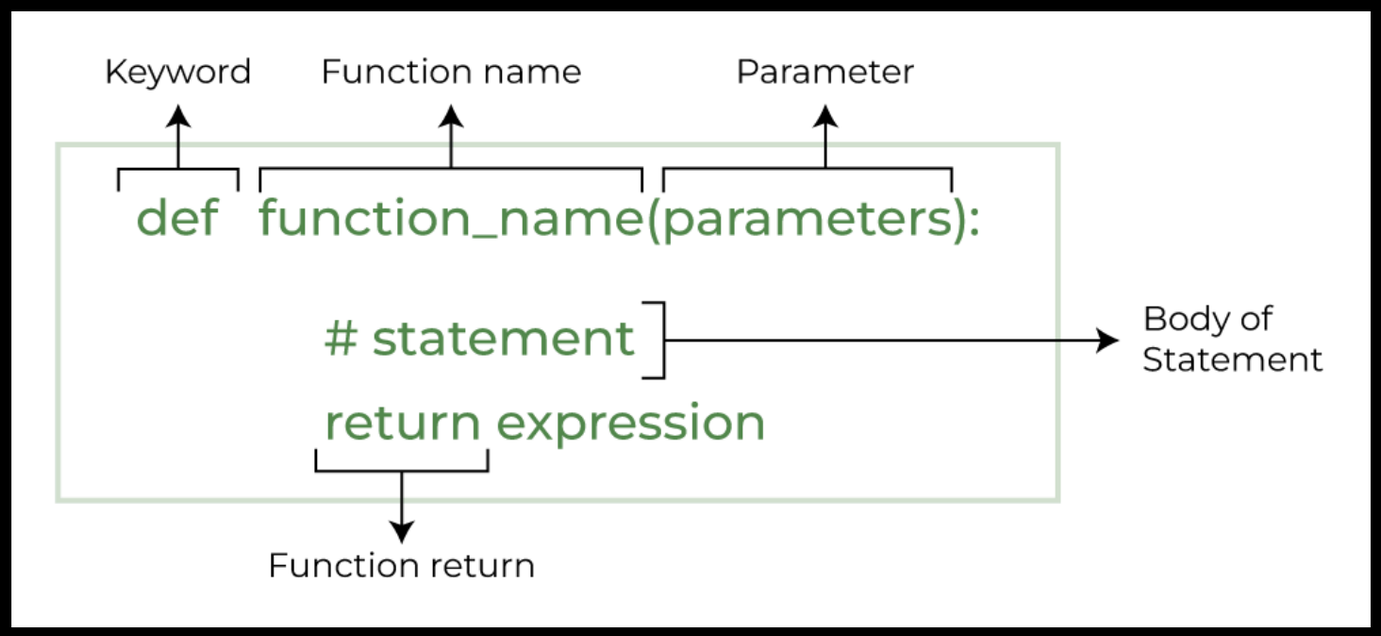
Functions in Python

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.

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




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

Calling a Function

```
my_function()
```

Arguments

Information can be passed into functions as arguments.

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

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

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

Arguments are often shortened to *args* in Python documentations.

Parameters or Arguments?

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

A parameter is a variable in a function definition.

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", “Refsnes")
```

Arbitrary Arguments, *args

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:

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

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

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

Arbitrary Arguments are often shortened to **args* in Python documentations.

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")
```

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

Arbitrary Keyword Arguments, **kwargs

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

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

Arbitrary Kword Arguments are often shortened to ***kwargs* in Python documentations.

Default Parameter Value

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

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

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Passing a List as an Argument

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

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

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

my_function(fruits)
```

Return Values

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

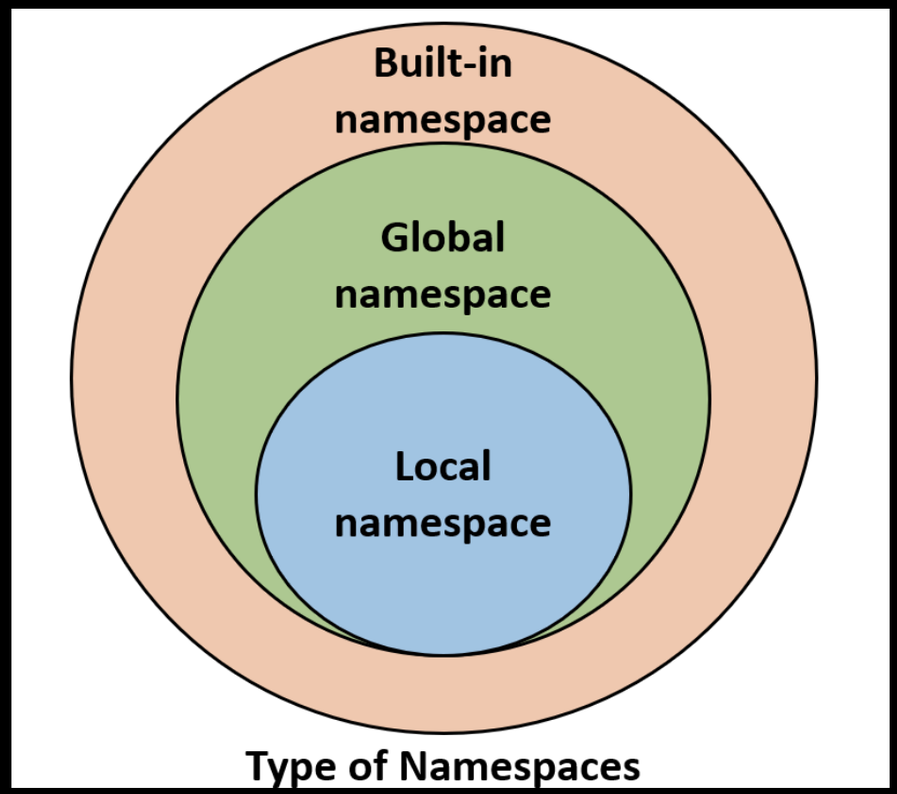
```
def my_function(x):
  return 5 * x

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

```
def add(num1, num2):
    """Add two numbers"""
    num3 = num1 + num2

    return num3

# Driver code
num1, num2 = 5, 15
ans = add(num1, num2)
print(f"The addition of {num1} and {num2} results {ans}.")
```

```
# A simple Python function to check
# whether x is even or odd
def evenOdd(x):
    if (x % 2 == 0):
        print("even")
    else:
        print("odd")

# Driver code to call the function
evenOdd(2)
evenOdd(3)
```

What is Namespace in Python?

A namespace is a system that has a unique name for each and every object in Python. An object might be a variable or a method.



Type of Namespaces

Namespace in Python

The TechPlatform

Built-in Namespace

- Created while starting the python interpreter, exists as long as interpreter runs.

Global Namespace

- Created when the program starts and exists until the program in terminated

Local Namespace

- Creates new namespace when function is executed

Enclosing Namespace

- For the Namespace of inner function or block of code.

Built-in Namespace

A built-in namespace contains the names of built-in functions and objects. It is created while starting the python interpreter, exists as long as the interpreter runs, and is destroyed when we close the interpreter.

```
builtin_names = dir(__builtins__)
for name in builtin_names:
    print(name)
```

The built-in namespace creates by the Python interpreter when its starts up. These are terminated when Python interpreter terminates.

Global Namespace

Global namespaces are defined at the program or module level. It contains the names of objects defined in a module or the main program. A global namespace is created when the program starts and exists until the program is terminated by the python interpreter.

```
# Python program processing
# global variable

count = 5
def some_method():
    global count
    count = count + 1
    print(count)
some_method()
```

Local Namespaces

The function uses the local namespaces; the Python interpreter creates a new namespace when the function is executed. The local namespaces remain in existence until the function terminates. The function can also consist of another function.

```
# var1 is in the global namespace
var1 = 5
def some_func():

    # var2 is in the local namespace
    var2 = 6
    def some_inner_func():

        # var3 is in the nested local
        # namespace
        var3 = 7
```

Enclosing namespace

As we know that we can define a block of code or a function inside another block of code or function, A function or a block of code defined inside any function can access the namespace of the outer function or block of code. Hence the outer namespace is termed as enclosing namespace for the namespace of the inner function or block of code.

Scope of Objects in Python :

Scope refers to the coding region from which a particular Python object is accessible. Hence one cannot access any particular object from anywhere from the code, the accessing has to be allowed by the scope of the object.

```
# Python program showing
# a scope of object

def some_func():
    print("Inside some_func")
    def some_inner_func():
        var = 10
        print("Inside inner function, value of var:",var)
    some_inner_func()
    print("Try printing var from outer function: ",var)
some_func()
```

Example 1: Scope and Namespace in Python

```
# global_var is in the global namespace
global_var = 10

def outer_function():
    #  outer_var is in the local namespace
    outer_var = 20

    def inner_function():
        #  inner_var is in the nested local namespace
        inner_var = 30

        print(inner_var)

    print(outer_var)

    inner_function()

# print the value of the global variable
print(global_var)

# call the outer function and print local and nested local variables
outer_function()
```

Example 2: Use of global Keyword in Python

```
# define global variable
global_var = 10

def my_function():
    # define local variable
    local_var = 20

    # modify global variable value
    global global_var
    global_var = 30

# print global variable value
print(global_var)

# call the function and modify the global variable
my_function()

# print the modified value of the global variable
print(global_var)
```

Here, when the function is called, the `global` keyword is used to indicate that `global_var` is a global variable, and its value is modified to **30**.

Python Lambda

A lambda function is a small anonymous function.

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

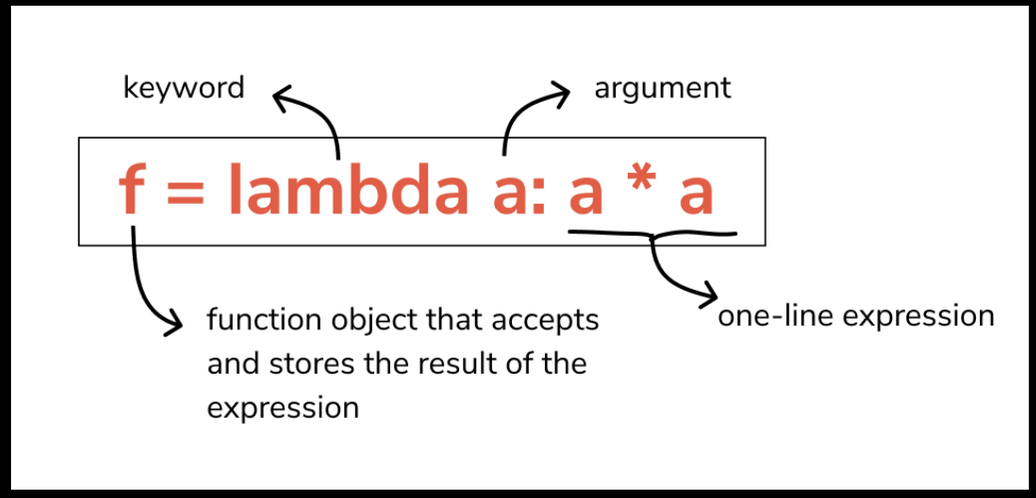
Syntax:

```
lambda arguments : expression
```

```
x = lambda a : a + 10
print(x(5))
```

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

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



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

mydoubler = myfunc(2)

print(mydoubler(11))
```

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

Use lambda functions when an anonymous function is required for a short period of time.

map() Function

map() function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

Syntax :

```
map(fun, iter)
```

fun : *It is a function to which map passes each element of given iterable.*

iter : *It is a iterable which is to be mapped.*

```
# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

We can also use lambda expressions with map to achieve above result.

```
numbers = (1, 2, 3, 4)
result = map(lambda x: x + x, numbers)
print(list(result))
```

```
# Add two lists using map and lambda

numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

result = map(lambda x, y: x + y, numbers1, numbers2)
print(list(result))
```

filter()

The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

Syntax: *filter(function, sequence)*

Parameters:

- ***function:*** *function that tests if each element of a sequence is true or not.*
- ***sequence:*** *sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.*

Returns: *an iterator that is already filtered.*

Filter the array, and return a new array with only the values equal to or above 18:

```
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
    return False
  else:
    return True

adults = filter(myFunc, ages)

for x in adults:
  print(x)
```

```python
# function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False


# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']

# using filter function
filtered = filter(fun, sequence)

print('The filtered letters are:')
for s in filtered:
    print(s)
```

```python
# a list contains both even and odd numbers.
seq = [0, 1, 2, 3, 5, 8, 13]

# result contains odd numbers of the list
result = filter(lambda x: x % 2 != 0, seq)
print(list(result))

# result contains even numbers of the list
result = filter(lambda x: x % 2 == 0, seq)
print(list(result))
```

reduce()

The reduce(fun,seq) function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along.This function is defined in “functools” module.

Working :

- At first step, first two elements of sequence are picked and the result is obtained.
- Next step is to apply the same function to the previously attained result and the number just succeeding the third element and the result is again stored.
- This process continues till no more elements are left in the container.
- The final returned result is returned and printed on console.

```
# python code to demonstrate working of reduce()

# importing functools for reduce()
import functools

# initializing list
lis = [1, 3, 5, 6, 2]

# using reduce to compute sum of list
print("The sum of the list elements is : ", end="")
print(functools.reduce(lambda a, b: a+b, lis))

# using reduce to compute maximum element from list
print("The maximum element of the list is : ", end="")
print(functools.reduce(lambda a, b: a if a > b else b, lis))
```

eval

Python eval() function parse the expression argument and evaluate it as a python expression and runs python expression (code) within the program.

Syntax: *eval(expression)*

Parameters:

- ***expression:*** *String is parsed and evaluated as a Python expression*

Return: *Returns output of the expression*

```
print(eval('1+2'))
```

```
expression = 'x*(x+1)*(x+2)'
print(expression)

x = 3

result = eval(expression)
print(result)
```

```
def function_creator():

    # expression to be evaluated
    expr = input("Enter the function(in terms of x):")

    # variable used in expression
    x = int(input("Enter the value of x:"))

    # evaluating expression
    y = eval(expr)

    # printing evaluated result
    print("y =", y)

if __name__ == "__main__":
    function_creator()
```

- The above function takes any expression in variable **x** as input.
- Then the user has to enter a value of **x**.
- Finally, we evaluate the python expression using **eval()** built-in function by passing the **expr** as an argument.

Generators in Python

A Generator in Python is a function that returns an iterator using the Yield keyword

Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.

```
# A generator function that yields 1 for first time,
# 2 second time and 3 third time
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

# Driver code to check above generator function
for value in simpleGeneratorFun():
    print(value)
```

Python Generator functions return a generator object that is iterable, i.e., can be used as an **Iterator**. Generator objects are used either by calling the next method of the generator object or using the generator object in a “for in” loop.

```
# A Python program to demonstrate use of
# generator object with next()

# A generator function
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

# x is a generator object
x = simpleGeneratorFun()

# Iterating over the generator object using next

# In Python 3, __next__()
print(next(x))
print(next(x))
print(next(x))
```