# Lambda Functions

**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*

**Example – 01**

```
a = lambda: 3    # with out parameters
def a(): return 3
```

**Example – 02**

```
a = lambda x, y: return x*y    #with parameters
def a(x, y): return x+y
```

## 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:

**Example – 03**

```
def myfunc(n):
      return lambda a : a * n
mydoubler = myfunc(2)  # mydoubler is also a function
print(mydoubler(11))   # returns 22
```

**Lambdas should only be used for very simple functions. If your lambda starts looking too complicated to be readable, you should rather write it out in full as a normal, named function.**

## Exercise

1. Define the following functions as lambdas, and assign them to variables:
    (a) Take one parameter; return its square
    (b) Take two parameters; return the square root of the sums of their squares
    (c) Take any number of parameters; return their average
    (d) Take a string parameter; return a string which contains the unique letters in the input string (in any order)
2. Rewrite all these functions as named functions.

## Solutions:

1. Here is an example program:

```
import math
a = lambda x: x**2
b = lambda x, y: math.sqrt(x**2 + y**2)
c = lambda *args: sum(args)/len(args)
d = lambda s: "".join(set(s))
```

2. Here is an example program:

```
import math
def a(x):
      return x**2
```

```
      def b(x, y):
            return math.sqrt(x**2 + y**2)
def c(*args):
      return sum(args)/len(args)
```

# Python program to illustrate *args for variable number of arguments

```
def myFun(*argv):
      for arg in argv:
      print (arg)
myFun('Hello', 'Welcome', 'to', 'Python', 'Course')
```

```
def d(s):
      return "".join(set(s))
```

# Generator Functions and Yield

New elements are generated as they are needed, instead of all being generated up-front.
We can create our own generators by writing functions which make use of the yield statement.

Consider this simple function which returns a range of numbers as a list:

```
def my_list(n):
      i=0
      l = []
      while i < n:
            l.append(i)
            i += 1 return l
```

This function builds the full list of numbers and returns it.
We can change this function into a generator function while preserving a very similar syntax, like this:

```
def my_gen(n):
      i=0
      while i < n:
            yield i
            i += 1
```

When we use it in a function, that function will return a generator.
We can test this by using the type function on the return value of my_gen. We can also try using it in a for loop, like we would use any other generator, to see what sequence the generator represents:

```
      g = my_gen(3)
      print(type(g))
      for x in g:
            print(x)
```

### What does the yield statement do?

Whenever a new value is requested from the generator, for example by our for loop in the example above, the generator begins to execute the function until it reaches the yield statement. The yield statement causes the generator to return a single value.

After the yield statement is executed, execution of the function does not end – when the *next* value is requested from the generator, it will go back to the beginning of the function and execute it *again*.

If the generator executes the entire function without encountering a yield statement, it will raise a StopIteration exception to indicate that there are no more values.

A for loop automatically handles this exception for us. In our my_gen function this will happen when i becomes equal to n – when this happens, the yield statement inside the while loop will no longer be executed.

### Exercise 8

1. Write a generator function which takes an integer n as a parameter. The function should return a generator which counts *down* from n to 0. Test your function using a for loop.

### Solution

```
def my_gen(n):
      i=n
      while i >= 0:
            yield i
            i -= 1

for x in my_gen(3):
print(x)
```