3.1 Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

3.1.1 Creating a Function

In Python a function is defined using the `def` keyword:

```
def my_function():
  print("Hello from a function")

print(“still inside the function”)
```

3.1.2 Calling a Function

To call a function, use the function name followed by parenthesis:

```
def my_function():
  print("Hello from a function")

my_function()
```

Output

Hello from a function

3.1.3 Parameters

Information can be passed to functions as parameter.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):
  print(fname + " Refsnes")
```

```
my_function("Emil")

my_function("Tobias")

my_function("Linus")
```

Output

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

3.1.4 Default Parameter Value

```
def my_function(country = "Norway"):
  print("I am from " + country)


my_function("Sweden")

my_function("India")

my_function()

my_function("Brazil")
```

Output

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

3.1.5 Passing a List as a Parameter

You can send any data types of parameter to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as a parameter, it will still be a List when it reaches the function:

```
def my_function(food):
  for x in food:
    print(x)
```

```
fruits = ["apple", "banana", "cherry"]


my_function(fruits)
```

Output

```
apple
banana
cherry
```

3.1.6 Return Values

To let a function return a value, use the `return` statement:

```
def my_function(x):
  return 5 * x


print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Output

```
15
25
45
```

3.1.8 Keyword Arguments

You can also send arguments with the *key = value* syntax.

This way the order of the arguments does not matter.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)


my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Output

```
The youngest child is Linus
```

3.1.9 Arbitrary Arguments

If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])
```

```
The youngest child is Linus
```

```
my_function("Emil", "Tobias", "Linus")
```

3.1.10 The pass Statement

`function` definitions cannot be empty, but if you for some reason have a `function` definition with no content, put in the `pass` statement to avoid getting an error.

```
def myfunction:
  pass
```

```
# having an empty function definition like this, would raise an error without the pass statement
```

Output

3.1.11 Recursion

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

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

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.

```
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)
```

```
Recursion Example Results
1
3
6
10
15
21
```

3.1.12 Global and Local Variables in Python

Global variables are the one that are defined and declared outside a function and we need to use them inside a function.

```
Ex 1.# This function uses global variable s
def f():
    print s

# Global scope
s = "I love India"
f()
```

Output

```
I love India
```

```
Ex 2.# This function has a variable with
# name same as s.
def f():
    s = "Me too."
```

```
    print s

# Global scope
s = "I love India"
f()
print s
```

Output

```
Me too.
I love India.
```

Ex 3.

```
def f():
    print s

    # This program will NOT show error
    # if we comment below line.
    s = "Me too."

    print s

# Global scope
s = "I love India"
f()
print s
```

```
Output Line 2: undefined: Error: local variable 's' referenced before assignment
```

Ex 4. `# This function modifies global variable 's'`

```
def f():
    global s
    print s
    s = "Look for this"
    print s

# Global Scope
s = "Python is great!"
f()
print s
```

Output

```
Python is great!
Look for this.
Look for this.
```

Ex 5.

```
a = 1

# Uses global because there is no local 'a'
def f():
    print 'Inside f() : ', a

# Variable 'a' is redefined as a local
def g():
    a = 2
    print 'Inside g() : ',a

# Uses global keyword to modify global 'a'
def h():
    global a
    a = 3
    print 'Inside h() : ',a

# Global scope
print 'global : ',a
f()
print 'global : ',a
g()
print 'global : ',a
h()
print 'global : ',a
```

Output

```
global :  1
Inside f() :  1
global :  1
Inside g() :  2
global :  1
Inside h() :  3
global :  3
```

3.2 String Literals

String literals in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:

print("Hello")

print('Hello')

Output

Hello
Hello

3.2.1 Assign String to a Variable

```
a = "Hello"
print(a)
```

Output

Hello

3.2.2 Multiline Strings

You can assign a multiline string to a variable by using three quotes

```
Ex 1. a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Output

Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.

```
Ex 2. a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(a)
```

Output

Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.

3.2.3 Strings are Arrays

Strings in Python are arrays of bytes representing unicode characters.

Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

```
a = "Hello, World!"
print(a[1])
```

Output

e

3.2.4 Slicing

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

Ex 1.

```
b = "Hello, World!"
print(b[2:5])
```

Output

llo

Ex 2.

```
b = "Hello, World!"
print(b[-5:-2])
```

Output

orl

3.2.5 String Length

To get the length of a string, use the len() function.

```
a = "Hello, World!"
print(len(a))
```

Output

13

3.2.6 String Methods

Python has a set of built-in methods that you can use on strings.

3.2.6.1 Strip

The strip() method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip())
```

Output

Hello, World!

3.2.6.2 lower

```
a = "Hello, World!"
print(a.lower())
```

Output

hello, world!

3.2.6.3 upper

```
a = "Hello, World!"
print(a.upper())
```

Output

HELLO, WORLD!

3.2.6.4 replace

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Output

Jello, World!

3.2.6.5 split

The split() method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
b = a.split(",")
```

```
print(b)
```

Output

```
['Hello', ' World!']
```

3.2.7 Check String

To check if a certain phrase or character is present in a string, we can use the keywords in or not in.

Ex 1.

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" in txt
print(x)
```

Output

True

Ex 2.

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" not in txt
print(x)
```

Output

False

3.2.8 String Concatenation

To concatenate, or combine, two strings you can use the + operator.

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

Output

HelloWorld

3.2.9 String Format

Ex 1.

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

Output

My name is John, and I am 36

Ex 2.

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

Output

I want 3 pieces of item 567 for 49.95 dollars.

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

Output

I want to pay 49.95 dollars for 3 pieces of item 567

3.2.10 Escape Character

```
txt = "We are the so-called \"Vikings\" from the north."
print(txt)
```

Output

We are the so-called "Vikings" from the north.

3.2.11 Index method

The index() method finds the first occurrence of the specified value.

The index() method raises an exception if the value is not found.

The index() method is almost the same as the find() method, the only difference is that the find() method returns -1 if the value is not found. (See example below)

Ex 1.

```
txt = "Hello, welcome to my world."

x = txt.index("welcome")

print(x)
```

Output

7

Ex 2.

Where in the text is the first occurrence of the letter "e" when you only search between position 5 and 10?:

```
txt = "Hello, welcome to my world."
x = txt.index("e", 5, 10)
print(x)
```

Output

8

Ex 3.

```
txt = "Hello, welcome to my world."

print(txt.find("q"))
print(txt.index("q"))
```

Output

```
-1
Traceback (most recent call last):
  File "demo_ref_string_find_vs_index.py", line 4 in <module>
    print(txt.index("q"))
ValueError: substring not found
```

3.2.12 Repeat operator in python

Repetition operator is denoted by a '*' symbol and is useful for repeating strings to a certain length.

Ex 1.

```
str = 'Python program'
print(str*3)
```

Output

Python programPython programPython program

Ex 2.

```
str = 'Python program'
print(str[7:9]*3) #Repeats the seventh and eighth character three times
```

Output

prprpr

3.3 Inbuilt Data Structures in Python

Python has four basic inbuilt data structures namely Lists, Dictionary, Tuple and Set. These almost cover 80% of the our real world data structures.

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

.

Above mentioned topics are divided into four sections below.

1. **Lists :** Lists in Python are one of the most versatile collection object types available. The other two types are dictionaries and tuples, but they are really more like variations of lists.
   - Python lists do the work of most of the collection data structures found in other languages and since they are built-in, you don't have to worry about manually creating them.
   - Lists can be used for any type of object, from numbers and strings to more lists.
   - They are accessed just like strings (e.g. slicing and concatenation) so they are simple to use and they're variable length, i.e. they grow and shrink automatically as they're used.
   - In reality, Python lists are C arrays inside the Python interpreter and act just like an array of pointers.

```
# Python program to illustrate
# A simple list

# Declaring a list
L = [1, "a" , "string" , 1+2]
print L

# add 6 to the above list
L.append(6)
print L

# pop deletes the last element of the
list
L.pop()
print L

print L[1]
```

Output:

```
[1, 'a', 'string', 3]
[1, 'a', 'string', 3, 6]
[1, 'a', 'string', 3]
a
```

There are various Functions that can be carried out on lists like append(), extend(), reverse(), pop() etc.

Dictionary: In python, dictionary is similar to hash or maps in other languages. It consists of key value pairs. The value can be accessed by unique key in the dictionary.
Keys are unique & immutable objects.
Syntax:
dictionary = {"key name": value}

```
# Python program to illustrate
# dictionary

# Create a new dictionary
d = dict() # or d = {}

# Add a key - value pairs to
```

```
dictionary
d['xyz'] = 123
d['abc'] = 345

# print the whole dictionary
print d

# print only the keys
print d.keys()

# print only values
print d.values()

# iterate over dictionary
for i in d :
   print "%s %d" %(i, d[i])

# another method of iteration
for index, value in enumerate(d):
   print index, value , d[value]

# check if key exist
print 'xyz' in d

# delete the key-value pair
del d['xyz']

# check again
print "xyz" in d
```

Output:

```
{'xyz': 123, 'abc': 345}
['xyz', 'abc']
[123, 345]
xyz 123
abc 345
0 xyz 123
1 abc 345
True
False
```

Tuple : Python tuples work exactly like Python lists except they are immutable, i.e. they can't be changed in place. They are normally written inside parentheses to distinguish them from lists (which use square brackets), but as you'll see, parentheses aren't always necessary. Since tuples are immutable, their length is fixed. To grow or shrink a tuple, a new tuple must be created.

Here's a list of commonly used tuples:

() An empty tuple
t1 = (0,) A one-item tuple (not an expression)
t2 = (0, 1, 2, 3) A four-item tuple
t3 = 0, 1, 2, 3 Another four-item tuple (same as prior line, just minus the parenthesis)
t3 = ('abc', ('def', 'ghi')) Nested tuples
t1[n], t3[n][j] Index
t1[i:j], Slice
len(tl) Length

```
# Python program to illustrate
# tuple
tup = (1, "a", "string", 1+2)
print tup
```

```
    print tup[1]
```
Output:
(1, 'a', 'string', 3)
a

Sets: Unordered collection of unique objects.
Set operations such as union (|) , intersection(&), difference(-) can be applied on a set.
Sets are immutable i.e once created further data can't be added to them
() are used to represent a set.Objects placed inside these brackets would be treated as a set.
filter_none
edit
play_arrow
brightness_4

```
    # Python program to demonstrate
    working of
    # Set in Python

    # Creating two sets
    set1 = set()
    set2 = set()

    # Adding elements to set1
    for i in range(1, 6):
        set1.add(i)

    # Adding elements to set2
    for i in range(3, 8):
        set2.add(i)

    print("Set1 = ", set1)
    print("Set2 = ", set2)
    print("\n")
```
Output:
('Set1 = ', set([1, 2, 3, 4, 5]))
('Set2 = ', set([3, 4, 5, 6, 7]))

First Class functions in Python

First class objects in a language are handled uniformly throughout. They may be stored in data structures, passed as arguments, or used in control structures. A programming language is said to support first-class functions if it treats functions as first-class objects. Python supports the concept of First Class functions.
Properties of first class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

Examples illustrating First Class functions in Python

1. Functions are objects: Python functions are first class objects. In the example below, we are assigning function to a variable. This assignment doesn't call the function. It takes the function object referenced by shout and creates a second name pointing to it, yell.

```
# Python program to illustrate
functions
# can be treated as objects
```

```
def shout(text):
   return text.upper()

print shout('Hello')

yell = shout

print yell('Hello')
```
Output
```
HELLO
HELLO
```
2. Functions can be passed as arguments to other functions: Because functions are objects we can pass them as arguments to other functions. Functions that can accept other functions as arguments are also called higher-order functions. In the example below, we have created a function **greet** which takes a function as an argument.
```
# Python program to illustrate functions
# can be passed as arguments to other functions
def shout(text):
   return text.upper()

def whisper(text):
   return text.lower()

def greet(func):
   # storing the function in a variable
   greeting = func("Hi, I am created by a function
passed as an argument.")
   print greeting

greet(shout)
greet(whisper)
```
Output
```
HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.
hi, i am created by a function passed as an argument.
```
3. Functions can return another function: Because functions are objects we can return a function from another function. In the below example, the create_adder function returns adder function.
```
# Python program to illustrate
functions
# Functions can return another
function

def create_adder(x):
   def adder(y):
      return x+y

   return adder

add_15 = create_adder(15)

print add_15(10)
```
```
25
```

What is Lambda?

Lambdas, also known as anonymous functions, are small, restricted functions which do not need a name (i.e., an identifier). Lambda functions were first introduced to the field of mathematics by Alonzo Church in the 1930s.

Today, many modern programming languages like Java, Python, C#, and C++ support lambda functions to add functionality to the languages.

Lambdas in Python

In Python, lambda expressions (or lambda forms) are utilized to construct anonymous functions. To do so, you will use the **lambda** keyword (just as you use **def** to define normal functions).

Every anonymous function you define in Python will have 3 essential parts:

- The lambda keyword.
- The parameters (or bound variables), and
- The function body.

A lambda function can have any number of parameters, but the function body can only contain **one** expression.

Moreover, a lambda is written in a single line of code and can also be invoked immediately. You will see all this in action in the upcoming examples.

Syntax and Examples

The formal syntax to write a lambda function is as given below:

```
lambda p1, p2: expression
```

Here, p1 and p2 are the parameters which are passed to the lambda function. You can add as many or few parameters as you need.

However, notice that we do not use brackets around the parameters as we do with regular functions. The last part (expression) is any valid python expression that operates on the parameters you provide to the function.

Example 1

Now that you know about lambdas let's try it with an example. So, open your IDLE and type in the following:

```
adder = lambda x, y: x + y
print (adder (1, 2))
```

Here is the output:

```
3
```

Code Explanation

Here, we define a variable that will hold the result returned by the lambda function.

1. The lambda keyword used to define an anonymous function.

2. x and y are the parameters that we pass to the lambda function.

3. This is the body of the function, which adds the 2 parameters we passed. Notice that it is a single expression. You cannot write multiple statements in the body of a lambda function.

4. We call the function and print the returned value.

Example 2

That was a basic example to understand the fundamentals and syntax of lambda. Let's now try to print out a lambda and see the result. Again, open your IDLE and type in the following:

```
#What a lambda returns
string='some kind of a useless lambda'
print(lambda string : print(string))
```

Now save your file and hit F5 to run the program. This is the output you should get.

Output:

```
<function <lambda> at 0x00000185C3BF81E0>
```

What's happening here? Let's look at the code to understand further.

Code Explanation

1. Here, we define a string that you'll pass as a parameter to the lambda.
2. We declare a lambda that calls a print statement and prints the result.

But why doesn't the program print the string we pass? This is because the lambda itself returns a function object. In this example, the lambda is not being **called** by the print function but simply **returning** the function object and the memory location where it is stored. That's what gets printed at the console.

Example 3

However, if you write a program like this:

```
#What a lambda returns #2
x="some kind of a useless lambda"
(lambda x : print(x))(x)
```

And run it by hitting F5, you'll see an output like this.

Output:

```
some kind of a useless lambda
```

Now, the lambda is being called, and the string we pass gets printed at the console. But what is that weird syntax, and why is the lambda definition covered in brackets? Let's understand that now.

Code Explanation

1. Here is the same string we defined in the previous example.
2. In this part, we are defining a lambda and calling it immediately by passing the string as an argument. This is something called an IIFE, and you'll learn more about it in the upcoming sections of this tutorial.

Example 4

Let's look at a final example to understand how lambdas and regular functions are executed. So, open your IDLE and in a new file, type in the following:

```
#A REGULAR FUNCTION
def guru( funct, *args ):
funct( *args )
def printer_one( arg ):
return print (arg)
def printer_two( arg ):
print(arg)
#CALL A REGULAR FUNCTION
guru( printer_one, 'printer 1 REGULAR CALL' )
guru( printer_two, 'printer 2 REGULAR CALL \n' )
#CALL A REGULAR FUNCTION THRU A LAMBDA
guru(lambda: printer_one('printer 1 LAMBDA CALL'))
guru(lambda: printer_two('printer 2 LAMBDA CALL'))
```

Now, save the file and hit F5 to run the program. If you didn't make any mistakes, the output should be something like this.

Output:

printer 1 REGULAR CALL

printer 2 REGULAR CALL

printer 1 LAMBDA CALL

printer 2 LAMBDA CALL

Code Explanation

1. A function called guru that takes another function as the first parameter and any other arguments following it.
2. printer_one is a simple function which prints the parameter passed to it and returns it.
3. printer_two is similar to printer_one but without the return statement.
4. In this part, we are calling the guru function and passing the printer functions and a string as parameters.
5. This is the syntax to achieve the fourth step (i.e., calling the guru function) but using lambdas.

In the next section, you will learn how to use lambda functions with **map(), reduce(),** and **filter()** in Python.