**Introduction to Python Functions**

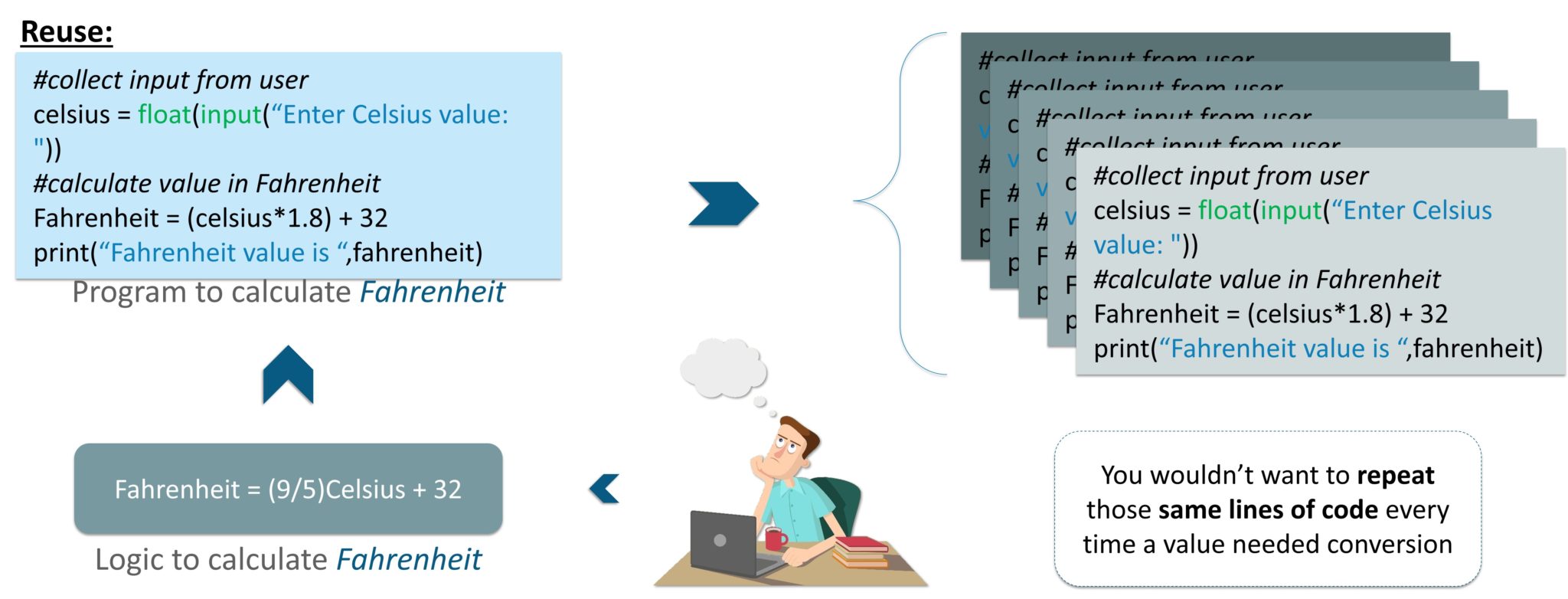
In today’s fast-paced IT world, it is always an advantage to have an edge over the others in terms of in-depth knowledge of a certain technology. Python is a widely used language and provides ‘n’ number of opportunities to enthusiastic learners. Learning to use the functions in Python in the right way is a notable skill for any **Python Developer**.

Why We Need Python Functions?

Functions manage the inputs and outputs in computer programs. Programming languages are designed to work on data and functions are an effective way to manage and transform this data.

The modifications are generally done to drive outcomes like performing tasks and finding results. And, the set of operations or instructions required to do so comes from logically functional blocks of code that can be reused independently from the main program.

In fact, the main code is also a function, just a very important one at that. Every other function is logically aligned and maintained to functionally execute from your main code. But, if the function has not been defined previously, you’ll just have to define one yourself before using it. This is because the definition lists the steps of its operation.



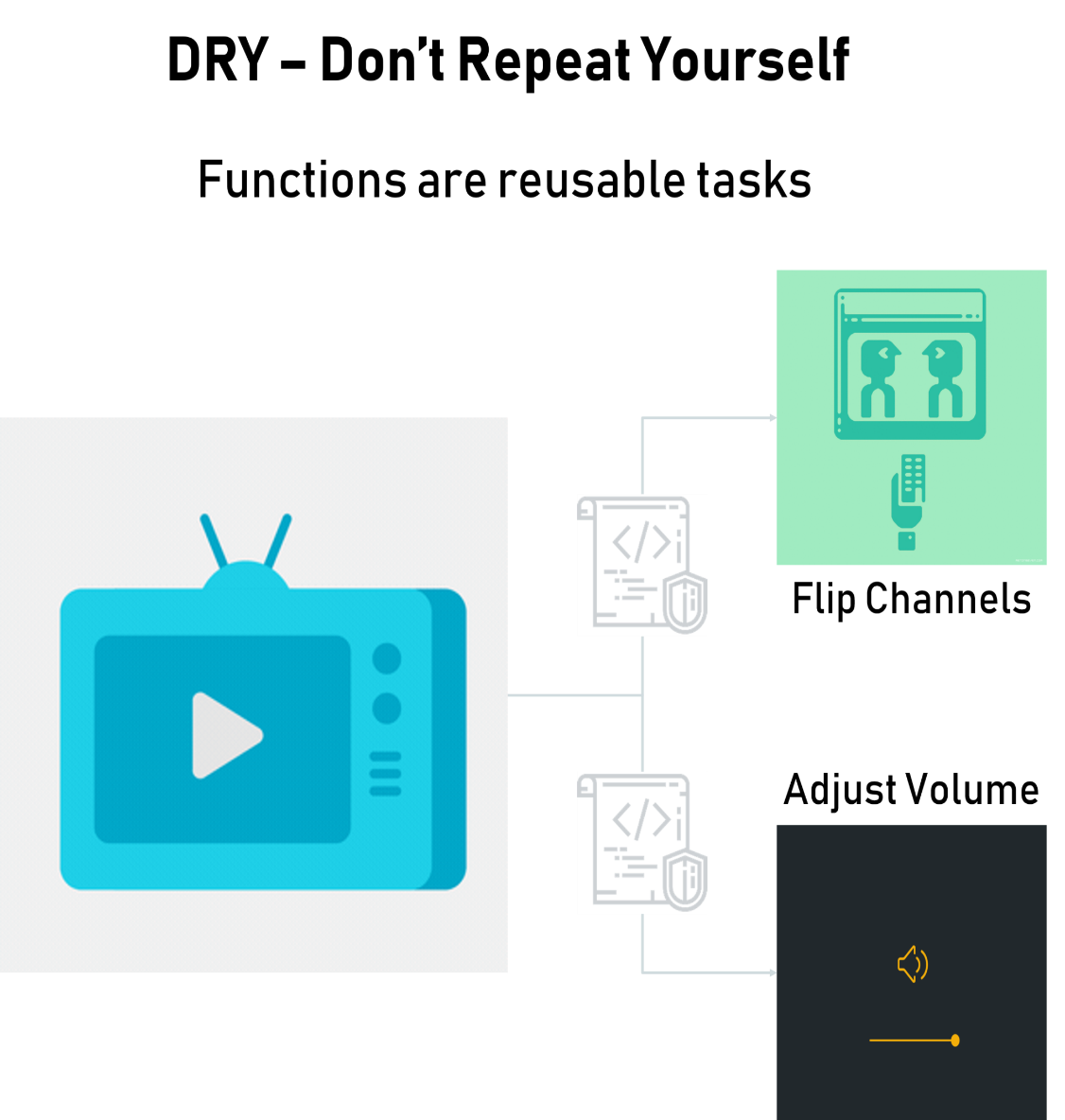
Would you rather write a single piece of code 10 times or just once and use it 10 times?

So, functions are nothing but tasks that a user wants to perform. But, defining it once with a name will let you reuse that functionality without making your main programs look too scary. This drastically reduces lines of code and even make debugging easier.

We’ll be getting into that shortly, but the first reason behind why use a function is because of its reusability. The fact that even complex operations could be put together as singular tasks that would run with just a call by its name is what has made computer codes of today so much clearer as well.

Every programming language lets you create and use these functions to perform various tasks with just a call. And, you could call it any number of times without having to worry about logically structuring its code into your main code every single time.

Let’s try and understand their need to us through a simple example first.



Say, you have a television that stores many channels on it, receives their digital radio broadcasts, converts them into what we watch, while also giving us additional options for a variety of other features as well.

But that doesn’t mean there is someone logically scripting the lines of codes for what you watch every time you turn on your tv or flip a channel. Rather, functions for each task in its working have been logically defined once and keep getting reused time and again according to the features you try to use.

All of it happens by calling its different functions as many times as needed from the main function that is running. So, even if you’re turning the volume up or down, its defined function is being called repeatedly.

And, having a system operating the main code to keep calling these functions as and when needed has also made designing and innovating upon it all the easier.

The important thing to note is that whenever this function is called, it executes its tasks depending on the instructions specified in it.

That’s how machines can have different functions. A calculator is probably the most common example of this. It has the provision of addition, subtraction, multiplication, division, and other functions. All its functions have been clearly predefined into it, but it only performs those that you choose to call by pressing its respective button.

Programmers reduce coding time and debugging time, thereby reducing overall development time, by using functions.

Next up on this Python Functions blog, let us look at what the Python Functions actually are.

## ****What Are Python Functions?****

The functions in Python are a classic example of such **reusability**. So, to serve a wide range of applications from GUI and mathematical computing to web development and testing, Python’s interpreter already comes equipped with numerous functions that are always available for use. And, you could also bring in other libraries or modules to your program that contain pre-defined functions readily available for use.

All you’ll really have to do is download required packages that according to their documentation and freely avail all its useful functionalities by just importing them over to your code.

So, once defined, a function can be used any number of times at any point in any of your codes. Now, this is because Python falls in-line with the DRY principle of software engineering, which aims to replace any repetition of software patterns or codes with abstractions to avoid redundancy and ensure that they can be used freely without revealing any inner details on their implementations.

**DRY expands to Don’t Repeat Yourself and** this concept of having re-usable blocks of codes is very crucial for achieving abstraction in Python. Thus, in order to use a function all that you’ll really need is its name, its purpose, its arguments if it takes any and its result’s type if it returns any.

It’s almost like using an automobile or a telephone, where you don’t necessarily need to understand the working of its components to use them. Rather, they’ve been already built to serve common purposes that you can just use directly to achieve your goals and devote your precious time to implementing all the innovative aspects of your application program. And, nobody really wants to know how a function in your program works on the inside, as long as it does the job.

So, with Python, unless you must write a new function or change how an existing one works, you won’t even need to understand anything about what goes on inside till it works the way you need it to. It’s just like with a vehicle or a phone again, where you’ll need to know how it works in order to build or fix one. And, similarly, once you’ve written a working function, you can use it repeatedly without having to look at the contents inside it ever again.

A function can be called as a section of a program that is written once and can be executed whenever required in the program, thus making code reusability.

The function is a subprogram that works on data and produces some output.

 To define a Python function, you’d have to use the ‘**def**’ keyword before the name of your function and add parentheses to its end, followed by a colon ():

Python uses indentation to indicate blocks instead of brackets to make codes more readable.

A function in Python may contain any number of parameters or none. So, for times when you need your function to operate on variables from other blocks of code or from your main program, it could take any number of parameters and produce results.

 A Python function could also optionally return a value. This value could be a result produced from your function’s execution or even be an expression or value that you specify after the keyword ‘return’. And, after a return statement is executed, the program flow goes back to the state next to your function call and gets executed from there.

So, to call a Python function at any place in your code, you’ll only have to use its name and pass arguments in its parentheses, if any.

The **rules** for naming a function are the same as naming a variable. It begins with either letter from A-Z, a-z in both upper & lower cases or an underscore(\_). The rest of its name can contain underscores(\_), digits(0-9), any letters in upper or lower case.

1. A reserved keyword may not be chosen as an identifier.
2. Good usage of grammar to ensure enhanced readability of code.

It is good practice to name a Python function according to what it does. use a docstring right under the first line of a function declaration. This is a documentation string, and it explains what the function does.

Next up in this Python Functions Section, let us check out the types of Functions available in Python.

## ****Types of Python Functions****

There are many types of Python Functions. And each of them is very vital in its own way. The following are the different types of Python Functions:

* Python Built-in Functions
* Python Recursion Functions
* Python Lambda Functions
* Python User-defined Functions

Let us check out these functions in detail. Beginning with Built-in functions as they are very easy to understand and implement.

## ****Python Built-in Functions:****

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, **print()** function prints the given object to the standard output device (screen) or to the text stream file.

In Python 3.6, there are 68 built-in functions. But for the sake of simplicity let us consider the majorly used functions and we can build on from there.

### **Python abs() Function:**

**Definition**

The abs() method returns the absolute value of the given number. If the number is a complex number, abs() returns its magnitude.

**Syntax**

The syntax of abs() method is:

**abs(num)**

**Parameters**

 The abs() method takes a single argument:

* **num** – A number whose absolute value is to be returned. The number can be:
  1. integer
  2. floating number
  3. complex number

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | # random integer  integer = -20  print('Absolute value of -20 is:', abs(integer))    #random floating number  floating = -30.33  print('Absolute value of -30.33 is:', abs(floating)) |

**Output**

Absolute value of -20 is: 20 Absolute value of -30.33 is: 30.33

### **Python all() Function:**

**Definition**

The all() method returns True when all elements in the given iterable are true. If not, it returns False.

**Syntax**

The syntax of all() method is:

all(**iterable**)

**Parameters**

The all() method takes a single parameter:

* **iterable** – Any iterable (list, tuple, dictionary, etc.) which contains the elements

**Example**

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

**Output**

True

False

False

False

True

### **Python ascii() Function:**

**Definition**

The ascii() method returns a string containing a printable representation of an object. It escapes the non-ASCII characters in the string using x, u or U escapes.

**Syntax**

The syntax of ascii() method is:

ascii(object)

**Parameters**

The ascii() method takes an object (like strings, list etc).

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | normalText = 'Python is interesting'  print(ascii(normalText))    otherText = 'Pythön is interesting'  print(ascii(otherText))    print('Pythn is interesting') |

**Output**

'Python is interesting'

Pythön is interesting

'Pythn is interesting'

### **Python bin() Function:**

**Definition**

The bin() method converts and returns the binary equivalent string of a given integer. If the parameter isn’t an integer, it has to implement \_\_index\_\_() method to return an integer.

**Syntax**

The syntax of bin() method is:

bin(num)

**Parameters**

The bin() method takes a single parameter:

* **num** – an integer number whose binary equivalent is to be calculated.  
  If not an integer, should implement \_\_index\_\_() method to return an integer.

**Example**

|  |  |
| --- | --- |
| 1  2 | number = 5  print('The binary equivalent of 5 is:', bin(number)) |

**Output**

The binary equivalent of 5 is: 0b101

### **Python bool() Function:**

**Definition**

The bool() method converts and returns the binary equivalent string of a given integer. If the parameter isn’t an integer, it has to implement \_\_index\_\_() method to return an integer.

**Syntax**

The syntax of **bool()** method is:

bool([value])

**Parameters**

It’s not mandatory to pass a value to bool(). If you do not pass a value, bool() returns False.

In general use, bool() takes a single parameter value.

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | test = []  print(test,'is',bool(test))    test = [0]  print(test,'is',bool(test))    test = 0.0  print(test,'is',bool(test))    test = None  print(test,'is',bool(test))    test = True  print(test,'is',bool(test))    test = 'Easy string'  print(test,'is',bool(test)) |

**Output**

[] is False

[0] is True

0.0 is False

None is False

True is True

Easy string is True

### **Python compile() Function:**

**Definition**

The compile() method returns a Python code object from the source (normal string, a byte string, or an AST object).

**Syntax**

The syntax of compile() method is:

compile(source, filename, mode, flags=0, dont\_inherit=False, optimize=-1)

**Parameters**

* source – a normal string, a byte string, or an AST object
* filename – file from which the code was read. If it wasn’t read from a file, you can give a name yourself
* mode – Either exec or eval or single.
  + eval – accepts only a single expression.
  + exec – It can take a code block that has Python statements, class and functions and so on.
  + single – if it consists of a single interactive statement
* flags (optional) and dont\_inherit (optional) – controls which future statements affect the compilation of the source. Default Value: 0
* optimize (optional) – optimization level of the compiler. Default value -1.
* **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | codeInString = ''  codeObejct = compile(codeInString, 'sumstring', 'exec')   a = 5/n b=6/n sum=a+b /n print("sum =",sum)  exec(codeObejct) |

* **Output**
* sum = 11

### **Python dict() Function:**

**Definition**

The dict() constructor creates a dictionary in Python.

**Syntax**

Different forms of dict() constructors are:

class dict(\*\*kwarg)

class dict(mapping, \*\*kwarg)

class dict(iterable, \*\*kwarg)

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | numbers = dict(x=5, y=0)  print('numbers = ',numbers)  print(type(numbers))    empty = dict()  print('empty = ',empty)  print(type(empty)) |

**Output**

numbers = {'x': 5, 'y': 0}

<class 'dict'>

empty = {}<class 'dict'>

### **Python enumerate() Function:**

**Definition**

The enumerate() method adds counter to an iterable and returns it (the enumerate object).

**Syntax**

The syntax of enumerate() method is:

enumerate(iterable, start=0)

**Parameters**

The enumerate() method takes two parameters:

* **iterable** – a sequence, an iterator, or objects that support iteration
* **start** (optional) – enumerate() starts counting from this number. If *start* is omitted, 0 is taken as the start.
* **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | grocery = ['bread', 'milk', 'butter']  enumerateGrocery = enumerate(grocery)    print(type(enumerateGrocery))    # converting to list  print(list(enumerateGrocery))    # changing the default counter   * enumerateGrocery = enumerate(grocery, 10)   print(list(enumerateGrocery)) |

* **Output**
* <class 'enumerate'>
* [(0, 'bread'), (1, 'milk'), (2, 'butter')]
* [(10, 'bread'), (11, 'milk'), (12, 'butter')]

### **Python eval() Function:**

**#Check !**

**Definition**

The eval() method parses the expression passed to this method and runs python expression (code) within the program.

**Syntax**

The syntax of eval() method is:

eval(expression, globals=None, locals=None)

**Parameters**

The eval() takes three parameters:

* **expression** – this string is parsed and evaluated as a Python expression
* **globals** (optional) – a dictionary
* **locals**(optional)- a mapping object. Dictionary is the standard and commonly used mapping type in Python.

**Example**

|  |  |
| --- | --- |
| 1  2 | x = 1  print(eval('x + 1')) |

**Output**

sum = 11

### **Python filter() Function:**

**Definition**

The filter() method constructs an iterator from elements of an iterable for which a function returns true.

**Syntax**

The syntax of filter() method is:

filter(function, iterable)

**Parameters**

The filter() method takes two parameters:

* **function** – function that tests if elements of an iterable return true or false  
  If None, the function defaults to Identity function – which returns false if any elements are false
* **iterable** – iterable which is to be filtered, could be sets, lists, tuples, or containers of any iterators

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | # list of alphabets  alphabets = ['a', 'b', 'd', 'e', 'i', 'j', 'o']    # function that filters vowels  def filterVowels(alphabet):      vowels = ['a', 'e', 'i', 'o', 'u']        if(alphabet in vowels):          return True      else:          return False    filteredVowels = filter(filterVowels, alphabets)    print('The filtered vowels are:')  for vowel in filteredVowels:      print(vowel) |

**Output**

The filtered vowels are:

a

e

i

o

### **Python getattr() Function:**

**Definition**

The getattr() method returns the value of the named attribute of an object. If not found, it returns the default value provided to the function.

**Syntax**

The syntax of getattr() method is:

getattr(object, name[, default])

**Parameters**

The getattr() method takes multiple parameters:

* **object** – object whose named attribute’s value is to be returned
* **name** – string that contains the attribute’s name
* **default (Optional)** – value that is returned when the named attribute is not found

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Person:      age = 23      name = "Adam"    person = Person()  print('The age is:', getattr(person, "age"))  print('The age is:', person.age) |

class Person:

    age = 23

    name = "Adam"

person = Person()

print('The age is:', getattr(person, "age")) # Using getattr()

print('The age is:', person.age)        # without using  getattr()

print('The name is:', person.name)     # without using  getattr()

**Output**

The age is: 23

The age is: 23

The name is: Adam

### **Python help() Function:**

**Definition**

The help() method calls the built-in Python help system.

**Syntax**

The syntax of help() method is:

help(object)

**Parameters**

The help() method takes the maximum of one parameter.

* **object** (optional) – you want to generate the help of the given object

**Example**

>>> help('print')

help('print')

help('tuple')

### **Python id() Function:**

As we can see the function accepts a single parameter and is used to return the identity of an object. **This identity has to be unique and constant for this object during the lifetime.** Two objects with non-overlapping lifetimes may have the same id() value. If we relate this to C, then they are actually the memory address, here in Python it is the unique id. This function is generally used internally in Python.

**Definition**

The id() function returns identity (unique integer) of an object.

**Syntax**

The syntax of id() method is:

id(object)

**Parameters**

The id() function takes a single parameter object.

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Foo:      b = 5    dummyFoo = Foo()  print('id of dummyFoo =',id(dummyFoo)) |

**Output**

id of dummyFoo = 140343867415240

### **Python len() Function:**

**Definition**

The len() function returns the number of items (length) in an object.

**Syntax**

The syntax of len() method is:

len(s)

**Parameters**

**s** – a sequence (string, bytes, tuple, list, or range) or a collection (dictionary, set or frozen set)

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | testList = []  print(testList, 'length is', len(testList))    testList = [1, 2, 3]  print(testList, 'length is', len(testList))    testTuple = (1, 2, 3)  print(testTuple, 'length is', len(testTuple))    testRange = range(1, 10)  print('Length of', testRange, 'is', len(testRange)) |

**Output**

[] length is 0

[1, 2, 3] length is 3

(1, 2, 3) length is 3

Length of range(1, 10) is 9

### **Python max() Function:**

**Definition**

The max() method returns the largest element in an iterable or largest of two or more parameters.

**Syntax**

The syntax of max() method is:

max(iterable, \*iterables[,key, default])

max(arg1, arg2, \*args[, key])

**Parameters**

max() has two forms of arguments it can work with.

1. max(iterable, \*iterables[, key, default])
   * **iterable** – sequence (tuple, string), collection (set, dictionary) or an iterator object whose largest element is to be found
   * **\*iterables (Optional)** – any number of iterables whose largest is to be found
   * **key (Optional)** – key function where the iterables are passed and the comparison is performed based on its return value
   * **default (Optional)** – default value if the given iterable is empty
2. max(arg1, arg2, \*args[, key])
   * **arg1** – mandatory first object for comparison (could be number, string or other object)
   * **arg2** – mandatory second object for comparison (could be number, string or another object)
   * **\*args (Optional)** – other objects for comparison
   * **key** – key function where each argument is passed and the comparison is performed based on its return value

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # using max(arg1, arg2, \*args)  print('Maximum is:', max(1, 3, 2, 5, 4))    # using max(iterable)  num = [1, 3, 2, 8, 5, 10, 6]  print('Maximum is:', max(num)) |

**Output**

Maximum is: 5

Maximum is: 10

### **Python min() Function:**

**Definition**

The min() method returns the smallest element in an iterable or smallest of two or more parameters.

The **syntax** of min() method is:

min(iterable, \*iterables[,key, default])

min(arg1, arg2, \*args[, key])

**Parameters**

min() has two forms of arguments it can work with.

1. min(iterable, \*iterables[, key, default])
   * **iterable** – sequence (tuple, string), collection (set, dictionary) or an iterator object whose smallest element is to be found
   * **\*iterables (Optional)** – any number of iterables whose smallest is to be found
   * **key (Optional)** – key function where the iterables are passed and the comparison is performed based on its return value
   * **default (Optional)** – default value if the given iterable is empty
2. min(arg1, arg2, \*args[, key])
   * **arg1** – mandatory first object for comparison (could be number, string or other object)
   * **arg2** – mandatory second object for comparison (could be number, string or other object)
   * **\*args (Optional)** – other objects for comparison
   * **key** – key function where each argument is passed and a comparison is performed based on its return value
3. **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # using min(arg1, arg2, \*args)  print('Minimum is:', min(1, 3, 2, 5, 4, 500))    # using min(iterable)  num = [3, 2, 8, 5, 10, 6, 1, 1000]  print('Minimum is:', min(num)) |

1. **Output**
2. Minimum is: 1
3. Minimum is: 2

### **Python oct() Function:**

**Definition**

The oct() method takes an integer number and returns its octal representation. If the given number is an int, it must implement \_\_index\_\_() method to return an integer.

The **syntax** of oct() method is:

oct(x)

**Parameters**

The oct() method takes a single parameter **x.**

This parameter could be:

* an integer number (binary, decimal or hexadecimal)
* if not an integer, must implement \_\_index\_\_() method to return an integer

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # decimal number  print('oct(10) is:', oct(10))    # binary number  print('oct(0b101) is:', oct(0b101))    # hexadecimal number  print('oct(0XA) is:', oct(0XA)) |

**Output**

oct(10) is: 0o12

oct(0b101) is: 0o5

oct(0XA) is: 0o12

### **Python pow() Function:**

**Definition**

The pow() method 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. pow(x, y) % z.

The **syntax** of pow() method is:

pow(x, y[, z])

**Parameters**

The pow() method takes three parameters:

* **x** – number which is to be powered
* **y** – number which is to be powered with x
* **z (Optional)** – number which is to be used for modulus operation

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # positive x, positive y (x\*\*y)  print(pow(2, 2))    # negative x, positive y  print(pow(-2, 2))    # positive x, negative y (x\*\*-y)  print(pow(2, -2))    # negative x, negative y  print(pow(-2, -2)) |

**Output**

4

4

0.25

0.25

### **Python reversed() Function:**

**Definition**

The reversed() method returns the reversed iterator of the given sequence.

The syntax of reversed() method is:

reversed(seq)

**Parameters**

The reversed() method takes a single parameter:

* **seq** – sequence that should be reversed  
  Could be an object that supports sequence protocol (\_\_len\_\_() and \_\_getitem\_\_() methods) as tuple, string, list or range  
  Could be an object that has implemented \_\_reversed\_\_()
* **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | # for string  seqString = 'Python'  print(list(reversed(seqString)))    # for tuple  seqTuple = ('P', 'y', 't', 'h', 'o', 'n')  print(list(reversed(seqTuple)))    # for range  seqRange = range(5, 9)  print(list(reversed(seqRange)))    # for list  seqList = [1, 2, 4, 3, 5]  print(list(reversed(seqList))) |

* **Output**
* ['n', 'o', 'h', 't', 'y', 'P']
* ['n', 'o', 'h', 't', 'y', 'P']
* [8, 7, 6, 5]
* [5, 3, 4, 2, 1]

### **Python sum() Function:**

**Definition**

The sum() method returns the reversed iterator of the given sequence.

The syntax of sum() method is:

sum(iterable, start)

**Parameters**

* **iterable** – iterable (list, tuple, dict etc) whose item’s sum is to be found. Normally, items of the iterable should be numbers.
* **start** (optional) – this value is added to the sum of items of the iterable. The default value of *start* is 0 (if omitted)

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | numbers = [2.5, 3, 4, -5]    # start parameter is not provided  numbersSum = sum(numbers)  print(numbersSum)    # start = 10  numbersSum = sum(numbers, 10)  print(numbersSum) |

**Output**

4.5

14.5

### **Python type() Function:**

**Definition**

If a single argument (object) is passed to type() built-in, it returns the type of the given object. If three arguments (name, bases, and dict) are passed, it returns a new type object.

The syntax of type() method is:

type(object)

type(name, bases, dict)

**Parameters**

* If the single *object* argument is passed to type(), it returns the type of the given object.

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | numberList = [1, 2]  print(type(numberList))    numberDict = {1: 'one', 2: 'two'}  print(type(numberDict))    class Foo:      a = 0    InstanceOfFoo = Foo()  print(type(InstanceOfFoo)) |

**Output**

<class 'dict'>

<class 'Foo'>

Next up on this Python Functions blog, let us check out the Recursive Function in Python

## ****Python Recursive Functions****

### **What is recursion in Python?**

Recursion is the process of defining something in terms of itself.

Python also accepts function recursion, which means a defined function can call itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

### **Python Recursive Function**

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Following is an example of a recursive function to find the factorial of an integer.

In this example, **tri\_recursion()** is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

**Example1:**

def **tri\_recursion**(k):  
  if(k>0):  
    result = k+tri\_recursion(k-1)  
    print(result)  
  else:  
    result = 0  
  return result  
  
print("\n\nRecursion Example Results")  
**tri\_recursion**(6)

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 5 (denoted as 5!) is 1\*2\*3\*4\*5 = 120.

**Example2:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # An example of a recursive function to  # find the factorial of a number    def calc\_factorial(x):      <em>"""This is a recursive function      to find the factorial of an integer"""        </em>if x == 1:          return 1      else:          return (x \* calc\_factorial(x-1))    num = 4  print("The factorial of", num, "is", calc\_factorial(num)) |

In the above example, **calc\_factorial()** is a recursive function as it calls itself.

1. # Factorial of a number using recursion
2. def recur\_factorial(n):
3. if n == 1:
4. return n
5. else:
6. return n\*recur\_factorial(n-1)
7. num = 7
8. # check if the number is negative
9. if num < 0:
10. print("Sorry, factorial does not exist for negative numbers")
11. elif num == 0:
12. print("The factorial of 0 is 1")
13. else:
14. print("The factorial of", num, "is", recur\_factorial(num))

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function call multiples the number with the factorial of number 1 until the number is equal to one.

Our recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

### **Advantages of Recursion**

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

### **Disadvantages of Recursion**

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

## ****Python Lambda Functions****

### **What Are Lambda functions?**

In Python, an anonymous function is a function that is defined without a name.

While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.

Hence, anonymous functions are also called lambda functions.

### **How To Use Lambda Functions In Python?**

A Lambda function in python has the following syntax:

**lambda arguments: expression**

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # Program to show the use of lambda functions    double = lambda x: x \* 2    # Output: 10  print(double(5)) |

**Output**

10

In [1]:

In the above program, lambda x: x \* 2 is the Lambda function. Here x is the argument and x \* 2 is the expression that gets evaluated and returned.

This function has no name. It returns a function object which is assigned to the identifierdouble We can now call it as a normal function. The statement

double = lambda x: x \* 2

is nearly the same as

def double(x):

return x \* 2

Next up on this Python Functions section , let us check out the how we can make use of User-defined functions in Python

Ex: 2:

x = lambda a, b : a \* b  
print(x(5, 6))

## ****Python User-Defined Functions****

### **What Are User-Defined Functions In Python?**

Functions that we define ourselves to do the certain specific task are referred to as user-defined functions. The way in which we define and call functions in Python are already discussed.

Functions that readily come with Python are called built-in functions. If we use functions written by others in the form of the library, it can be termed as library functions.

All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

**Advantages of user-defined functions**

1. User-defined functions help to decompose a large program into small segments which makes the program easy to understand, maintain and debug.
2. If repeated code occurs in a program. The function can be used to include those codes and execute when needed by calling that function.
3. Programmers working on a large project can divide the workload by making different functions.

**Syntax**

def **function\_name**(argument1, argument2, ...) :

    statement\_1

    statement\_2

    ....

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # Program to illustrate  # the use of user-defined functions    def add\_numbers(x,y):     sum = x + y     return sum    num1 = 5  num2 = 6    print("The sum is", add\_numbers(num1, num2)) |

**Output**

The sum is 11

Next up on this Python Functions blog, let us check out how we can create a simple application using Python.

**Python Program To Create A Simple Calculator Application**

In this example, you will learn to create a simple calculator that can add, subtract, multiply or divide depending upon the input from the user.

**Code**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | # Program make a simple calculator that can add, subtract, multiply and divide using functions    # This function adds two numbers  def add(x, y):     return x + y    # This function subtracts two numbers  def subtract(x, y):     return x - y    # This function multiplies two numbers  def multiply(x, y):     return x \* y    # This function divides two numbers  def divide(x, y):     return x / y    print("Select operation.")  print("1.Add")  print("2.Subtract")  print("3.Multiply")  print("4.Divide")    # Take input from the user  choice = input("Enter choice(1/2/3/4):")    num1 = int(input("Enter first number: "))  num2 = int(input("Enter second number: "))    if choice == '1':     print(num1,"+",num2,"=", add(num1,num2))    elif choice == '2':     print(num1,"-",num2,"=", subtract(num1,num2))    elif choice == '3':     print(num1,"\*",num2,"=", multiply(num1,num2))    elif choice == '4':     print(num1,"/",num2,"=", divide(num1,num2))  else:     print("Invalid input") |

**Output**

Select operation.

1.Add

2.Subtract

3.Multiply

4.Divide

Enter choice(1/2/3/4): 3

Enter first number: 15

Enter second number: 14

15 \* 14 = 210

In this program, we ask the user to choose the desired operation. Options 1, 2, 3 and 4 are valid. Two numbers are taken and an if...elif...else branching is used to execute a particular section. User-defined functions add(), subtract(), multiply() and divide() evaluate respective operations.

**Conclusion**

The concepts discussed in this section should help you build your own functions using Python by adding functionality and operability to the same.