Special Note on python basic programing

# 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.

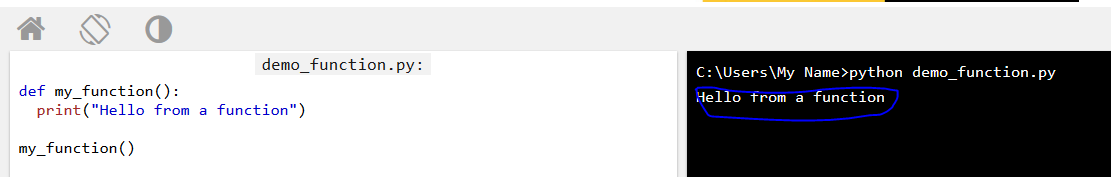
A function can return data as a result.

## Creating a Function

In Python a function is defined using the def keyword:

## Calling a Function

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

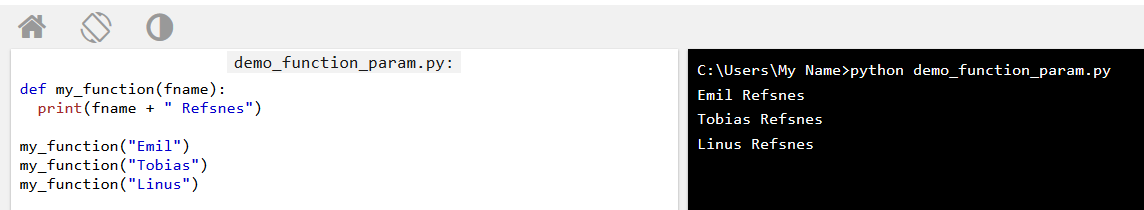


## Parameters

Information can be passed to functions as parameter.

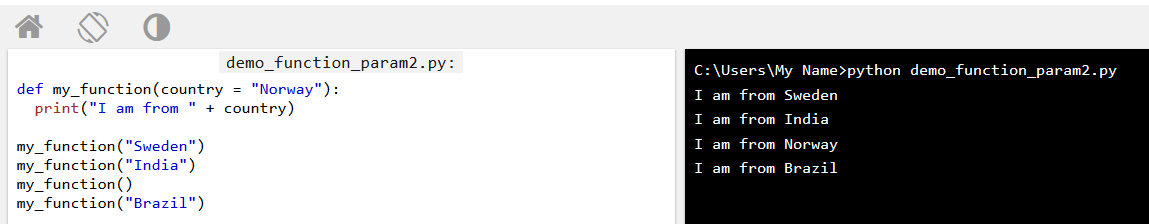
Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

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:

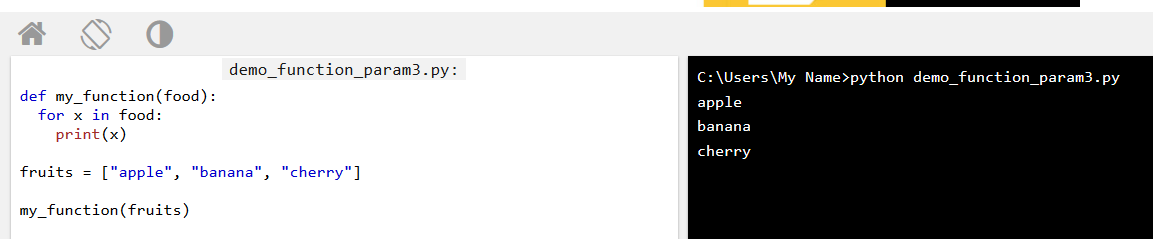


## Default Parameter Value

If we call the function without parameter, it uses the default value:

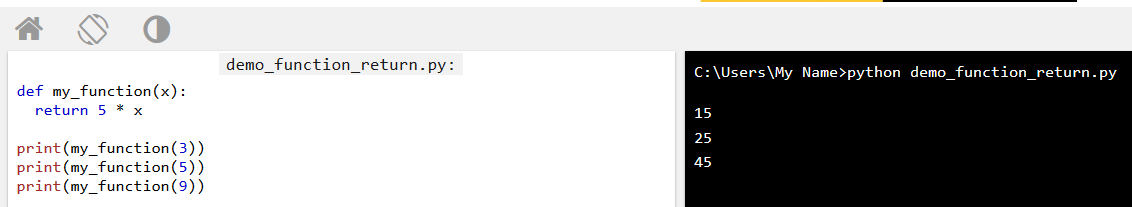


## Passing a List as a Parameter

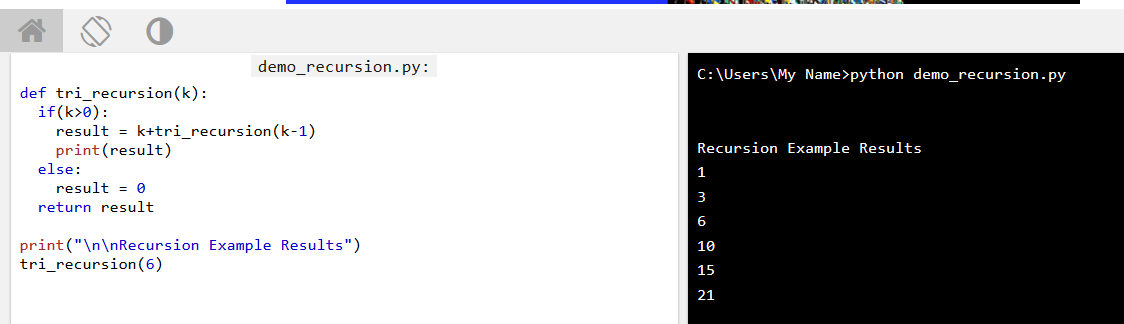


## Return Values

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



## Recursion



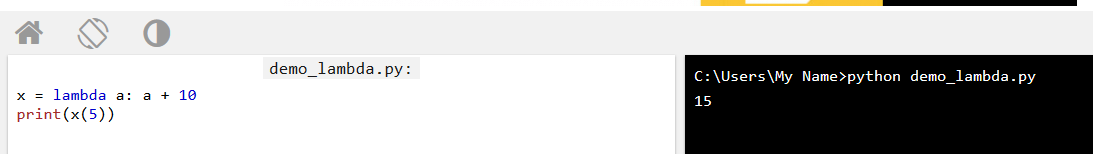
# Python Lambda

A lambda function can take any number of arguments, but can only have one expression.

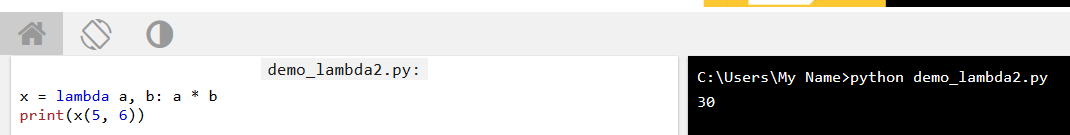
## Syntax

lambda arguments/parameter : expression

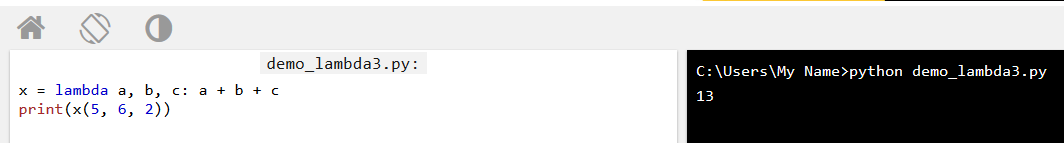
A lambda function that adds 10 to the number passed in as an argument, and print the result:



A lambda function that multiplies argument a with argument b and print the result:



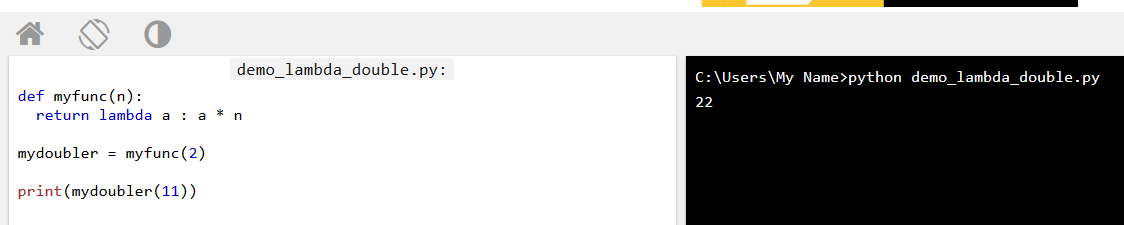
A lambda function that sums argument a, b, and c and print the result:



## Why Use Lambda Functions?

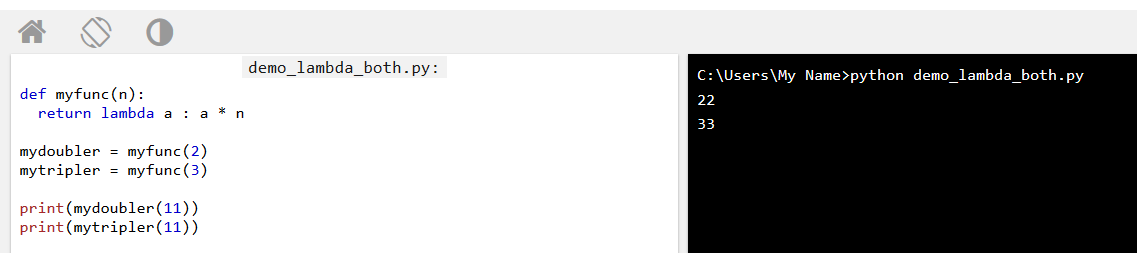
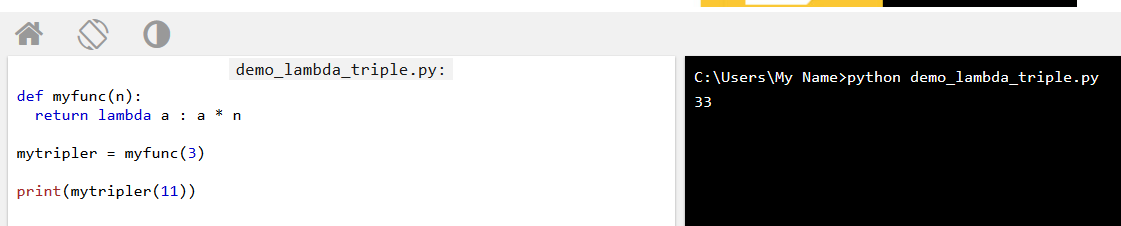
The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:



Or, use the same function definition to make a function that always triples the number you send in:

Or, use the same function definition to make both functions, in the same program:



# ফাংশন ও মডিউল

যেকোনো রকম কম্পিউটার প্রোগ্রামিং -এ দুটি টার্ম শুনতে পাওয়া যায়। **WET** এবং **DRY**. WET মানে বলা হয় **Write Everything Twice** অথবা **We Enjoy Writing**. অর্থাৎ যেকোনো প্রোগ্রামের মধ্যে অপ্রয়োজনীয় এবং অতিরিক্ত স্টেটমেন্ট লেখা বা ব্যবহার করা যেখানে সেই একই কাজ অনেক কম কোড লিখেও আরও অপ্টিমাইজ ভাবে করা যেত। এটা খারাপ অভ্যাস। আবার DRY মানে বলা হয় - **Do not Repeat Yourself**. অর্থাৎ ওই যে ১০ বার ১০টা লাইন প্রিন্ট করার জন্য ১০ বার প্রিন্ট স্টেটমেন্ট ব্যবহার না করা। যেহেতু কাজটা একই (কিছু একটা প্রিন্ট করা) সেহেতু একবার প্রিন্ট স্টেটমেন্ট লিখেই কোন না কোন ভাবে ১০ বার প্রিন্ট করা সম্ভব। এটা ভালো অভ্যাস।

## ফাংশন

একটি ফাংশন আসলে কিছু স্টেটমেন্টের সংকলন । যখনই কোন ফাংশন কল করা হয় তখন এই ফাংশনের ভিতরে থাকা স্টেটমেন্টগুলো এক্সিকিউট করা হয় । পাইথনে আমরা ফাংশন ডিক্লেয়ার করার জন্য def কি-ওয়ার্ডটি ব্যবহার করি

# Defining the function named hello

def hello():

    print("Hello World!")

# Calling the function to use it

hello()

hello world

আমরা def কি-ওয়ার্ডটি লিখেছি । তারপর ফাংশনের একটা নাম দিয়েছি – hello, এবং তারপর একজোড়া প্রথম বন্ধনী (). এরপর একটি কোলন তথা : চিহ্ন দিয়ে এর নিচে ফাংশনের আওতাভুক্ত কোড ব্লক বা ফাংশনের কাজ ডিফাইন (নির্ধারণ) করেছি।আমরা print ফাংশন ব্যবহার করি যখন দরকার হয় তখন। উপরের ফাংশনটিকে কল করতে হবে নিচের মত করে, একটা ফাংশনকে প্রোগ্রামে একবার ডিফাইন করলেও সেটাকে বার বার কল বা ব্যবহার করা যাবে।

## ফাংশন আর্গুমেন্ট

ধরে নিচ্ছি আমাদের বানানো মেশিনটির এক পাশ দিয়ে ময়দা, চিনি, দুধ, ক্রিম এসব দিলে আরেক পাশ দিয়ে সুন্দর কেক তৈরি হয়ে বের হয়। তাহলে সেই ময়দা, চিনি, দুধ, ক্রিম এসব হচ্ছে সেই মেশিনের **আর্গুমেন্ট** আর কেক বানানোর জন্য মেশিনের মধ্যে বিভিন্ন যন্ত্রের যে সেটআপ আছে সেটাকে বলা যেতে পারে **ফাংশন বডি**। আর শেষে যে সুস্বাদু কেক পাওয়া যায় তাকে বলা যেতে পারে ফাংশনের **রিটার্ন ভ্যালু**।

def show\_double(x):

    print(x\*2)

show\_double(2)

show\_double(100)

4

200

উপরে show\_double ফাংশনের আর্গুমেন্ট একটি। আর তাই যখনই আমরা এই ফাংশনকে কল করেছি বা ব্যবহার করতে চেয়েছি তখনি সেই ফাংশনের আর্গুমেন্ট (মেশিনের ক্ষেত্রে ইনপুট) পাঠিয়ে দিয়েছি এভাবে show\_double(2)। একবার কল করার সময় ইনপুট দিয়েছি 2 আবার আরেকবার কল করার সময় ইনপুট দিয়েছি 100 এবং আমাদের ফাংশনের কাজ হচ্ছে এর কাছে আসা যেকোনো আর্গুমেন্টকে দ্বিগুণ করে স্ক্রিনে প্রিন্ট করে। তাই দুইবারই আমাদের ফাংশন কাজটি সঠিক ভাবে করেছে

আর্গুমেন্টকে ফাংশনের দুটি প্রথম বন্ধনীর মধ্যে ডিফাইন করতে হয়।

একটি ফাংশন কিন্তু একাধিক আর্গুমেন্ট নিয়ে কাজ করতে পারে অর্থাৎ এর একাধিক আর্গুমেন্ট থাকতে পারে।

def make\_sum(x, y):

    z = x + y

    print(z)

make\_sum(5, 10)

make\_sum(500, 500)

15

1000

একটি বিষয় খেয়াল করুন, ফাংশনের আর্গুমেন্ট গুলোকে তার নিজের বডির মধ্যে একই নামের ভ্যারিয়েবল হিসেবে ব্যবহার করা যায়। যেমন উপরের উদাহরণে, make\_sum ফাংশনের কাছে দুটো আর্গুমেন্ট এসেছে x, এবং y নামে এবং এই দুটি ভ্যালুকে সে নিজের বডির মধ্যে ব্যবহার করেছে যোগ করার জন্য এবং যোগফল জমা করেছে z নামের আরেকটি ভ্যারিয়েবলে।

কিন্তু এই x, y বা z কে উক্ত ফাংশনের বাইরে থেকে অ্যাক্সেস করা যাবে না বা ব্যবহার করা যাবে না। যেমন –

def make\_sum(x, y):

    z = x + y

    print(z)

make\_sum(5, 10)

print(z)

15

...

NameError: name 'z' is not defined

উপরের উদাহরণে, print(z) স্টেটমেন্টটি এরর দেখাচ্ছে কারণ z ভ্যারিয়েবলের গণ্ডি বা স্কোপ ছিল শুধুমাত্র make\_sum ফাংশনের মধ্যেই। তাই বাইরে থেকে একে অ্যাক্সেস করা যায় নি।

## মাল্টিপল প্যারামিটার হ্যান্ডলিং | আর্বিটরারি আর্গুমেন্ট লিস্ট

মনে করুন, আপনি make\_sum ফাংশনটিতে অনেকগুলো প্যারামিটার পাঠাতে চাচ্ছেন যেমন, 10, 20, 30 ... ইত্যাদি। যদি আপনি make\_sum (a, b) হিসেবে ডিক্লেয়ার করেন তাহলে দুইটার বেশি প্যারামিটার পাঠাতে পারবেন না।

def make\_sum(\*args):

    sum = 0

    for num in args: # Here, args is like a Tuple which is (10, 20, 30, 40)

        sum += num

    return sum

print(make\_sum(10, 20, 30, 40))

100

## পাইথনে \* এর অর্থ

\* এর আর্গুমেন্টে ভ্যালু Tuple হিসেবে প্যাকড থাকে। এর মানে \* দিয়ে প্যারামিটার ডিক্লেয়ার করলে আমরা যেকোন সংখ্যক পজিশনাল আর্গুমেন্ট পাস করতে পারি. শুরুতে make\_sum মাত্র দুইটা আর্গুমেন্ট নিলেও পরবর্তীতে আমরা প্যারামিটারে \* বসিয়ে দিলাম তখন সে অনেকগুলো আর্গুমেন্ট পাস করতে পারছে।

## পাইথনে \*\* এর অর্থ

আমরা চাইলে ফাংশনের প্যারামিটারে ডাবল অ্যাস্টেরিস্কস বসিয়েও ডিক্লেয়ার করতে পারি। ডাবল স্টারের মানে হল যেকোন সংখ্যক named parameter থাকতে পারে। এই মানগুলো ডিকশনারি হিসেবে প্যাকড থাকে।

def print\_dict(\*args):

    print (args)

print\_dict(a=1, b=2)

সিঙ্গেল অ্যাস্টেরিস্কস ব্যবহার করলে আমরা নেমড আর্গুমেন্ট পাস করতে পারব না। তাই আমাদের এসব ক্ষেত্রে ডাবল অ্যাস্টেরিস্কস ব্যবহার করতে হবে, যেমন

def print\_dict(\*\*kwargs):

    print(kwargs)

print\_dict(a=1, b=2, c=3)

{'a': 1, 'c': 3, 'b': 2}

আমরা যদি কোডটা আরেকটু গুছিয়ে লেখি,

def print\_dict(\*\*kwargs):

    for args in kwargs:

        print("{0} : {1}".format(args, kwargs[args]))

print\_dict(a=1, b=2, c=3)

চাইলে আমরা মিক্সড ভ্যারিয়েডিক আর্গুমেন্ট পাঠাতে পারি। মানে একই ফাংশনে তিন ধরণের আর্গুমেন্ট, তবে খেয়াল রাখতে হবে প্যারামিটারগুলো এমন ভাবে ডিফাইন করা হয় যেন প্রথমে সাধারণ প্যারামিটার তারপরে সিঙ্গেল অ্যাস্টেরিস্কের প্যারামিটার এবং অবশেষে ডাবল অ্যাস্টেরিস্কস এর প্যারামিটার থাকে। মানে আমাদের অবশ্যই ক্রম মানতে হবে এইক্ষেত্রে।

def print\_all(a, \*args, \*\*kwargs):

    print(a)

    print(args)

    print(kwargs)

print\_all(10, 20, 30, 50, b=5, c=10)

## প্যারামিটার ও আর্গুমেন্ট

যখন একটি ফাংশনকে ডিফাইন করা হয় তখন এর ভ্যারিয়েবল গুলোকে প্যারামিটার বলা হয়। আর যখন একটি ফাংশনকে কল করা হয় তখন সেই ফাংশনের প্যারামিটার হিসেবে যে ভ্যালু পাঠানো হয় তাকে আর্গুমেন্ট বলা হয়।

# ফাংশন রিটার্ন

কোন একটি ফাংশন তার বডির মধ্যে কিছু কাজ করে চুপ চাপ থাকতে পারে অথবা কোন কিছু রিটার্নও দিতে পারে। কাকে রিটার্ন দিবে? যে এই ফাংশনকে কল করবে বা ব্যবহার করবে। কোথায় রিটার্ন দিবে? যেখান থেকে কল করা হবে সেখানেই রিটার্ন ভ্যালু পৌঁছে যাবে।

def get\_larger(x, y):

    if x > y:

        return x

    else:

        return y

larger\_value = get\_larger(23, 32)

print(larger\_value)

32

ফাংশনের মধ্যে return কিওয়ার্ড ব্যবহার করে ফাংশন থেকে কোন কিছু রিটার্ন করা হয়। উপরের উদাহরণে get\_larger ফাংশনের কাছে আশা দুটো আর্গুমেন্টের মধ্যে সে তুলনা করে বড়টি বের করে এবং সেটি রিটার্ন করে। আর তাই নিচে যখন = get\_larger(23, 32) স্টেটমেন্টের মাধ্যমে একে কল করা হয়েছে এবং এর চাহিদা মোতাবেক দুটো আর্গুমেন্ট পাঠিয়ে দেয়া হয়েছে তখন বস্তুত সেই ফাংশনের রিটার্ন করা ভ্যালুটি = চিহ্নের ডান পাশে এসে জমা হয়। আর যেহেতু আমরা জানি = চিহ্ন দিয়ে কোন ভ্যালুকে কোন ভ্যারিয়েবলে অ্যাসাইন করা হয়, তাই larger\_value এর মধ্যে সেই রিটার্ন করা ভ্যালু স্টোর হচ্ছে।

খুব গুরুত্বপূর্ণ একটি বিষয় হচ্ছে, যখন কোন ফাংশনের মধ্যে একটি return স্টেটমেন্ট থাকে এবং সেটি এক্সিকিউট হয়, তারপর থেকে ফাংশনের মধ্যে থাকা আর কোন কোড এক্সিকিউট হয় না। অর্থাৎ, ফাংশন তার কাজ শেষে কিছু একটা রিটার্ন করেই থেমে যায়

def add\_numbers(x, y):

    total = x + y

    return total

    print("This won't be printed")

print(add\_numbers(4, 5))

9

উপরের প্রোগ্রামের ফাংশনটির কাজ হচ্ছে দুটো আর্গুমেন্ট ভ্যারিয়েবলকে যোগ করে নতুন একটি ভ্যারিয়েবলে রেখে সেটিকে তার caller এর কাছে রিটার্ন করা । রিটার্ন করেই সে ক্ষান্ত। এরপরে আরেকটি প্রিন্ট স্টেটমেন্ট থাকলেও সেটার কোন এক্সিকিউশন নেই।

### Use (\*args)

but using (\*args) we can pass unlimited parameter, it output show a tuple

def coco(\*args):

    print(args)

coco(1,2,3,4,5,6)

(1, 2, 3, 4, 5, 6)

def mah(\*args):

    for x in args:

        print(x)

mah(1,1,1,1,1,)

11111

But not work below program

def mah(\*args):

    for x in args:

        print(x)

mah(a=1,b=1,c=1,d=1)

TypeError: mah() got an unexpected keyword argument 'a'

### Use (\*\*kwargs)

By using this formate we can pass/show output parameter and argument both in a carly braket/second braket

def colo(\*\*kwargs):

    print(kwargs)

colo(a=2,b=3,c=4)

{'a': 2, 'b': 3, 'c': 4}

### পাইথনে \* এর অর্থ

\* এর আর্গুমেন্টে ভ্যালু Tuple হিসেবে প্যাকড থাকে। এর মানে \* দিয়ে প্যারামিটার ডিক্লেয়ার করলে আমরা যেকোন সংখ্যক পজিশনাল আর্গুমেন্ট পাস করতে পারি. শুরুতে make\_sum মাত্র দুইটা আর্গুমেন্ট নিলেও পরবর্তীতে আমরা প্যারামিটারে \* বসিয়ে দিলাম তখন সে অনেকগুলো আর্গুমেন্ট পাস করতে পারছে।

### পাইথনে \*\* এর অর্থ

আমরা চাইলে ফাংশনের প্যারামিটারে ডাবল অ্যাস্টেরিস্কস বসিয়েও ডিক্লেয়ার করতে পারি। ডাবল স্টারের মানে হল যেকোন সংখ্যক named parameter থাকতে পারে। এই মানগুলো ডিকশনারি হিসেবে প্যাকড থাকে।

সিঙ্গেল অ্যাস্টেরিস্কস ব্যবহার করলে আমরা নেমড আর্গুমেন্ট পাস করতে পারব না। তাই আমাদের এসব ক্ষেত্রে ডাবল অ্যাস্টেরিস্কস ব্যবহার করতে হবে, যেমন

def print\_dict(\*\*kwargs):

    print(kwargs)

print\_dict(a=1, b=2, c=3)

{'a': 1, 'c': 3, 'b': 2}

def print\_dict(\*\*kwargs):

    for args in kwargs:

        print("{0} : {1}".format(args, kwargs[args]))

print\_dict(a=1, b=2, c=3)

চাইলে আমরা মিক্সড ভ্যারিয়েডিক আর্গুমেন্ট পাঠাতে পারি। মানে একই ফাংশনে তিন ধরণের আর্গুমেন্ট, তবে খেয়াল রাখতে হবে প্যারামিটারগুলো এমন ভাবে ডিফাইন করা হয় যেন প্রথমে সাধারণ প্যারামিটার তারপরে সিঙ্গেল অ্যাস্টেরিস্কের প্যারামিটার এবং অবশেষে ডাবল অ্যাস্টেরিস্কস এর প্যারামিটার থাকে। মানে আমাদের অবশ্যই ক্রম মানতে হবে এইক্ষেত্রে।

def print\_all(a, \*args, \*\*kwargs):

    print(a)

    print(args)

    print(kwargs)

print\_all(10, 20, 30, 50, b=5, c=10)

# মাল্টিপল প্যারামিটার হ্যান্ডলিং | আর্বিটরারি আর্গুমেন্ট লিস্ট/\*args and \*\*kwargs

মনে করুন, আপনি make\_sum ফাংশনটিতে অনেকগুলো প্যারামিটার পাঠাতে চাচ্ছেন যেমন, 10, 20, 30 ... ইত্যাদি। যদি আপনি make\_sum (a, b) হিসেবে ডিক্লেয়ার করেন তাহলে দুইটার বেশি প্যারামিটার পাঠাতে পারবেন না।

ef multi\_sum(x,y):

    mul=x+y

    print(mul)

multi\_sum(10,20)

30

### Use (\*args)

but using (\*args) we can pass unlimited parameter, it output show a tuple

def coco(\*args):

    print(args)

coco(1,2,3,4,5,6)

(1, 2, 3, 4, 5, 6)

def mah(\*args):

    for x in args:

        print(x)

mah(1,1,1,1,1,)

11111

But not work below program

def mah(\*args):

    for x in args:

        print(x)

mah(a=1,b=1,c=1,d=1)

TypeError: mah() got an unexpected keyword argument 'a'

### Use (\*\*kwargs)

By using this formate we can pass/show output parameter and argument both in a carly braket/second braket

def colo(\*\*kwargs):

    print(kwargs)

colo(a=2,b=3,c=4)

{'a': 2, 'b': 3, 'c': 4}

### পাইথনে \* এর অর্থ

\* এর আর্গুমেন্টে ভ্যালু Tuple হিসেবে প্যাকড থাকে। এর মানে \* দিয়ে প্যারামিটার ডিক্লেয়ার করলে আমরা যেকোন সংখ্যক পজিশনাল আর্গুমেন্ট পাস করতে পারি. শুরুতে make\_sum মাত্র দুইটা আর্গুমেন্ট নিলেও পরবর্তীতে আমরা প্যারামিটারে \* বসিয়ে দিলাম তখন সে অনেকগুলো আর্গুমেন্ট পাস করতে পারছে।

### পাইথনে \*\* এর অর্থ

আমরা চাইলে ফাংশনের প্যারামিটারে ডাবল অ্যাস্টেরিস্কস বসিয়েও ডিক্লেয়ার করতে পারি। ডাবল স্টারের মানে হল যেকোন সংখ্যক named parameter থাকতে পারে। এই মানগুলো ডিকশনারি হিসেবে প্যাকড থাকে।

সিঙ্গেল অ্যাস্টেরিস্কস ব্যবহার করলে আমরা নেমড আর্গুমেন্ট পাস করতে পারব না। তাই আমাদের এসব ক্ষেত্রে ডাবল অ্যাস্টেরিস্কস ব্যবহার করতে হবে, যেমন

def print\_dict(\*\*kwargs):

    print(kwargs)

print\_dict(a=1, b=2, c=3)

{'a': 1, 'c': 3, 'b': 2}

def print\_dict(\*\*kwargs):

    for args in kwargs:

        print("{0} : {1}".format(args, kwargs[args]))

print\_dict(a=1, b=2, c=3)

চাইলে আমরা মিক্সড ভ্যারিয়েডিক আর্গুমেন্ট পাঠাতে পারি। মানে একই ফাংশনে তিন ধরণের আর্গুমেন্ট, তবে খেয়াল রাখতে হবে প্যারামিটারগুলো এমন ভাবে ডিফাইন করা হয় যেন প্রথমে সাধারণ প্যারামিটার তারপরে সিঙ্গেল অ্যাস্টেরিস্কের প্যারামিটার এবং অবশেষে ডাবল অ্যাস্টেরিস্কস এর প্যারামিটার থাকে। মানে আমাদের অবশ্যই ক্রম মানতে হবে এইক্ষেত্রে।

def print\_all(a, \*args, \*\*kwargs):

    print(a)

    print(args)

    print(kwargs)

print\_all(10, 20, 30, 50, b=5, c=10)

## Creating a Function

In Python a function is defined using the def keyword:

### Example

def my\_function():  
  print("Hello from a function")

## Calling a Function

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

### Example

def my\_function():  
  print("Hello from a function")  
  
**my\_function()**

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function)

## 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:

### Example

def my\_function(**fname**):  
  print(fname + " Refsnes")  
  
my\_function(**"Emil"**)  
my\_function(**"Tobias"**)  
my\_function(**"Linus"**)

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_param)

## Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without parameter, it uses the default value:

### Example

def my\_function(**country = "Norway"**):  
  print("I am from " + country)  
  
my\_function("Sweden")  
my\_function("India")  
my\_function()  
my\_function("Brazil")

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_param2)

## 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:

### Example

def my\_function(food):  
  for x in food:  
    print(x)  
  
fruits = ["apple", "banana", "cherry"]  
  
my\_function(fruits)

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_param3)

## Return Values

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

### Example

def my\_function(x):  
  **return 5 \* x**  
print(my\_function(3))  
print(my\_function(5))  
print(my\_function(9))

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_return)

## Keyword Arguments

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

This way the order of the arguments does not matter.

### Example

def my\_function(child3, child2, child1):  
  print("The youngest child is " + child3)  
  
my\_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_kwargs)

The phrase Keyword Arguments are often shortened to kwargs in Python documentations.

## 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:

### Example

If the number of arguments are unknown, add a \* before the parameter name:

def my\_function(\*kids):  
  print("The youngest child is " + kids[2])  
  
my\_function("Emil", "Tobias", "Linus")

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_args)

## 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.

### Example

def myfunction:  
  pass

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_function_pass)

## 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.

### Example

Recursion Example

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)

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_recursion)

# Python Lambda

[❮ Previous](https://www.w3schools.com/python/python_functions.asp)[Next ❯](https://www.w3schools.com/python/python_arrays.asp)

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

## Syntax

lambda arguments : expression

The expression is executed and the result is returned:

### Example

A lambda function that adds 10 to the number passed in as an argument, and print the result:

x = lambda a : a + 10  
print(x(5))

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_lambda)

Lambda functions can take any number of arguments:

### Example

A lambda function that multiplies argument a with argument b and print the result:

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

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_lambda2)

### Example

A lambda function that sums argument a, b, and c and print the result:

x = lambda a, b, c : a + b + c  
print(x(5, 6, 2))

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_lambda3)

## Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

def myfunc(n):  
  return lambda a : a \* n

Use that function definition to make a function that always doubles the number you send in:

### Example

def myfunc(n):  
  return lambda a : a \* n  
  
mydoubler = myfunc(2)  
  
print(mydoubler(11))

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_lambda_double)

Or, use the same function definition to make a function that always triples the number you send in:

### Example

def myfunc(n):  
  return lambda a : a \* n  
  
mytripler = myfunc(3)  
  
print(mytripler(11))

[Run example »](https://www.w3schools.com/python/showpython.asp?filename=demo_lambda_triple)

Or, use the same function definition to make both functions, in the same program:

### Example

def myfunc(n):  
  return lambda a : a \* n  
  
mydoubler = myfunc(2)  
mytripler = myfunc(3)  
  
print(mydoubler(11))  
print(mytripler(11))

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

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*.
* 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.

Syntax

def functionname( parameters ):

"function\_docstring"

function\_suite

return [expression]

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

def printme( str ):

"This prints a passed string into this function"

print str

return

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 the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

[Live Demo](http://tpcg.io/3mrnYY)

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme("I'm first call to user defined function!")

printme("Again second call to the same function")

When the above code is executed, it produces the following result −

I'm first call to user defined function!

Again second call to the same function

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

[Live Demo](http://tpcg.io/ZwGczd)

#!/usr/bin/python

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

mylist.append([1,2,3,4]);

print "Values inside the function: ", mylist

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

[Live Demo](http://tpcg.io/xgrZcs)

#!/usr/bin/python

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

mylist = [1,2,3,4]; # This would assig new reference in mylist

print "Values inside the function: ", mylist

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result −

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

Function Arguments

You can call a function by using the following types of formal arguments −

* Required arguments
* Keyword arguments
* Default arguments
* Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

[Live Demo](http://tpcg.io/KIgYav)

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme()

When the above code is executed, it produces the following result −

Traceback (most recent call last):

File "test.py", line 11, in <module>

printme();

TypeError: printme() takes exactly 1 argument (0 given)

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

[Live Demo](http://tpcg.io/A0FJD1)

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str

return;

# Now you can call printme function

printme( str = "My string")

When the above code is executed, it produces the following result −

My string

The following example gives more clear picture. Note that the order of parameters does not matter.

[Live Demo](http://tpcg.io/hdONo3)

#!/usr/bin/python

# Function definition is here

def printinfo( name, age ):

"This prints a passed info into this function"

print "Name: ", name

print "Age ", age

return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

[Live Demo](http://tpcg.io/JWyo6q)

#!/usr/bin/python

# Function definition is here

def printinfo( name, age = 35 ):

"This prints a passed info into this function"

print "Name: ", name

print "Age ", age

return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

Name: miki

Age 35

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

def functionname([formal\_args,] \*var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

[Live Demo](http://tpcg.io/qofSxL)

#!/usr/bin/python

# Function definition is here

def printinfo( arg1, \*vartuple ):

"This prints a variable passed arguments"

print "Output is: "

print arg1

for var in vartuple:

print var

return;

# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )

When the above code is executed, it produces the following result −

Output is:

10

Output is:

70

60

50

The *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
* Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
* Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

lambda [arg1 [,arg2,.....argn]]:expression

Following is the example to show how *lambda* form of function works −

[Live Demo](http://tpcg.io/TGEvCu)

#!/usr/bin/python

# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function

print "Value of total : ", sum( 10, 20 )

print "Value of total : ", sum( 20, 20 )

When the above code is executed, it produces the following result −

Value of total : 30

Value of total : 40

The *return* Statement

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.

All the above examples are not returning any value. You can return a value from a function as follows −

[Live Demo](http://tpcg.io/FHnLKA)

#!/usr/bin/python

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2

print "Inside the function : ", total

return total;

# Now you can call sum function

total = sum( 10, 20 );

print "Outside the function : ", total

When the above code is executed, it produces the following result −

Inside the function : 30

Outside the function : 30

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

* Global variables
* Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

[Live Demo](http://tpcg.io/LFZtHG)

#!/usr/bin/python

total = 0; # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2; # Here total is local variable.

print "Inside the function local total : ", total

return total;

# Now you can call sum function

sum( 10, 20 );

print "Outside the function global total : ", total

When the above code is executed, it produces the following result −

Inside the function local total : 30

Outside the function global total : 0

# Python - Modules

Advertisements

[Previous Page](https://www.tutorialspoint.com/python/python_functions.htm)

[Next Page](https://www.tutorialspoint.com/python/python_files_io.htm)

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.

## Example

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, support.py

def print\_func( par ):

print "Hello : ", par

return

## The *import* Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax −

import module1[, module2[,... moduleN]

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module support.py, you need to put the following command at the top of the script −

#!/usr/bin/python

# Import module support

import support

# Now you can call defined function that module as follows

support.print\_func("Zara")

When the above code is executed, it produces the following result −

Hello : Zara

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

## The *from...import* Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax −

from modname import name1[, name2[, ... nameN]]

For example, to import the function fibonacci from the module fib, use the following statement −

from fib import fibonacci

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

## 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 modname import \*

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

## Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences −

* The current directory.
* If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
* If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

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.

## The *PYTHONPATH* Variable

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system −

set PYTHONPATH = c:\python20\lib;

And here is a typical PYTHONPATH from a UNIX system −

set PYTHONPATH = /usr/local/lib/python

## Namespaces and Scoping

Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable *Money* in the global namespace. Within the function *Money*, we assign *Money* a value, therefore Python assumes *Money* as a local variable. However, we accessed the value of the local variable *Money* before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

#!/usr/bin/python

Money = 2000

def AddMoney():

# Uncomment the following line to fix the code:

# global Money

Money = Money + 1

print Money

AddMoney()

print Money

## The dir( ) Function

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example −

[Live Demo](http://tpcg.io/ZoifUr)

#!/usr/bin/python

# Import built-in module math

import math

content = dir(math)

print content

When the above code is executed, it produces the following result −

['\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', 'acos', 'asin', 'atan',

'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',

'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',

'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',

'sqrt', 'tan', 'tanh']

Here, the special string variable *\_\_name\_\_* is the module's name, and *\_\_file\_\_* is the filename from which the module was loaded.

## The *globals()* and *locals()* Functions

The *globals()* and *locals()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If locals() is called from within a function, it will return all the names that can be accessed locally from that function.

If globals() is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

## The *reload()* Function

When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the *reload()* function. The reload() function imports a previously imported module again. The syntax of the reload() function is this −

reload(module\_name)

Here, *module\_name* is the name of the module you want to reload and not the string containing the module name. For example, to reload *hello* module, do the following −

reload(hello)

## Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

Consider a file *Pots.py* available in *Phone* directory. This file has following line of source code −

#!/usr/bin/python

def Pots():

print "I'm Pots Phone"

Similar way, we have another two files having different functions with the same name as above −

* *Phone/Isdn.py* file having function Isdn()
* *Phone/G3.py* file having function G3()

Now, create one more file \_\_init\_\_.py in *Phone* directory −

* Phone/\_\_init\_\_.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in \_\_init\_\_.py as follows −

from Pots import Pots

from Isdn import Isdn

from G3 import G3

After you add these lines to \_\_init\_\_.py, you have all of these classes available when you import the Phone package.

#!/usr/bin/python

# Now import your Phone Package.

import Phone

Phone.Pots()

Phone.Isdn()

Phone.G3()

When the above code is executed, it produces the following result −

I'm Pots Phone

I'm 3G Phone

I'm ISDN Phone

In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

## Factorial

def fun(n):

    if (n==0):

        return 0

    elif (n==1):

        return 1

    return n\*fun(n-1)

s=fun(4)

print(s)