Section A2 Function

User-defined Functions

```
def function_name(param1, param2, ....):
    """docstring"""
    action 1
    action 2
```

You can also choose to import just certain functions if you don't need all of the functions in the module by using from random import randint. For example:

```
from math import pow
print(pow(2,12))
```

having a **return** statement is not mandatory in a function and can be omitted if there is no necessity to return a value. A function that
omits return returns None.
in Python, we can also have multiple return values, not just one as we saw above? This is performed by simply specifying the values we need to return separated by commas after the return keyword.

Python has the **global** keyword, if you want to make a variable global. Therefore, you can use the global keyword in a function body to create a global variable to make it accessible outside of the function, as shown in the example below.

```
# example of creating a global variable
def function_name():
    global x
    x = 9

function_name()
print(x) #x is accessible outside the function
```

Side effects in functions

In [15]:

```python
def mystery(num):
    num = num+1
    print('inside mystery, num=',num)

num = 1
print('outside mystery, num=',num)
mystery(num)
print('outside mystery, num=',num)
```

Out[15]:

```
outside mystery, num= 1
inside mystery, num= 2
outside mystery, num= 1
```

If you pass an immutable type to a function, then you can only update it using assignment, '='. As the type itself is immutable, Python will make a new local variable for the result. Even if it uses the same name, the local variable will be used within the function and will only be available within the function. So when num = num+1 is run, a new variable called num, local to and only available in the function, is created. Any changes to this new variable do not affect any variables outside of the function (even if they have the same name).

Arguments

Default Arguments

you will have to arrange all your default parameters after non-default parameters.

```python
def construct_login_greeting(fname, lname, message = "Hello"):
    return message+" "+fname+" "+lname+"!"

print(construct_login_greeting("David", "Smith"))
print(construct_login_greeting("System Maintenance", "Team", "Welcome"))
```

Keyword Arguments/Named Arguments

In Python documentation, the phrase Keyword Arguments are often shortened to *kwargs*.
Python allows you to have *named arguments* (also known as *keyword arguments*) with the key = value syntax

```python
def construct_login_greeting(fname, lname, message = "Hello", login_type = "general user", detailed_message = "Welcome to the user's d
ashboard"):
    return message+" "+fname+" "+lname+"!"+"\n"+"Your login type is: "+login_type+"\n"+detailed_message

print(construct_login_greeting("David", "Smith", login_type = "privileged user"))
```

```
Hello David Smith!
Your login type is: privileged user
Welcome to the user's dashboard
```

you need to have the keyword arguments after the after positional arguments and before default arguments in a function call.

```
def greet(name, project, group=None):
    print(group, ": Welcome", name, "from", project)

greet("John", project = "XYZ")
```

```
None : Welcome John from XYZ
```

Variable Number of Arguments

- The single ***** will place all of the values into a *tuple*
- The second option (******) will place all of the values into a *dictionary*.

In Python documentation, the phrase Arbitrary Arguments are often shortened to **args*.

```
# example variable number of parameters
def sum_vals(*numbers):
    sum = 0
    for num in numbers:
        sum = sum + num
    return sum

print(sum_vals(1,2,3))
print(sum_vals(1,2,3,4,5))
```

```
6
15
```

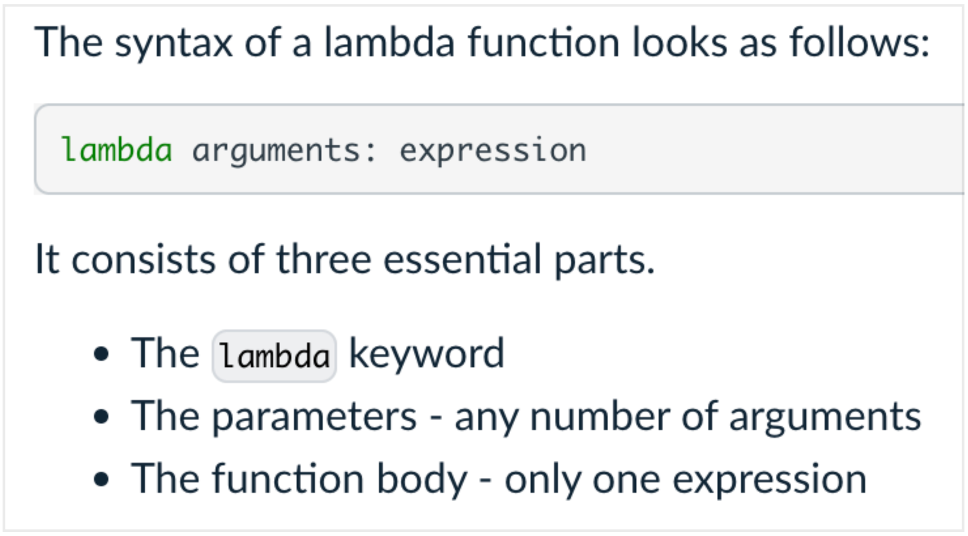
In Python documentation, the phrase Arbitrary Keyword Arguments are often shortened to **kargs*.

```
# example variable number of key=value parameters
def isInAdelaide(**customer):
    sum = 0
    if customer['postcode'] >= 5000 and customer['postcode'] < 6000:
        return True

if isInAdelaide(name='John',postcode=5126):
    print("John is in adelaide")

if isInAdelaide(name='John',postcode=5126, age=52):
        print("John is in adelaide")
```

```
John is in adelaide
John is in adelaide
```

Anonymous Functions

To define an anonymous function we use the **lambda** keyword. It is a small and restricted function that has no more than one line.

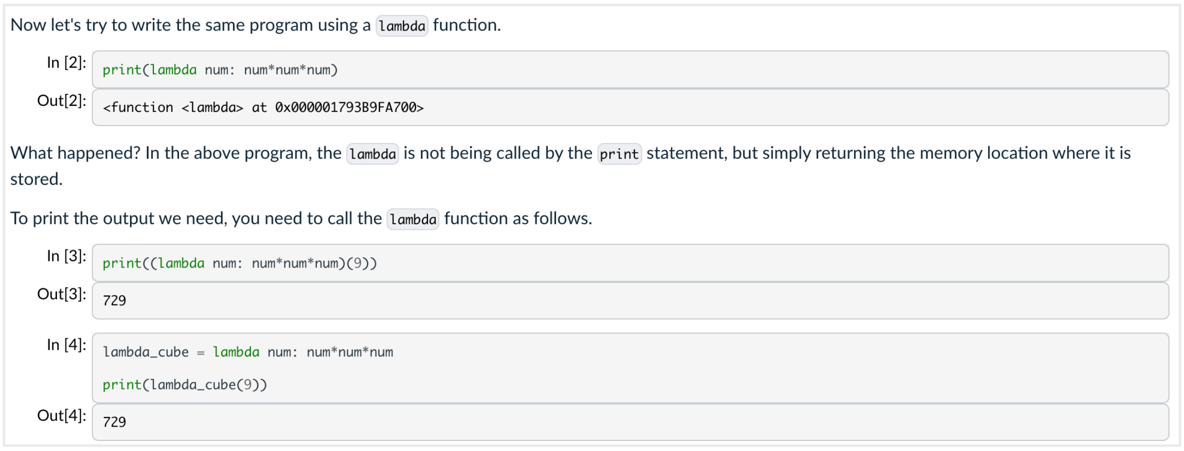


The syntax of a lambda function looks as follows:

```
lambda arguments: expression
```

It consists of three essential parts.

- The `lambda` keyword
- The parameters - any number of arguments
- The function body - only one expression



Now let's try to write the same program using a `lambda` function.

```
In [2]: print(lambda num: num*num*num)
Out[2]: <function <lambda> at 0x000001793B9FA700>
```

What happened? In the above program, the `lambda` is not being called by the `print` statement, but simply returning the memory location where it is stored.

To print the output we need, you need to call the `lambda` function as follows.

```
In [3]: print((lambda num: num*num*num)(9))
Out[3]: 729
```

```
In [4]: lambda_cube = lambda num: num*num*num

        print(lambda_cube(9))
Out[4]: 729
```

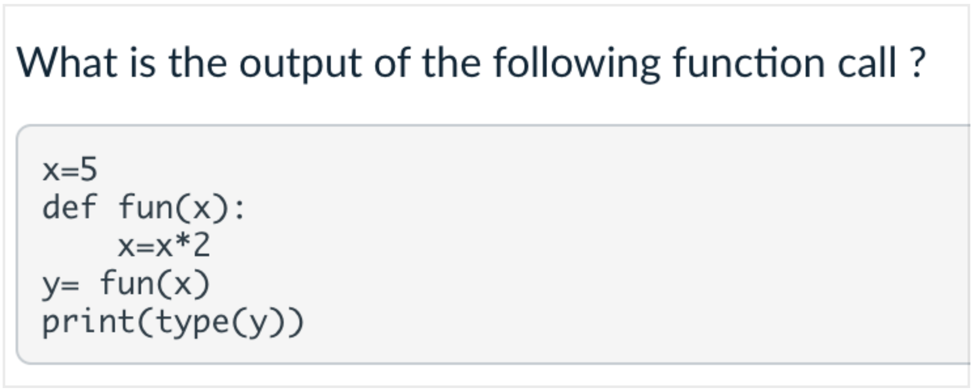
In the normal def function, we needed to return the result (from where the function was called) using the return keyword. However, in the lambda function definition, you do not need to include a return statement as it always contains an expression that is returned.
You can use if-else statements with lambda functions as follows.

```
max_value = lambda num1, num2 : num1 if(num1 &gt; num2) else num2
print(max_value(1, 2))
```

```
2
```

```
check_num = lambda x: 'Num is greater than 5' if x &gt; 5 else 'Num is less than 5'
print(check_num(9))
```

```
Num is greater than 5
```


What is the output of the following function call ?

```
x=5
def fun(x):
    x=x*2
y= fun(x)
print(type(y))
```

The function fun doesn't have a return statement. When a function in Python doesn't explicitly return a value, it returns None by default.