Data Science with Python Programming

- Presentation By Uplatz
- Contact us: https://training.uplatz.com
- Email: info@uplatz.com
- Phone: +44 7836 212635

Uplatz

Functions and Modules

Learning outcomes:

- What is a Function?
- Defining a Function and Calling a Function
- Ways to write a function
- Types of functions
- Anonymous Functions
- Recursive function
- What is a module?
- Creating a module
- import Statement
- Locating modules

What is a Function?

A function is a set of statements sorted out together to play out a specific task. Python has a enormous number of in-built functions and the user can also make their own functions.
Functions are utilized to sensibly break our code into less complex parts which become simple to keep up and comprehend.
A developer develops a function to abstain from rehashing a similar assignment, or reduce complexity.

What is a Function?

- In Python, function is a group of related statements that perform a specific task.
- Functions help break our program into smaller and modular chunks.
- Furthermore, it avoids repetition and makes code reusable.

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.

Defining a Function

- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Defining a Function

Syntax of a Function:

```
def function_name(parameters):
    statement(s)
```

Example of Function:

```
def pow (x, y) :
  result = x**y
  print(x,"raised to the power", y, "is", result)
```

Here, we created a function called **pow().**

It takes two arguments, finds the first argument raised to the power of second argument and prints the result in appropriate format.

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
Once a function is defined, you may call by it's **function name** with the required number of arguments/parameters.
Following is the example to call **pow()** function:

```
def pow (x, y) :
   result = x**y
   print(x,"raised to the power", y, "is", result)

pow(5,3)
```

Ways to write a function

With argument with return type

With argument without return type

Without argument with return type

Without argument without return type

Uplatz

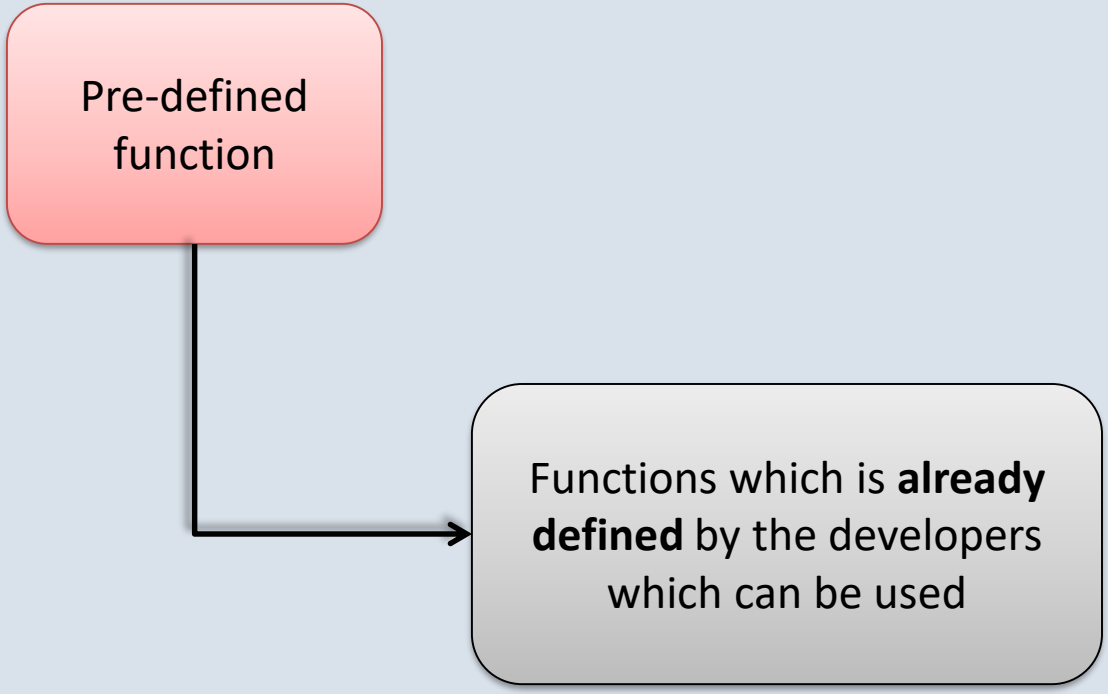
Types of Functions

Pre-defined function

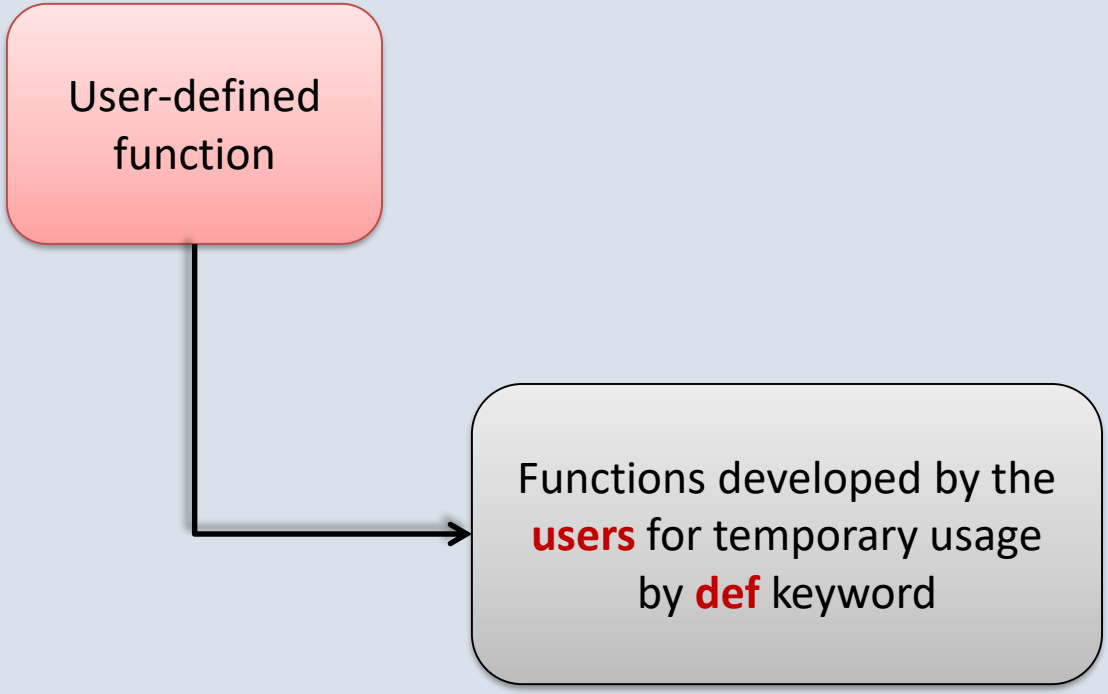
User-defined function

Anonymous function

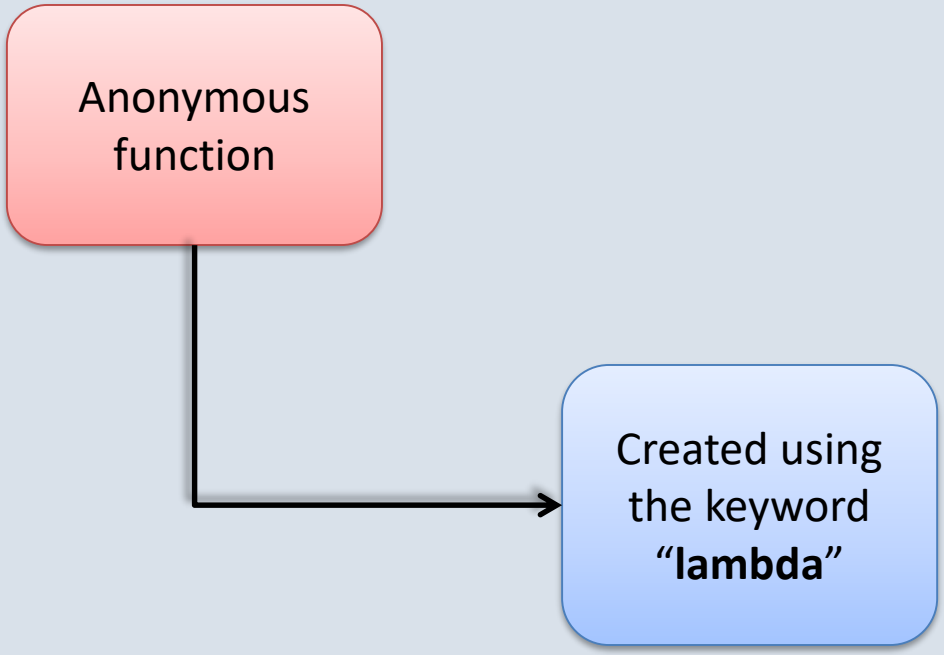
Types of Functions



Types of Functions



Types of Functions



Anonymous Functions

These functions are called **anonymous** because they are not declared in the standard manner by using the *def* keyword. You can use the ***lambda*** keyword to create small anonymous functions.

- Lambda forms can take **any number of arguments** but **return just one value** in the form of an expression.
- They **cannot** contain **commands** or **multiple expressions**.
- An anonymous function **cannot be a direct call to print** because **lambda requires an expression**

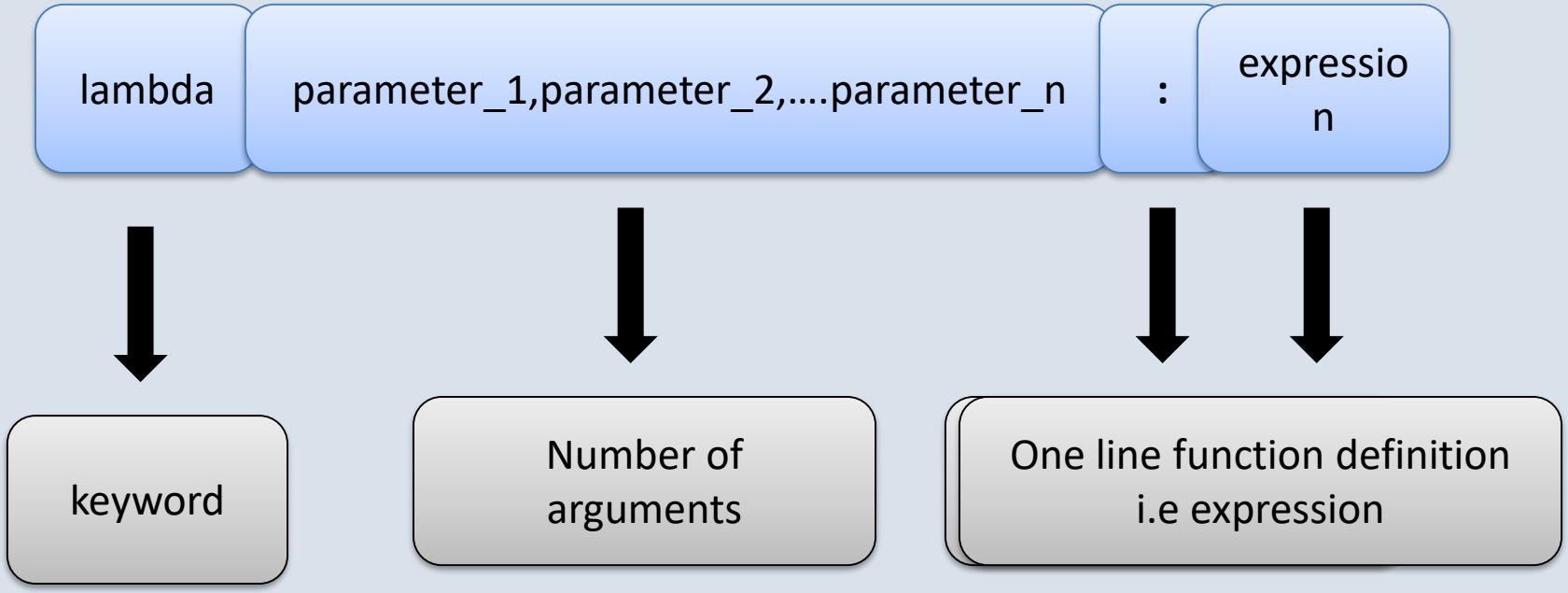
Anonymous Functions

These functions are called **anonymous** because they are not declared in the standard manner by using the *def* keyword. You can use the ***lambda*** keyword to create small anonymous functions.

- Lambda forms can take **any number of arguments** but **return just one value** in the form of an expression.
- They **cannot** contain **commands** or **multiple expressions**.
- An anonymous function **cannot be a direct call to print** because **lambda requires an expression**

Anonymous Functions

Syntax:



Anonymous Functions

Program to add two numbers using function

```
def sum(n1,n2):
    s=n1+n2
    return s
x=int(input("Enter the number"))
y=int(input("Enter the number"))
print(sum(x,y))
```

Input:
Enter the number10
Enter the number20

Program to add two numbers using lambda function :

```
sum = lambda n1,n2 :
    n1+n2
x=int(input("Enter the number"))
y=int(input("Enter t number"))
print(sum(x,y))
```

Output:
30

Uplatz

Recursive Functions

A function that calls itself is called a **recursive function** and this technique is known as **recursion.** This special programming technique can be used to solve problems by breaking them into smaller and simpler sub-problems.

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.

Recursive Functions

An example can help clarify this concept.
Let us take the example of finding the factorial of a number. Factorial of a positive integer number is defined as the product of all the integers from 1 to that number. For example, the factorial of 5 (denoted as 5!) will be
1*2*3*4*5 = 120

Recursive Functions

This problem of finding factorial of 5 can be broken down into a sub-problem of multiplying the factorial of 4 with 5.

5! = 5*4!

Or more generally,

n! = n*(n-1)!

Now we can continue this until we reach 0! which is 1.

Let's see the example.

What is a Module?

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.
Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Creating a module

To create a module just save the code you want in a file with the file extension **.py**

For example:

A simple module, **Calculator.py**

```
def add(x, y):
    return (x+y)

def subtract(x, y):
    return (x-y)

def multiply(x, y):
    return (x*y)
```

import Statement

You can use any Python source file as a module by executing an import statement in some other Python source file.
Now we can use the module we just created, by using the import statement:
When using a function from a module, use the syntax: ***module_name.function_name.***

import Calculator

print (Calculator.add(100, 12))
print (Calculator.multiply(100, 12))

The *from...import* Statement

Python's ***from*** statement lets you import specific attributes from a module.

For Example:

```
from math import sqrt, factorial

# if we simply do "import math", then
# math.sqrt(16) and math.factorial() are needed

print (sqrt(16))
print (factorial(6))
```

Re-naming a Module

You can create an alias when you import a module, by using the **as** keyword:

For Example:

```
import Calculator as clc
print (clc.subtract(100, 12))
print (clc.multiply(10, 12))
```

The *from...import* * Statement

It is also possible to import all names from a module into the current namespace by using the following import statement:

from modulename import *

This provides an easy way to import all the items from a module into the current namespace.

For Example:

```
from math import *
print(pow(5,3))
print(radians(30))
print(sqrt(36))
print(degrees(0.5239))
```

Locating modules

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

By the way, we can find the current working directory of python as follows:

```
import os
print(os.getcwd())
```