Department of Computer Science and Engineering
PES University, Bangalore, India

Python For Computational Problem Solving (UE22CS151A)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

---

lambda functions: Functions with no name

There are cases where the function body is just an expression and we may want to use it once and so we do not care to give a name for it. In such cases, we use what are called lambda functions – functions with no name.

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.

A lambda function has the following characteristics:

- It can only contain expressions and can't include statements in its body.
- It is written as a single line of execution.
- It does not support type annotations.
- It can be immediately invoked.

In [1]:
```
# Example 1:

lambda x : x * x      # this expression is callable.
print("res : ", (lambda x : x * x)(10))
```

```
res :  100
```

The lambda function above is defined and then immediately called with argument 10. It returns the value 100, which is the square of the argument.

A lambda function is an expression, it can be named. Therefore you could write the previous code as follows:

In [7]:
```
sq_fn = lambda x : x * x
print(sq_fn(5))
```

```
25
```

In [2]:
```
# Example 2:
f1 = lambda x, y : x * y
```

```
f2 = lambda x, y : x * y

print(f1, type(f1))
print(f2, type(f2))
print(f1(10, 20))
print(f2(10, 20))
```

```
<function <lambda> at 0x0000025BBFB9F790> <class 'function'>
<function <lambda> at 0x0000025BBFB9F700> <class 'function'>
200
200
```

In [3]:
```
# Example 3:
# Python code to illustrate cube of a number showing difference between def()and lambda(

def cube1(y):
    return y*y*y;

print(cube1(5))

cube2 = lambda x: x*x*x
print(cube2(7))
```

```
125
343
```

Arguments

Like a normal function object defined with def, Python lambda expressions support all the different ways of passing arguments. This includes:

- Positional arguments
- Named arguments (sometimes called keyword arguments)
- Variable list of arguments (often referred to as varargs)
- Variable list of keyword arguments
- Keyword-only arguments

In [3]:
```
(lambda x,y,z: x+y+z) (1,2,3)
```

Out[3]:
```
6
```

In [7]:
```
(lambda x,y,z: x+y+z) (1,z=3,y=2)
```

Out[7]:
```
6
```

In [ ]:
```
(lambda x,y=3,z=4: x+y+z) (1,2,3)
```

In [8]:
```
(lambda *args: sum(args)) (1,3,2)
```

Out[8]:
```
6
```

In [10]:
```
(lambda **kwargs: sum(kwargs.values())) (x=1,y=2,z=3)
```

Out[10]:
```
6
```

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:

In [ ]:
```
def myfunc(n):
  return lambda a : a * n
```

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

In [13]:
```
def MultiplyBy(n):
  return lambda a : a * n

double=MultiplyBy(2)
print(double(5))
triple=MultiplyBy(3)
print(triple(5))
times10=MultiplyBy(10)
print(times10(5))
```

```
10
15
50
```

The lambda function takes the value n from the parent function, so that in double the value of n is 2, in triple it is 3 and in times10 it is 10.

Lambda definition does not include a “return” statement, it always contains an expression which is returned. We can also put a lambda definition anywhere a function is expected, and we don’t have to assign it to a variable at all. This is the simplicity of lambda functions.

Lambda functions can be used along with built-in functions like filter(), map() and reduce().

Defining Function Again:

We know that a variable can be re-assigned giving a new value for it.

In [ ]:
```
# Example:
a = 10  # a is an int with the value 10
a = 20  # a is still an int with the value 20
```

What happens if we redefine a function?

The earlier definition gets replaced and from that point onwards, the new function manifests.

In [6]:
```
def foo():
    print("one")

foo() # one

# replaces the old function with the new one
def foo():
    print("two")

foo() # two
```

```
one
two
```

References: 1)

2)