**1. Built in functions in Python**

**1.abs()**

Returns the absolute value of a number

ex: number = -4

print(abs(number))

result : 4

**2. all()**

Returns True if all items in an iterable object are true

Ex: dict1 = {1: 'True', 2: 'False'}

print(all(dict1)

result: True

**3. any()**

Returns True if any item in an iterable object is true

Ex: print (any ([False, False, False, False])) result: False

   print (any([False, True, False, False])) result : True

**4.ascii()**

Returns a readable version of an object. Replaces none-ascii characters with escape character

**5. bin()**

Returns the binary version of a number

Ex: print(bin(11)) result: 0b1011

**6. bool()**

Returns the boolean value of the specified object

Ex: print(bool(2)) result: True

Print(bool(0)) result: False

**7. bytearray()**

Returns an array of bytes

Ex:

**8. bytes()**

Returns a bytes object

**9. callable()**

Returns True if the specified object is callable, otherwise False

Ex: x = 5

print('int :',callable(x))

def test():

return 'hey'

print('fun :', callable(test))

class foo2:

def \_\_call\_\_(): # \_\_init\_\_, \_\_random\_\_

return 2 + 4

print('class :', callable(foo2))

OUTPUT:

int : False

fun : True

class : True

**11.classmethod()**

Converts a method into a class method

**12.compile()**

Returns the specified source as an object, ready to be executed

Ex : srcCode = 'x = 10\ny = 20\nmul = x \* y\nprint("mul =",mul)'

execCode = compile(srcCode,'','exec')

print(type(execCode))

print(exec(execCode))

OUTPUT:

<class 'code'>

mul = 200

**13.complex()**

Returns a complex number

Ex: a = 7

Print(complex(a)) result : 7 + 0j

**14. delattr()**

Deletes the specified attribute (property or method) from the specified object

**15.dict()**

Returns a dictionary (Array)

Ex : dict(a=1, b={'c':5})

Output: {'a': 1, 'b': {'c': 5}}

**16.dir()**

Returns a list of the specified object's properties and methods

Ex : a = ‘anjan’

Output : ['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

**17.divmod()**

Returns the quotient and the remainder when argument1 is divided by argument2

EX: divmod(10,4)

Output : (2, 2)

**18 . enumerate()**

Takes a collection (e.g. a tuple) and returns it as an enumerate object

EX: a = [1,2,3]

Print(list(enumerate(a))

Output:[(0, 1), (1, 2), (2, 3)]

**19. eval()**

Evaluates and executes an expression

EX: eval('-1')

Output : -1

**20. exec()**

Executes the specified code (or object)

EX: exec("print(min(10,1))")

Output : 1

**21. filter()**

Use a filter function to exclude items in an iterable object

Ex : def fun(var):

letters = ['a', 'e', 'i', 'o', 'u']

if var in letters:

return True

else:

return False

seq = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']

filtered = filter(fun, sequence)

print('The filtered letters are:')

for s in filtered:

print(s)

output: The filtered letters are: e, e

|  |
| --- |
|  |

**22. float()**

Returns a floating-point number

Ex : float(“inf”) 🡺inf

Float(“7”) 🡺 7.0

**23 . format()**

Formats a specified value

Ex : a = “anjan “

B = “kumar”

Print(“{} and {} are friends “.format(a,b))

Output : anjan and kumar are friends

**24 . frozenset()**

Returns a frozenset object

Ex: a = {"anjan":1 ,"kumar":3}

print(frozenset(a))

output : frozenset({'anjan', 'kumar'})

**25 . getattr()**

Returns the value of the specified attribute (property or method)

EX : class Person:

age = 23

name = "Adam"

person = Person()

# when default value is provided

print('The sex is:', getattr(person, 'sex', 'Male'))

# when no default value is provided

print('The sex is:', getattr(person, 'sex')

**Output**

The sex is: Male

AttributeError: 'Person' object has no attribute 'sex'

**26. globals()**

Returns the current global symbol table as a dictionary

EX : age = 23

globals()[‘age’] = 25

print(“my age is “ , age)

output : my age is 25

**27. hasattr()**

Returns True if the specified object has the specified attribute (property/method)

EX: class Person:

age = 23

name = 'Adam'

person = Person()

print('Person has age?:', hasattr(person, 'age'))

print('Person has salary?:', hasattr(person, 'salary'))

output : Person has age?: True

Person has salary?: False

**28. hash()**

Returns the hash value of a specified object

Ex: # hash for integer unchanged

print('Hash for 181 is:', hash(181))

# hash for decimal

print('Hash for 181.23 is:',hash(181.23))

# hash for string

print('Hash for Python is:', hash('Python'))

Output : Hash for 181 is: 181

Hash for 181.23 is: 530343892119126197

Hash for Python is: 2230730083538390373

**29. help()**

Executes the built-in help system and gives the methods how we can use that.

**30. hex()**

Converts a number into a hexadecimal value

EX : number = 435

print(number, 'in hex =', hex(number))

number = 0

print(number, 'in hex =', hex(number))

number = -34

print(number, 'in hex =', hex(number))

returnType = type(hex(number))

print('Return type from hex() is', returnType)

Output : 435 in hex = 0x1b3

0 in hex = 0x0

-34 in hex = -0x22

Return type from hex() is <class 'str'>

**31.id()**

Returns the id of an object i.e memory address

**32.input()**

Allowing user input

In python3 input() method default a string.

**33. int()**

Returns an integer number

EX : a = “89”

Print(int(a))

Output : 89

**34. isinstance()**

Returns True if a specified object is an instance of a specified object

**35. issubclass()**

Returns True if a specified class is a subclass of a specified object

issubclass() returns:

* True if class is subclass of a class, or any element of the tuple
* False otherwise

EX: class Polygon:

def \_\_init\_\_(polygonType):

print('Polygon is a ', polygonType)

class Triangle(Polygon):

def \_\_init\_\_(self):

Polygon.\_\_init\_\_('triangle')

print(issubclass(Triangle, Polygon))

print(issubclass(Triangle, list))

print(issubclass(Triangle, (list, Polygon)))

print(issubclass(Polygon, (list, Polygon)))

Output :

True

False

True

True

**36. iter()**

Returns an iterator object

The iter() function creates an object which can be iterated one element at a time.

Ex: # list of vowels

vowels = ['a', 'e', 'i', 'o', 'u']

vowels\_iter = iter(vowels)

print(next(vowels\_iter)) # 'a'

print(next(vowels\_iter)) # 'e'

print(next(vowels\_iter)) # 'i'

print(next(vowels\_iter)) # 'o'

print(next(vowels\_iter)) # 'u'

**37. len()**

Returns the length of an object

EX : a = “anjankumar”

Print(len(a))

Output : 10

**38. list()**

Returns a list

The list() constructor returns a list.

* If no parameters are passed, it returns an empty list
* If iterable is passed as a parameter, it creates a list consisting of iterable's items.

Ex :

# empty list

print(list())

# vowel string

vowel\_string = 'aeiou'

print(list(vowel\_string))

Output :

[]

['a', 'e', 'i', 'o', 'u']

**39. locals()**

Returns an updated dictionary of the current local symbol table

Global symbol table

Local symbol table

A Global symbol table stores all information related to the global scope of the program, and is accessed in Python using globals() method.

The global scope contains all functions, variables which are not associated with any class or function.

Likewise, Local symbol table stores all information related to the local scope of the program, and is accessed in Python using locals() method.

The local scope could be within a function, within a class, etc.

**40. map()**

Returns the specified iterator with the specified function applied to each item

Ex:

def calculateSquare(n):

return n\*n

numbers = (1, 2, 3, 4)

result = map(calculateSquare, numbers)

print(result)

# converting map object to set

numbersSquare = set(result)

print(numbersSquare)

Output:

<map object at 0x7f722da129e8>

{16, 1, 4, 9}

**41. max()**

Returns the largest item in an iterable

**42. memoryview()**

Returns a memory view object

**43. min()**

Returns the smallest item in an iterable

**44. next()**

Returns the next item in an iterable

**45. object()**

Returns a new object

**46. oct()**

Converts a number into an octal

EX: # binary to octal

print('oct(0b101) is:', oct(0b101))

# hexadecimal to octal

print('oct(0XA) is:', oct(0XA))

Output :

oct(10) is: 0o12

oct(0b101) is: 0o5

oct(0XA) is: 0o12

**47. open()**

Opens a file and returns a file object

**48. ord()**

Convert an integer representing the Unicode of the specified character

EX: print(ord('5')) # 53

print(ord('A')) # 65

print(ord('$')) # 36

**49. pow()**

Returns the value of x to the power of y

The pow() function takes three parameters:

* **x** - a number, the base
* **y** - a number, the exponent
* **z (optional)** - a number, used for modulus

Hence,

* pow(x, y) is equal to xy
* pow(x, y, z) is equal to xy % z

EX:

# positive x, positive y (x\*\*y)

print(pow(2, 2)) # 4

# negative x, positive y

print(pow(-2, 2)) # 4

# positive x, negative y

print(pow(2, -2)) # 0.25

# negative x, negative y

print(pow(-2, -2)) # 0.25

**50. print()**

Prints to the standard output device

**51. property()**

Gets, sets, deletes a property

**52.range()**

Returns a sequence of numbers, starting from 0 and increments by 1 (by default)

The range() type returns an immutable sequence of numbers between the given start integer to the stop integer.

**53. repr()**

Returns a readable version of an object

EX :

var = 'foo'

print(repr(var)) 🡺 'foo'

**54.reversed()**

Returns a reversed iterator

**55. round()**

Rounds a numbers

The round() function takes two parameters:

1. number - the number to be rounded

2. ndigits (optional) - number up to which the given number is rounded; defaults to 0

EX: # for integers

print(round(10)) 🡺 10

# for floating point

print(round(10.7)) 🡺 11

# even choice

print(round(5.5)) 🡺 6

**56. set()**

Returns a new set object

**57. setattr()**

Sets an attribute (property/method) of an object

EX:

class Person:

name = 'Adam'

p = Person()

print('Before modification:', p.name)

# setting name to 'John'

setattr(p, 'name', 'John')

print('After modification:', p.name

Output :

Before modification: Adam

After modification: John

**58. slice()**

Returns a slice object

**59. sorted()**

Returns a sorted list

**60. @staticmethod()**

Converts a method into a static method

**61. str()**

Returns a string object

**62. sum()**

Sums the items of an iterator

**63. super()**

Returns an object that represents the parent class

**64. tuple()**

Returns a tuple

**65. type()**

Returns the type of an object

**66. vars()**

Returns the \_\_dict\_\_ property of an object

EX:

class Foo:

def \_\_init\_\_(self, a = 5, b = 10):

self.a = a

self.b = b

object = Foo()

print(vars(object))

Output: {'a': 5, 'b': 10}

**67. zip()**

Returns an iterator, from two or more iterators

The zip() function takes iterables (can be zero or more), aggregates them in a tuple, and return it.

The zip()function returns an iterator of tuples based on the iterable objects.

* If we do not pass any parameter, zip() returns an empty iterator
* If a single iterable is passed, zip() returns an iterator of tuples with each tuple having only one element.
* If multiple iterables are passed, zip() returns an iterator of tuples with each tuple having elements from all the iterables.  
    
  Suppose, two iterables are passed to zip(); one iterable containing three and other containing five elements. Then, the returned iterator will contain three tuples. It's because iterator stops when the shortest iterable is exhausted.

EX:

number\_list = [1, 2, 3]

str\_list = ['one', 'two', 'three']

# No iterables are passed

result = zip()

# Converting iterator to list

result\_list = list(result)

print(result\_list)

# Two iterables are passed

result = zip(number\_list, str\_list)

# Converting iterator to set

result\_set = set(result)

print(result\_set)

Output:

[]

{(2, 'two'), (3, 'three'), (1, 'one')}

**2. What is list comprehension and lambda function.?**

List comprehension is used to **create a new list based on the existing list**. The list comprehension is used to work on the list. List comprehension **returns the list** and It contains **expression and brackets.**

**Ex: a = [i for i in range(10) if i%2==0]**

**Result : [0,2,4,6,8]**

Lambda is a function that is defined without a name. Usually in python **functions**are defined with a **keyword def**but **anonymous functions**are defined by the **keyword lambda.** The lambda function is used along with the built-in function **map(), filter().**

**Ex :** cube = lambda a: a \* a \* a

print(cube(5))

Result : 125

**3. Difference between Generator and Decorator.?**

[**Decorators**](https://www.geeksforgeeks.org/function-decorators-in-python-set-1-introduction/) are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.

**Generator-Object :** Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for in” loop.

**4. Why to use decorator.?**

Python has an interesting feature called **decorators** to add functionality to an existing code.

This is also called **metaprogramming** because a part of the program tries to modify another part of the program at compile time.

If you want to add a functionality before and after with changing the original functionality of a function then go for decorator.

**5. Why we go for Generator.?**

Now we introduce an important type of object called a **generator**, which allows us to generate arbitrarily-many items in a series, without having to store them all in memory at once.

**6. Difference between List comprehension and Tuple comprehension.?**

List comprehension will give the object as list.

Tuple comprehension will gives the generator object.