```
t1= ()
t2= (1, 6, 9)
t1= ('J', 'a', 'v', 'a')
t1= (4, 5)
```

Python type()

The python **type()** returns the type of the specified object if a single argument is passed to the type() built in function. If three arguments are passed, then it returns a new type object.

Python type() Function Example

```
List = [4, 5]
print(type(List))

Dict = {4: 'four', 5: 'five'}
print(type(Dict))

class Python:
    a = 0

InstanceOfPython = Python()
print(type(InstanceOfPython))
```

Output:

```
<class 'list'>
<class 'dict'>
<class '__main__.Python'>
```

Python vars() function

The python **vars()** function returns the __dict__ attribute of the given object.

Python vars() Function Example

```
class Python:
  def __init__(self, x = 7, y = 9):
    self.x = x
    self.y = y

InstanceOfPython = Python()
print(vars(InstanceOfPython))
```

Output:

```
{'y': 9, 'x': 7}
```

Python zip() Function

The python **zip()** Function returns a zip object, which maps a similar index of multiple containers. It takes iterables (can be zero or more), makes it an iterator that aggregates the elements based on iterables passed, and returns an iterator of tuples.

Python zip() Function Example

```
numList = [4,5, 6]
strList = ['four', 'five', 'six']

# No iterables are passed
result = zip()

```

```
# Converting itertor to list
resultList = list(result)
print(resultList)

# Two iterables are passed
result = zip(numList, strList)

# Converting itertor to set
resultSet = set(result)
print(resultSet)
```

Output:

```
[]
{(5, 'five'), (4, 'four'), (6, 'six')}
```