Python Tutorials
Introduction to Python
Features of Python
Pros & Cons of Python
Python Books
Syntax of Python
Applications of Python
Python Variables
Python Variable Scope
Python Identifiers
Python Namespace & Scope
Python Operators
Python Bitwise Operators
Python Comparison Operators
Python Operator Precedence
Python Operator Overloading
Python Ternary Operator
Python Numbers
Python Strings
Python String Functions
Python Data Structures
Python Lists
Python List Comprehension
Python Tuples
Python Dictionaries
Python Sets
Python Booleans
Python Tuples vs Lists
Python Sequences
Python Collections
Python Slicing
Python Decision Making
Python Switch
Python Loops
Python Functions
Python Built-in Functions
Python Range Function
Python Zip Function
Python Classes
Python Objects
Python Polymorphism
Python Inheritance
Python Multiple Inheritance
Python Constructors
Python Iterators
Python Iterables
Python Decorators
Python Generators
Python Counter
Python Exception Handling
Python Errors & Exceptions
Python Comments, Statements
Python Installation

Python Built-in Functions – Learn the functions with syntax and examples

In this article, we are going to see all the **pre-defined functions** that are available in Python.

You have already used some of the Python **built-in functions**, for example, the **print() function** is used to **output** a **string** on the **console**. As of now, the latest version of **Python 3.8 has 69 built-in functions**.

We will go through each of them.

Python Built-in Functions

The Python interpreter contains a **number of functions** that are always **available** to use anywhere in the program. These functions are **built-in** functions.

Below is the list of all the available built-in functions in **chronological order**.

List of Python Built-in Functions

abs()	enumerate()	iter()	reversed()
all()	eval()	len()	round()
any()	exec()	list()	set()
ascii()	filter()	locals()	setattr()
bin()	float()	map()	slice()
bool()	format()	max()	sorted()
breakpoint()	frozenset()	memoryview()	staticmethod()
bytearray()	getattr()	min()	str()
bytes()	globals()	next()	sum()
callable()	hasattr()	object()	super()
chr()	hash()	oct()	tuple()
classmethod()	help()	open()	type()
compile()	hex()	ord()	vars()
complex()	id()	pow()	zip()
delattr()	input()	print()	__import__()
dict()	int()	property()	
dir()	isinstance()	range()	
divmod()	issubclass()	repr()	

Here is a detailed explanation of built-in functions in **Python**.

1. abs(x)

The **abs() function** returns the **absolute value** of the number which is the **distance of a point** from **zero index**. The **argument x** can be an **integer** or a **floating-point value**. In case of complex numbers, their **magnitude** is **returned**.

Code:

```
print( abs(4) )
print( abs(-2.5) )
print( abs(3j + 2) )
```

Output:

```
4
2.5
3.6055512754639896
```

2. all(*iterable*)

The **all() function** takes an **iterable** container as an **argument** and returns **True** when **all elements** of the iterable are **True(or is empty) otherwise** it returns **False**.

Code:

```
print( all([]))
print( all([True, True, False]))
print( all({1,3,5,2}))
```

Output:

```
True
False
True
```

3. any(*iterable*)

The **any() function** takes an **iterable** container as an **argument** and returns **True** when **one of the elements** inside the iterable container is **True**, **otherwise**, it returns **False**.

Code:

```
print( any([]) )
print( any([False, False, True, 1, 3]) )
print( any({10,20,30,40}) )
```

Output:

```
False
True
True
```

4. ascii(object)

The **ascii() function** returns a **printable representation** of the **object**. It escapes the **non-ASCII** characters in the **string**.

The string returned by **ascii()** is similar to the **repr() function** in Python2.

Code:

```
ascii(2020)
ascii(‘a’)
ascii(‘Hello \n World’)
```

Output:

```
‘2020’
“‘a’”
“‘Hello \\n World'”
```

5. bin(x)

The function **bin()** will convert an **integer** into its **binary representation** in **string format**.

Binary numbers are **prefixed** with **‘ob’**. It only takes **integer numbers** and giving a **string** or a **float value** to the function will result in an **error**.

Code:

```
bin(12)
bin(-12)
```

Output:

```
‘0b1100’
‘-0b1100’
```

6. bool([x])

The **bool()** function returns a **True** or **False** by converting the argument into a **boolean value.** It returns **True** when the **argument passed** is **True** otherwise **empty containers** and **False** value will **return False**.

- Python Interpreter
- Python Function Arguments
- Python Recursion
- Python Methods
- Python Methods vs Functions
- Python Exec()
- Python Eval()
- Python Repr()
- Python Modules
- Python pprint Module
- Python Sys Module
- Python Defaultdict Module
- Python OrderedDict Module
- Python Calendar Module
- Python Itertools Module
- Python Iterator vs Generator
- Python Assertion
- Python Deep Copy & Shallow ...
- Python Terminologies Part 1
- Python Terminologies Part 2
- Python Facts
- Python Career
- Become a Python Developer
- Python Demand in Market
- Python Resume
- Python Libraries for Data Scie...
- Python Web Frameworks
- Data Visualization in Python
- Matplotlib Introduction
- Python Lambda Expression
- Python Read Write File
- Modules in Python
- Python Compilers & Interpret...
- Closures in Python
- Python NumPy Introduction
- Python for Machine Learning
- Python SciPy Introduction
- SciPy Installation

Python Projects

- Python Projects with Source C...

Python Interview Questions

- Python Interview Ques Part-1
- Python Interview Ques Part-2
- Python Interview Ques Part-3

Code:

```
bool(False)
bool([])
bool(20)
bool({1,2,4})
```

Output:

```
False
False
True
True
```

7. breakpoint(*args, **kws)

The **breakpoint()** function is introduced from **Python 3.7** and it helps in **debugging**.

For example, when you use **pdb debugger** then you **call** the **pdb.set_trace()** in your **program code**. Then for a machine that has **web-pdb debugger** will have to **change** the **code** to **web-pdb.set_trace()** method.

This becomes an **overhead** and for that, we have the **breakpoint()** method which allows us to write **loosely coupled debugging code**.

Code:

```
msg = “Hi”
breakpoint()
```

Output:

```
>
c:\users\Techvidvan\appdata\local\programs\python\python37-32\bp.py(4)<module>()
-> print(msg)
(Pdb) msg
‘Hi’
(Pdb)
```

8. bytearray([source[, encoding[, errors]]])

It returns a **mutable version** of **bytes array** of integers between **0-256**.

- If an **integer** is passed, then it will return us an array of that size with **null bytes**.
- If a **string** is passed, then it is necessary to provide **encoding** in the second argument.

Code:

```
bytearray(4)
bytearray('abc','utf-8')
bytearray([1,2,3])
```

Output:

```
bytearray(b’\x00\x00\x00\x00')
bytearray(b’abc’)
bytearray(b’\x01\x02\x03')
```

9. bytes([source[, encoding[, errors]]])

The **byte()** function is similar to the **bytearray()** function. The only difference is that **bytes() returns** an **immutable object**. We **cannot change elements** of a bytes function.

Code:

```
bytes(3)
bytes([3,2,1])
```

Output:

```
b’\x00\x00\x00'
b’\x03\x02\x01'
```

10. callable(Object)

The **callable function** tells us whether an **object** is **callable** or **not**. It returns **True** when the argument passed is **callable** otherwise it returns **False**.

User-defined and all the **built-in functions** are **callable**.

Code:

```
callable(print)
callable([1,2,3])
callable(abs)
```

Output:

```
True
False
True
```

11. chr(i)

The function **chr()** is an **inverse** of **ord()** function. It takes **unicode code** point as an **argument** and **returns** the **string representation** of the **character**.

The **input range** is from **0** to **1,114,111**. Outside of this range, it will raise an **error**.

Code:

```
chr(65)
chr(120)
```

Output:

```
‘A’
‘x’
```

12. @classmethod()

The **@classmethod()** is a **decorator** that is used to **create class methods** that will be **passed** on all the **objects** just like self is **passed**.

Syntax:

```
@classmethod()
def func(cls, args...):
  ...
```

Code:

```
class Person:

    @classmethod
    def display(cls):
        print("Person's age is 42")


Person.display()
```

Output:

```
Person’s age is 42
```

13. compile(source, filename, mode)

The **compile()** functions **compiles** the **source code** into an **executable object**. The object can be **executed** by using **exec()** or **eval() functions**.

The first parameter is the **source code**, second is the **filename** and the third parameter is the **mode**.

Code:

```
exec(compile('num1=10;num2=20;print(num1+num2);','', 'exec'))
```

Output:

```
30
```

14. complex([real[, imag]])

The **complex()** function **returns** or **converts** a **number** into a **complex number**.

The **first argument** is the **real part** of the complex number and the **second argument(optional)** is the **imaginary part**.

Code:

```
complex(1,2)
complex(5.5)
complex(3+7j)
```

Output:

```
(1+2j)
(5.5+0j)
(3+7j)
```

15. delattr(object, name)

The **delattr() function** is used to **delete** an **attribute** of an **object**. It takes **two arguments**, the object from which you want to **delete** and the **attribute name** that you want to **delete**.

You can only delete the attribute when you have **permission** for it.

Code:

```
class Car:
  color = ‘Blue’

c = Car()
print(c.color)
delattr(c, ‘color’)
print(c.color)
```

Output:

```
Blue
Traceback (most recent call last):
  File “C:/Users/Techvidvan/AppData/Local/Programs/Python/Python37-32/bp.py”, line 6, in <module>
    delattr(c, ‘color’)
AttributeError: color
```

16. dict()

The **dict()** function **returns** or **creates** a new **dictionary** which is useful in **mapping values**. It takes an **iterable**.

Code:

```
Numbers1 = dict(a=1, b=2, c=3, d=4)
Numbers2 = dict([(‘a’, 1), (‘b’: 2), (‘c’:3), (‘d’:4) ])
print(Numbers1)
print(Numbers2)
```

Output:

```
{‘a’ : 1, ‘b’: 2, ‘c’:3, ‘d’:4 }
{‘a’ : 1, ‘b’: 2, ‘c’:3, ‘d’:4 }
```

17. dir([object])

The **dir()** object returns a **list of all the names** of the current **local scope** if **no argument** is **passed**.

Code:

```
Variable1 = 10
Variable2 = ‘Hey’
dir()
```

Output:

```
[‘Variable1’, ‘Variable2’, ‘__annotations__’, ‘__builtins__’, ‘__doc__’, ‘__loader__’, ‘__name__’, ‘__package__’, ‘__spec__’]
```

When we pass an **object** as an **argument** then it will return a **list** of all the **valid attribute names** of that **object**.

Let’s see the attributes of a **string**.

Code:

```
dir(str)
```

Output:

```
[‘__add__’, ‘__class__’, ‘__contains__’, ‘__delattr__’, ‘__dir__’, ‘__doc__’, ‘__eq__’, ‘__ge__’,’__format__’,
‘__getattribute__’, ‘__getitem__’, ‘__getnewargs__’, ‘__gt__’, ‘__hash__’, ‘__init__’, ‘__init_subclass__’, ‘__iter__’,
‘__le__’, ‘__len__’, ‘__lt__’, ‘__mod__’, ‘__mul__’, ‘__ne__’, ‘__new__’, ‘__reduce__’, ‘__repr__’, ‘__reduce_ex__’, ‘__rmod__’,
‘__rmul__’, ‘__setattr__’, ‘__sizeof__’, ‘__str__’, ‘__subclasshook__’, ‘capitalize’, ‘casefold’, ‘center’, ‘count’, ‘encode’,
‘endswith’, ‘expandtabs’, ‘find’, ‘format’, ‘format_map’, ‘index’, ‘isalnum’, ‘isdecimal’, ‘isalpha’, ‘isdigit’, ‘isidentifier’,
‘islower’, ‘isnumeric’, ‘isprintable’, ‘isspace’, ‘istitle’, ‘isupper’, ‘join’, ‘lstrip’, ‘ljust’, ‘lower’, ‘maketrans’,
‘partition’, ‘replace’, ‘rfind’, ‘rindex’, ‘rjust’, ‘rpartition’, ‘rsplit’, ‘rstrip’, ‘split’, ‘splitlines’, ‘startswith’,
‘strip’, ‘swapcase’, ‘title’, ‘translate’, ‘upper’, ‘zfill’]
```

18. divmod(a,b)

The function **divmod()** takes **two integer** or **float numbers** as **arguments** and then **returns** a **tuple** whose **first element** is the **quotient** and **second element** is **remainder**.

Code:

```
divmod(20,2)
divmod(48,5)
divmod(11,2.5)
```

Output:

```
(10, 0)
(9, 3)
(4.0, 1.0)
```

19. enumerate(iterable, start=0)

The function returns us an **enumerate object** which is used in **loops** to **iterate** over **iterable objects**. It is useful when we want to have a **counter** to **calculate something**.

The **numbers start** from **zero** if you want to start with **another number** then you can **specify** that in the **second argument**.

Code:

```
for i, countryin enumerate([‘USA’, ‘UK’, ‘NYC’, ‘TKY’ ]):
  print(i, country)
```

Output:

```
0 USA
1 UK
2 NYC
3 TKY
```

20. eval()

The **eval()** function **evaluates** a **Python expression** that is **passed** in a **string**. It **parses** into a **Python expression** and then the **function evaluates it**.

Code:

```
x=5
eval(‘10<20’)
eval(‘x+ 10’)
```

Output:

```
True
15
```

21. exec()

The **exec()** function is used to **execute** or **run a Python code dynamically**. We can write Python code in a **string** and **pass** it as an argument to the **exec() function**. It will parse the **string** and **execute** the **Python code** inside it.

Code:

```
exec(‘print(“Hello”)’)
exec('a=20;b=30; print(a*b)')
```

Output:

```
Hello
600
```

22. filter(function, iterable)

The **filter function** is used to **filter out the data**. It does that by iterating on the **second iterable argument** and the **first argument** is a function that decides **how we will filter** the **elements**. This is mostly used with **lambda expressions**.

Code:

```
list(filter(lambda x:x>5 ,[1,2,3,4,5,6,8,10]))
```

Output:

```
[6,8,10]
```

Here we used a **lambda function** in which we want the **elements greater than 5** and the **list** is filtered out the elements **less than** or **equal** to 5.

23. float([x])

The **float** functions **returns** or **convert** the argument into a **floating-point** value if it is **compatible**. We can convert **integers** and **strings** that only contain **digits**.

Code:

```
float(45)
float(‘12’)
```

Output:

```
45.0
12.0
```

A **complex number** or a **string** with other characters like **alphabets** will raise an **error**.

24. format(value[, format_spec])

The **format()** function is similar to the **format method** in **strings**. It is used to **modify** a value according to a **specific format**.

The **first argument** is the value that needs to be **formatted** and the **second argument** is the specifier of how **value** is **specified**.

Code:

```
#format decimal number into binary value
format( 24, “b” )

#format a float value to have two decimal digits.
format(123.456, “0.2f” )
```

Output:

```
‘11000’
‘123.46’
```

25. frozenset([iterable])

The **frozenset()** function takes an **iterable** as an **argument** and **converts** it into an **immutable set**.

Sets are **mutable** by **default**. If we want the **same properties** of **set** but in an **immutable object** then we use **frozenset**.

Code:

```
frozenset({1,2,3,4})
frozenset([30, 20, 10])
frozenset((1, 2.5, 8.5, 4))
```

Output:

```
frozenset({1, 2, 3, 4, 5})
frozenset({10, 20, 30})
frozenset({8.5, 1, 2.5, 4})
```

26. getattr(object, name)

The **getattr()** function is **used** to get the **value** of an **object's attribute**.

The **first argument** is the **object** from which you want the **value** and the **second argument** is a **string** that represents the **name** of the **attribute**.

Code:

```
class Car:
  color = ‘Blue’

c = Car()
print( getattr(c, ‘color’) )
```

Output:

```
Blue
```

27. globals()

The function returns a **dictionary** in which all the global objects are **accessible** in the **current scope** or **module**.

Let's create a **list** in **global scope** and see the **dictionary** of **objects** in the **global scope**.

Code:

```
list1=[1,2,3,4]
globals()
```

Output:

```
{‘__name__’: ‘__main__’, ‘__doc__’: None, ‘__package__’: None, ‘__loader__’: <class ‘_frozen_importlib.BuiltinImporter’>,
‘__spec__’: None, ‘__annotations__’: {}, ‘__builtins__’: <module ‘builtins’ (built-in)>, ‘Car’: <class ‘__main__.Car’>, ‘c’:
<__main__.Car object at 0x03A79208>, ‘list1’: [1, 2, 3, 4]}
```

28. hasattr(object, name)

This function is also **similar** to the **getattr()** function instead it checks if the **object** contains the **specified attribute** or not. It returns a **boolean value**.

Code:

```
class Car:
  color= “Green”

c= Car()
print(hasattr(c, “color”))
print(hasattr(c, “price”))
```

Output:

```
True
False
```

29. hash(object)

In Python, everything is an object, **numbers**, **strings**, etc. all are object.

The **hashable objects** are **mapped** with an **integer value** in **Python**. The function **hash() returns** us the hash of the **specified object**.

Code:

```
print( hash(45) )
print( hash(“hello”) )
print( hash(94387593420) )
print( hash(True )
print(hash(2.5))
```

Output:

```
45
-1010369850
2045796599
1
1073741826
```

30. help([object])

Python has an **inbuilt help system** which you can use to see details about any **module**, **method, object, keyword, symbol,** etc.

Let’s see details about the **string object**.

Code:

```
help(str)
```

Output:

```
class str(object)
| str(object=”) -> str
| str(bytes_or_buffer[, encoding[, errors]]) -> str
|
| Create a new string object from the given object. If encoding or
| errors is specified, then the object must expose a data buffer
| that will be decoded using the given encoding and error handler.
| Otherwise, returns the result of object.__str__() (if defined)
| or repr(object).
| encoding defaults to sys.getdefaultencoding().
| errors defaults to ‘strict’.
|
| Methods defined here:
|
| __add__(self, value, /)
|      Return self+value.
|
| __contains__(self, key, /)
|      Return key in self.
|
| __eq__(self, value, /)
|      Return self==value.
|
| __format__(self, format_spec, /)
|      Return a formatted version of the string as described by format_spec.
|
| __ge__(self, value, /)
— More —
```

31. hex(x)

The **hex()** function **converts** or **returns** the **string representation** of the **hexadecimal** value of the **number**. It takes only **integer number** as an **argument**.

Code:

```
hex(123)
hex(-12)
```

Output:

```
‘0x7b’
‘-0xc’
```

32. id(object)

The **id() function** takes an **object** as an **argument** and **returns** the **identity** of the **object**. The id is **unique** and **constant** for each object.

Code:

```
id(131)
id(“Hello”)
```

Output:

```
1933365200
81300384
```

Two objects with the **same value** will have the **same identity**.

Code:

```
name = “TechVidvan”
person = “TechVidvan”
print( id(name) == id(person) )
```

Output:

```
True
```

33. input()

The Python has an **inbuilt function** for taking **input** from the **user**. The **input()** function **reads** a **string** from the **user**, which we can **store** in a **variable**.

Code:

```
msg = input(“-->”)
print(msg)
```

Output:

```
->Input as a string
‘Input as a string’
```

The function only takes a **string**, if we want an **integer value** from the user then we have to use **typecasting**.

We can achieve this by using **int() function**.

Code:

```
num = int( input(“Enter number : ” ))
type(num)
```

Output:

```
Enter number : 342
<class ‘int’ >
```

34. int([x])

The **int()** function **returns** or **converts** a **compatible number** or **string** into an **integer**. A **string** containing only **numbers** or a **float value** can easily be **converted** into **integer** using this **function**.

Code:

```
int(10.2345)
int(‘12020’)
```

Output:

```
10
12020
```

35. isinstance(object, classinfo)

The function **isinstance()** checks whether the **object argument** is an **instance** of the class given in the **second argument**, it returns a **boolean value**.

We can check this for **built-in classes** and also **user-defined classes**.

Code:

```
isinstance(“String object”, str)
isinstance( 2.5, int)

class Peep():
  msg=”Hey”
```

```
p = Peep()
isinstance( p, Peep)
```

Output:

```
True
False
True
```

36. issubclass(class, classinfo)

This function checks whether a **class (first argument)** is a **subclass** of the class in **second argument**. It will return **True** when there is a **direct** or an **indirect subclass relation** between the **classes**.

Code:

```
class A:
  pass

class B(A):
  pass

issubclass(B, A)
issubclass(A, B)
```

Output:

```
True
False
```

37. iter(object)

The **inbuilt** function **iter()** is used to **return** an **iterator object** that we can use to **iterate over the elements** in the **object**. This is mostly used in a **for loop**.

Code:

```
nums= [2,4,6,8,10,12]
for num in iter( nums):
  print(num)
```

Output:

```
2
4
6
8
10
12
```

38. len(s)

The **len()** function takes an **argument** which can be **either** a **sequence(string, list, tuple, etc)** or a **collection(dictionary, set, etc)** and returns the **number of elements** present **inside** them.

Code:

```
len([1,2,3,4,5])
len({10,20})
len(“Give me food!”)
```

Output:

```
5
2
13
```

39. list()

The **list()** function **returns** or **creates** a **new list**. It takes **iterable** like **sets**, **tuples**, etc. and **converts** them into the **list**.

Code:

```
list(“Hello”)
list({1,3,4,5,3,2})
list((“rose”, “hibiscus”, “lily”))
```

Output:

```
[‘H’, ‘e’, ‘l’, ‘l’, ‘o’]
[1, 2, 3, 4, 5]
[‘rose’, ‘hibiscus’, ‘lily’]
```

40. locals()

The **locals()** in-built method is similar to the **globals()** method which we saw earlier. It **returns** a **dictionary** of the **current local symbol table**.

Code:

```
locals()
```

Output:

```
{‘__name__’: ‘__main__’, ‘__doc__’: None, ‘__package__’: None, ‘__loader__’: <class ‘_frozen_importlib.BuiltinImporter’>,
‘__spec__’: None, ‘__annotations__’: {}, ‘__builtins__’: <module ‘builtins’ (built-in)>, ‘var1’: 3, ‘var2’: 3, ‘name’:
‘Shrangi’, ‘person’: ‘Shrangi’, ‘msg’: ‘Input as a string’, ‘num’: 342, ‘Car’: <class ‘__main__.Car’>, ‘c’: <__main__.Car object
at 0x04D891C0>, ‘A’: <class ‘__main__.A’>, ‘B’: <class ‘__main__.B’>}
```

41. map(function, iterable)

The **map()** function is used to **map each element** of an **iterable element** to a **function**. It is similar to the **filter()** method that we saw before. It is **useful** in **modifying** each **element** of an iterable according to a **function**.

Code:

```
list(map(lambda x:x+10, [1,2,3,4,5] ))
```

Output:

```
[11, 12, 13, 14, 15]
```

42. max(iterable)

The **max()** function is **self-explanatory**, it takes an **iterable container** or **sequence** as an argument and returns the **maximum value** from the **list**.

Code:

```
max([1,3,5,7,123, 435,-2678,65])

max( {-20, 80, 20, 30} )
```

Output:

```
435
80
```

43. memoryview(object)

The **memoryview()** function takes a **bytes object** as **argument** and **returns** a view of the **memory** and it’s a **safe** way to expose **buffer protocol**.

Code:

```
var = bytes(6)
memoryview(var)
```

Output:

```
<memory at 0x04C60328>
```

44. min(iterable)

The **min() function** is also similar to the **max() functions**. It returns the **minimum value** from a **group** of items in an **iterable**.

Code:

```
min([7,4,2,1])

min({-6, -10, 20, 30})
```

Output:

```
1
-10
```

45. next(iterator)

The **next() function** is used to get the **next item** from the **iterator object**. Every time we call the **next() method** the iterator points to the **next element**.

When there are **no next element** present, then the function raises a **StopIteration error**.

Code:

```
myIterator = iter([10,20,30])
next(myIterator)
next(myIterator)
next(myIterator)
next(myIterator)
```

Output:

```
10
20
30
Traceback (most recent call last):
  File “<stdin>”, line 1, in <module>
StopIteration
```

46. object()

The **object() method** does not take any arguments and it **returns** a **featureless object**. It is the **base** for all **classes** and it contains methods that are **common** to all the Python objects.

Code:

```
obj = object()
type(obj)
dir(obj)
```

Output:

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

47. oct(x)

The **oct()** function **converts** or **returns** an **octal representation** of a **number**.

Octal numbers are **prefixed** with **“0o”**. It only takes an **integer value** and **returns** its **octal value**.

Code:

```
oct(10)
oct(-200)
```

Output:

```
‘0o12’
‘-0o310’
```

48. open(file, mode=’r’)

The **open()** function is used in **working** with **files**. It can open **any file**.

The **first argument** is the **file path** and the **second argument** is the **mode** by which we open the file, **for example**, **read**, **write**, **append**, etc. we use characters **‘r’**, **‘w’** and **‘a’** respectively to **represent** these **modes**.

The default mode is **read mode**.

Code:

```
f=open('E:\\techvidvan/test.txt')
print(f)
```

Output:

```
<_io.TextIOWrapper name=’E:\\techvidvan/test.txt’ mode=’r’ encoding=’cp1252'>
```

To read the contents of the file we use the **read() method** on the file.

Code:

```
contents = f.read()
print(contents)
```

Output:

```
Hello World!
```

49. ord(c)

The **ord()** method takes a **Unicode character** as an **argument** and **returns** an **integer representation** of the **character**. It is the **opposite** of the **chr()** function.

Code:

```
ord(‘a’)
ord(‘$’)
ord(‘9’)
```

Output:

```
97
36
57
```

50. pow(base, exp)

The **pow()** function is used for **calculating** the **mathematical power** of a number. This function returns the **base** to the **power of exp**.

For example **pow(a,b)** will return a to the **power of b**.

Code:

```
pow(2, 4)
pow(5.5, 2 )
pow(8,-1)
```

Output:

```
16
30.25
0.125
```

51. print(*objects, sep=” ”, end=”\n”)

You have already used this function thousands of times.

The **print()** function **prints** the objects to the **text stream file**. It **separates** the object by **space** by **default** and in the end, it appends a **newline** by **default**. We can change this by specifying **different arguments**.

Code:

```
print(“Hello!”)
print(1,2,3,4, sep=”-”)
print(“$$”, end=””);
print(“@@”)
```

Output:

```
Hello!
1-2-3-4
$$@@
```

52. property()

The **property() method** is used to **return a property** attribute from the given **getter**, **setter**, or **deleter**.

The **syntax** of property() method is –

Syntax:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

- The fget is a function used to get attribute value.
- The fset is a function for setting an attribute value.
- The fdel is a function for deleting an attribute.
- The doc is a string used for docstrings.

53. range(start, stop, step)

The **range() function** is used to **generate** a **sequence of numbers** from a **starting range** to the **stop number**. It is useful to **iterate** over a **range of elements**.

Code:

```
for i in range(5,10):
  print(i)
```

Output:

```
5
6
7
8
9
```

54. repr()

The **repr()** function is used to **return** a **printable version** of the **Python objects**.

Code:

```
repr(“Hey”)
a = 5.5
repr(a)
repr({1,2,3,4})
```

Output:

```
“‘Hey’”
‘5.5’
‘{1, 2, 3, 4}’
```

55. reversed(seq)

The **reversed()** function takes a **sequence** as an **argument** and **returns** a **reverse iterator** to the **sequence**. It is used when we want to iterate the **elements backward**.

Code:

```
for i in reversed([1,2,4,6,8]):
  print(i)
```

Output:

```
8
6
4
2
1
```

56. round(numbers [,digits])

The **round()** function round offs a number to specified **n-digits.** If the **digits** are **not specified** then it round offs to a **natural number**.

Code:

```
round(3.5)
round(3.2)
round(1.666666, 2 )
```

Output:

```
4
3
1.67
```

57. set([iterable])

Set is a **built-in** class in **Python**.

The set() function takes an **iterable** as an **argument** and **returns** a set object of that **iterable**.

Code:

```
set([1,3,3,5,6,5])
set((10,20,50,20))
```

Output:

```
{1, 3, 5, 6}
{10,20,50}
```

58. setattr(object, name, value)

We have seen **getattr()** and **hasattr()**.

Now the **setattr()** function is used to set a **value** of an **attribute**. We can set a **new attribute** or **update** an attribute if the class **allows** us to **modify**.

Code:

```
class Student:
  pass

s= Student()
setattr(s, “name”, “Rambo”)
s.name
```

Output:

```
Rambo
```

59. slice(start, stop [,step])

The **slice()** function returns a **slice object** just like a **range**. We can use the slice object to slice a **sequence** like **lists**, **strings**, etc.

Code:

```
s1= slice(4)
s2= slice(1,6,2)
print(s1)
print(s2)

print(“123456789”[s1])
print(“123456789”[s2])
```

Output:

```
slice(None, 4, None)
slice(1,6,2)
1234
246
```

60. sorted(iterable)

The **sorted()** function sorts the given **iterable** and **returns** a list of all the elements in **ascending order** by **default**. It will sort a **list**, **string**, **sets**, etc and will always return a list.

Code:

```
sorted([7,5,3,2,1])
sorted(“Hello”)
sorted({1,2,3,4,5}, reverse=True)
```

Output:

```
[1, 2, 3, 5, 7]
[‘H’, ‘e’, ‘l’, ‘l’, ‘o’]
[5, 4, 3, 2, 1]
```

61. @staticmethod()

This is a decorator which is used to **transform** a method into a **static method**. A static method can be **directly called** with the **class name** without **creating any instance**.

Code:

```
class Letter:
    @staticmethod
    def msg():
        print("static method")

Letter.msg()
```

Output:

```
Static method
```

62. str(object)

The **str() function** is used to **convert** an **object** into a **string**. **str** is the **built-in** class for **strings**. It can be used in **type conversion** of numbers into **strings**.

Code:

```
str()
str(125)
str(“Hello”)
str({1,10,60})
```

Output:

```
‘’
‘125’
‘Hello’
‘{1, 10, 60}’
```

63. sum(iterable)

The function **sum()** is also **self-explanatory**. It takes an **iterable collection** or **sequence** as an **argument** and **returns** the **sum of all the elements**.

The elements should be **only numbers** else it will not be able to **add elements** and **throw errors**.

Code:

```
sum([1,2,3,4,5])
sum((10,30,10))
sum([1, 4.5, 8.6, 100])
```

Output:

```
15
50
114.1
```

64. super()

The **super()** method is used to **return** a **proxy object** that refers to the **parent class**. By using the super() method we can access the **parent class methods** or **attributes**.

Code:

```
class A:
    def __init__(self):
        print("Class A")
```

```

class B(A):
    def __init__(self):
        super().__init__()
        print("Class B")

b = B()
```

Output:

```
Class A
Class B
```

65. tuple([iterable])

Tuple is an **immutable sequence** of **elements**. The **tuple()** function is used to **create** or **convert** other sequences like **lists**, **strings,** etc into tuples.

Code:

```
tuple([1,2,3,4,5])
tuple(“Techvidvan”)
```

Output:

```
(1, 2, 3, 4, 5)
(‘T’, ‘e’, ‘c’, ‘h’, ‘v’, ‘i’, ‘d’, ‘v’, ‘a’, ‘n’)
```

66. type()

The **type()** function **returns** the **type** of the **Python objects** or the **class** of the **Python objects**.

Code:

```
type(“Tech”)
type(3.5)
type([])
type({1,2,3})
```

Output:

```
<class ‘str’>
<class ‘float’>
<class ‘list’>
<class ‘set’>
```

67. vars([object])

The **vars()** function returns the **__dict__ attribute** of a **module**, **class**, **instance**, or any **Python object.** If arguments are **not passed** then it is similar to the **locals() function**.

Code:

```
vars(tuple)
```

Output:

```
mappingproxy({‘__repr__’: <slot wrapper ‘__repr__’ of ‘tuple’ objects>, ‘__hash__’: <slot wrapper ‘__hash__’ of ‘tuple’
objects>, ‘__getattribute__’: <slot wrapper ‘__getattribute__’ of ‘tuple’ objects>, ‘__lt__’: <slot wrapper ‘__lt__’ of ‘tuple’
objects>, ‘__le__’: <slot wrapper ‘__le__’ of ‘tuple’ objects>, ‘__eq__’: <slot wrapper ‘__eq__’ of ‘tuple’ objects>, ‘__ne__’:
<slot wrapper ‘__ne__’ of ‘tuple’ objects>, ‘__gt__’: <slot wrapper ‘__gt__’ of ‘tuple’ objects>, ‘__ge__’: <slot wrapper
‘__ge__’ of ‘tuple’ objects>, ‘__iter__’: <slot wrapper ‘__iter__’ of ‘tuple’ objects>, ‘__len__’: <slot wrapper ‘__len__’ of
‘tuple’ objects>, ‘__getitem__’: <slot wrapper ‘__getitem__’ of ‘tuple’ objects>, ‘__add__’: <slot wrapper ‘__add__’ of ‘tuple’
objects>, ‘__mul__’: <slot wrapper ‘__mul__’ of ‘tuple’ objects>, ‘__rmul__’: <slot wrapper ‘__rmul__’ of ‘tuple’ objects>,
‘__contains__’: <slot wrapper ‘__contains__’ of ‘tuple’ objects>, ‘__new__’: <method ‘__getnewargs__’ of ‘tuple’ objects><built-
in method __new__ of type object at 0x733B8588>, ‘__getnewargs__’: , ‘index’: <method ‘index’ of ‘tuple’ objects>, ‘count’:
<method ‘count’ of ‘tuple’ objects>, ‘__doc__’: “Built-in immutable sequence.\n\nIf no argument is given, the constructor
returns an empty tuple.\nIf iterable is specified the tuple is initialized from iterable’s items.\n\nIf the argument is a tuple,
the return value is the same object.”})
```

68. zip(*iterables)

The **zip function** returns us **iterators** of **tuples**. It can take any number of **iterables** and packs their **same index positions** into **tuples**.

Code:

```python
for i in zip([10,20,30,40],[1,2,3],[1,2,3,4,5]):
  print(i)
```

Output:

```
(10, 1, 1)
(20, 2, 2)
(30, 3, 3)
```

69. __import__(name)

This is an **advanced function** that is not used in everyday programming.

Whenever we use an import statement like – **import numpy, i**t calls **the __import__() function** automatically which **imports** the **statements**.

Syntax:

```python
__import__(name, globals, locals, fromlist, level)
```

Code:

```python
__import__('math', globals(), locals(), [], 0)
```

This statement is equivalent to **'import math'**. This function is useful when we want to **import** a **module** during **runtime**.

Summary

This article is a bit lengthy and you should congratulate yourself for making this far.

We have discussed all the **69 Python built-in functions**. The built-in functions are available to use anywhere in the **Python programme**.

You can use TechVidvan's Python built-in functions article as a reference when you want to quickly grasp information about a function.

Tags: @classmethod() bin(x) bool([x]) Built-In functions in Python dict() eval() python built in functions list Python Built-In functions

LEAVE A REPLY

Enter your comment here...