Python Programming Language

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

spring 2020

sw.eng.MohamedAyman@gmail.com

facebook.com/cs.MohamedAyman

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 6

Functions

Course Roadmap

Part 1: Python Basics and Functions

Lecture 1: Python Overview

Lecture 2: Variable Types

Lecture 3: Basic Operations

Lecture 4: Conditions

Lecture 5: Loops

Lecture 6: Functions

Lecture Agenda

We will discuss in this lecture the following topics

1- Function Definition
2- Calling a Function
3- Return Statement
4- Passing by Reference & Value
5- Function Arguments
6- Anonymous Function
7- Inner Functions
8- Global & Local Variables


Let's
STARTUP

Lecture Agenda

python

Section 1: Function Definition

Section 2: Calling a Function

Section 3: Return Statement

Section 4: Passing by Reference & Value

Section 5: Function Arguments

Section 6: Anonymous Function

Section 7: Inner Functions

Section 8: Global & Local Variables

Function Definition

- You can define function to provide the required functionality. Here are simple rules to define a function in Python.

```
def functionName(parameters):
    statement(s)
    return expression
```

- A function is a block of organized, reusable code that is used to perform a single, related action. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

- Functions provide better modularity for your application and a high degree of code reusing. As you already know, Python given you many built-in function like print(), etc. but you can also create your own functions. These functions are called user defined functions.

Function Definition

- Above shown is a function definition which consists of following components.
  1- Keyword def marks the start of function header.
  2- A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
  3- Parameters (arguments) through which we pass values to a function. They are optional.
  4- A colon (:) to mark the end of function header.
  5- Optional documentation string (docstring) to describe what the function does.
  6- One or more valid python statements that make up the function body.
  7- An optional return statement to return a value from the function.
- Any input parameters or argument 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 doctoring.
- The code blocks 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 argument is the same as return None.

Why Functions?

- It may not be clear why it is worth the trouble to divide a program into functions.

There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Lecture Agenda

- ✔ Section 1: Function Definition
- **Section 2: Calling a Function**
- Section 3: Return Statement
- Section 4: Passing by Reference & Value
- Section 5: Function Arguments
- Section 6: Anonymous Function
- Section 7: Inner Functions
- Section 8: Global & Local Variables

Calling a Function

- Defining a function gives it a name, specifies the parameters that are to be include in the function and structure the blocks of code.
- Once the basics structure of a function is finalized, you can execute it by calling it form another function or directly from the Python prompt.
- Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
def functionName():
    ... .. ...
    ... .. ...

... .. ...
... .. ...

functionName()

... .. ...
... .. ...
```

Calling a Function

Example:

```
# Function definition is here
def summation(x, y):
    res = x + y
    return res

# Now you can call summation function
r = summation(4, 3)
print(r)

a, b = 6, 8
r = summation(a, b)
print(r)
```

Output:

```
7

14
```

Calling a Function

Iterations Trace:

1-
```python
# Function definition is here
def summation(x, y):
    res = x + y
    return res

# Now you can call summation function
r = summation(4, 3)
print(r)

a, b = 6, 8
r = summation(a, b)
print(r)
```

2-
```python
# Now you can call summation function
r = summation(4, 3)
print(r)                    7
```

1.1-
```python
# Function definition is here
def summation(x, y):
    res = x + y
    return res
```

1.2-
```python
# Function definition is here
def summation(x, y):
    res = x + y
    return res
```

1.3-
```python
# Function definition is here
def summation(x, y):
    res = x + y
    return res
```

Calling a Function

Iterations Trace:

1-
```
a, b = 6, 8
r = summation(a, b)
print(r)
```

2-
```
a, b = 6, 8
r = summation(a, b)
print(r)
```

3-
```
a, b = 6, 8
r = summation(a, b)
print(r)                    14
```

2.1-
```
# Function definition is here
def summation(x, y):
    res = x + y
    return res
```

2.2-
```
# Function definition is here
def summation(x, y):
    res = x + y
    return res
```

2.3-
```
# Function definition is here
def summation(x, y):
    res = x + y
    return res
```

Calling a Function

- The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does.
- We have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as doc_attribute of the function.

Example:

```
def summation(x, y):
    """
    This function calculates the summation of two number
    parameter: (x, y) numbers which we will calculate the sum of them
    return: result number the summation of the given numbers
    """
    res = x + y
    return res

print(summation.__doc__)
```

Output:

```
This function calculates the summation of two number
parameter: (x, y) numbers which we will calculate the sum of them
return: result number the summation of the given numbers
```

Lecture Agenda

- ✓ Section 1: Function Definition
- ✓ Section 2: Calling a Function
- Section 3: Return Statement
- Section 4: Passing by Reference & Value
- Section 5: Function Arguments
- Section 6: Anonymous Function
- Section 7: Inner Functions
- Section 8: Global & Local Variables

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. Python support multiple return objects with different data types.
- This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.
- The return statement causes your function to exit and hand back a value to its caller. The point of functions in general is to take in inputs and return something. The return statement is used when a function is ready to return a value to its caller.

Return Statement

Example:

```
# Function definition is here
def add_100(x, y):
    x += 100
    y += '100'
    return x, y

# Now you can call summation function
a, b = 6, '8'
a, b = add_100(a, b)
print(a)
print(b)
```

Output:

```
106
8100
```

Quiz

- Which of the following statement is true according to python functions?

- [ ] Functions are used to create objects in Python.
- [ ] Functions make your program run faster.
- [ ] A function is a piece of code that performs a specific task.
- [ ] All of the above

Quiz Solution

- Which of the following statement is true according to python functions?

- [ ] Functions are used to create objects in Python.
- [ ] Functions make your program run faster.
- [x] A function is a piece of code that performs a specific task.
- [ ] All of the above

Quiz

- What is the output of the following code according to python functions?

```python
def printLine(text):
    print(text, 'is awesome.')

printLine('Python')
```

```python
def f1():
    x = 1
def f2():
    x = 1
    return
def f3():
    x = 1
    return None

print(f1(), f2(), f3())
```

- ☐ Python
- ☐ Python is awesome.
- ☐ text is awesome.
- ☐ is awesome.

- ☐ None None None
- ☐ 0 None None
- ☐ 0 0 None
- ☐ None 0 None

Quiz Solution

- What is the output of the following code according to python functions?

```python
def printLine(text):
    print(text, 'is awesome.')

printLine('Python')
```

- [ ] Python
- [x] Python is awesome.
- [ ] text is awesome.
- [ ] is awesome.

```python
def f1():
    x = 1
def f2():
    x = 1
    return
def f3():
    x = 1
    return None

print(f1(), f2(), f3())
```

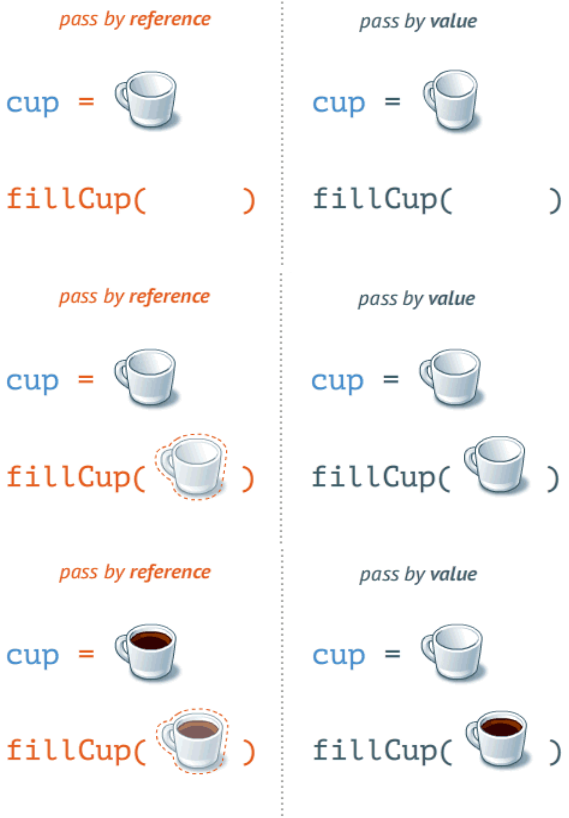
- [x] None None None
- [ ] 0 None None
- [ ] 0 0 None
- [ ] None 0 None

Lecture Agenda

- ✓ Section 1: Function Definition
- ✓ Section 2: Calling a Function
- ✓ Section 3: Return Statement
- **Section 4: Passing by Reference & Value**
- Section 5: Function Arguments
- Section 6: Anonymous Function
- Section 7: Inner Functions
- Section 8: Global & Local Variables

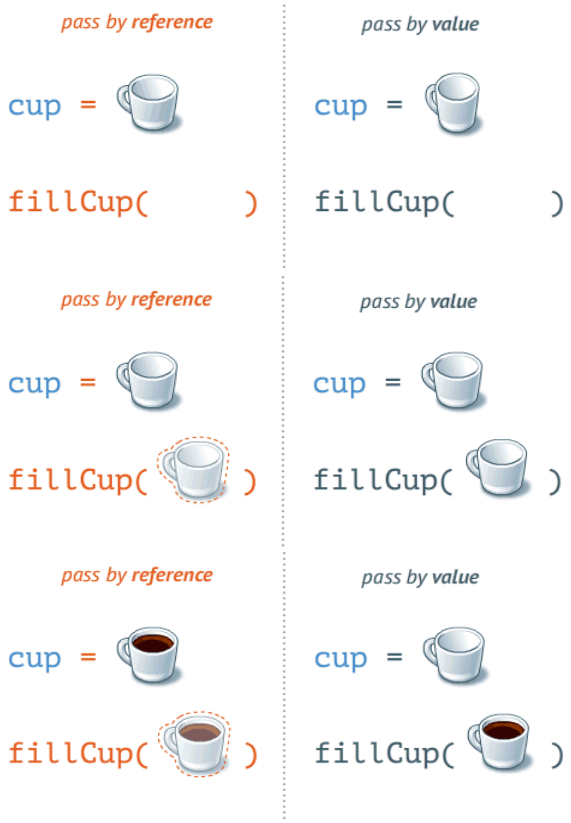
Passing by Reference & Value

- In call-by-value, the argument expression is evaluated, and the result of this evaluation is bound to the corresponding variable in the function. So, if the expression is a variable, a local copy of its value will be used, i.e. the variable in the caller's scope will be unchanged when the function returns.
- In call-by-reference evaluation, which is also known as pass-by-reference, a function gets an implicit reference to the argument, rather than a copy of its value. As a consequence, the function can modify the argument, i.e. the value of the variable in the caller's scope can be changed.



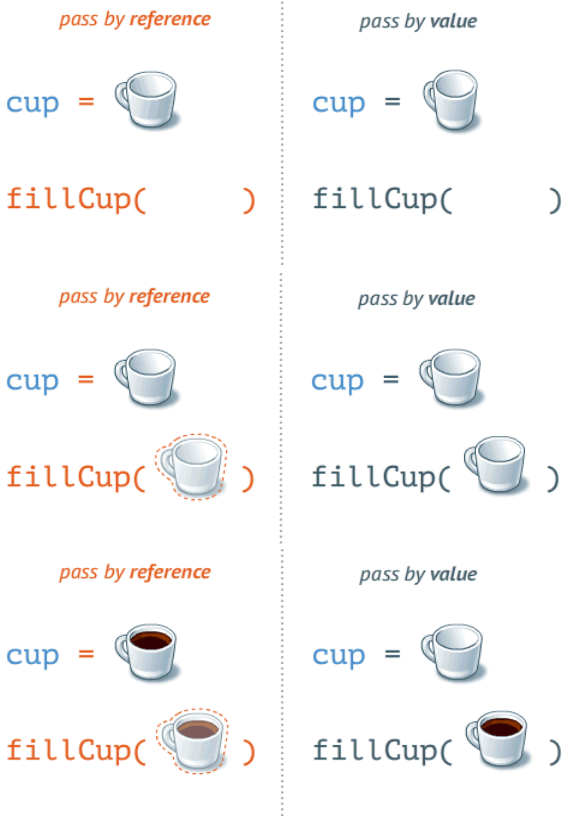
Passing by Reference & Value

- The advantage of call-by-reference consists in the advantage of greater time- and space-efficiency, because arguments do not need to be copied.
- On the other hand this harbors the disadvantage that variables can be "accidentally" changed in a function call. So special care has to be taken to "protect" the values, which shouldn't be changed.
- All parameters (arguments) in the Python language are passed by reference.
- Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing".



Passing by Reference & Value

- If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters.
- They can't be changed within the function, because they can't be changed at all, i.e. they are immutable. It's different, if we pass mutable arguments. They are also passed by object reference, but they can be changed in place in the function. If we pass a list to a function, we have to consider two cases:
- Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope. If a new list is assigned to the name, the old list will not be affected, i.e. the list in the caller's scope will remain untouched.



Passing by Reference & Value

Example:

```
def change_int(x):
    print(x)
    x += 11.5
    print(x)

y = 5
print(y)
change_int(y)
print(y)
```

Output:

```
5
16.5
5
5
```

```
second one
third one
first one
fourth one
```

- If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters.

Passing by Reference & Value

Example:

```
def change_str(x):
    print(x)
    x += 'cd'
    print(x)

y = 'ab'
print(y)
change_str(y)
print(y)
```

Output:

```
ab
abcd
ab
ab
```

```
second one
third one
first one
fourth one
```

- If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters.

Passing by Reference & Value

Example:

```
def change_tuple(x):
    print(x)
    x += (2.3, 'cd')
    print(x)

y = ('ab', 8)
print(y)
change_tuple(y)
print(y)
```

Output:

```
('ab', 8)                    second one
('ab', 8, 2.3, 'cd')         third one

('ab', 8)                    first one
('ab', 8)                    fourth one
```

- If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters.

Passing by Reference & Value

Example:

```
def change_list(x):
    print(x)
    x += [2.3, 'cd']
    print(x)

y = ['ab', 8]
print(y)
change_list(y)
print(y)
```

Output:

```
['ab', 8]                    second one
['ab', 8, 2.3, 'cd']         third one

['ab', 8]                    first one
['ab', 8, 2.3, 'cd']         fourth one
```

- All parameters (arguments) in the Python language are passed by reference.
- Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing".

Passing by Reference & Value

Example:

```
def change_dict(x):
    print(x)
    x['python'] = 3
    x['java'] = 10
    print(x)

y = {'c++': 17, 'python': 2}
print(y)
change_dict(y)
print(y)
```

Output:

```
{'c++': 17, 'python': 2}                second one
{'c++': 17, 'python': 3, 'java': 10}    third one
{'c++': 17, 'python': 2}                first one
{'c++': 17, 'python': 3, 'java': 10}    fourth one
```

- All parameters (arguments) in the Python language are passed by reference.
- Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing".

Passing by Reference & Value

Example:

```
def change_set(x):
    print(x)
    x |= {3.2, 'gp'}
    print(x)

y = {'ed', 9}
print(y)
change_set(y)
print(y)
```

Output:

```
{9, 'ed'}                    second one
{'gp', 9, 3.2, 'ed'}         third one

{9, 'ed'}                    first one
{'gp', 9, 3.2, 'ed'}         fourth one
```

- All parameters (arguments) in the Python language are passed by reference.
- Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing".

Lecture Agenda

- ✓ Section 1: Function Definition
- ✓ Section 2: Calling a Function
- ✓ Section 3: Return Statement
- ✓ Section 4: Passing by Reference & Value
- **Section 5: Function Arguments**
- Section 6: Anonymous Function
- Section 7: Inner Functions
- Section 8: Global & Local Variables

Function Arguments

Up until now functions had fixed number of arguments. In Python there are other ways to define a function which can take variable number of arguments.

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

- Required Argument
- Keyword Argument
- Default Argument
- Variable Length Argument

Function Arguments - Required Argument

- Required argument are the argument passed to a function in correct positional order. Here the number of argument in the function call should match exactly with the function definition.
- Required arguments are the mandatory arguments of a function. These argument values must be passed in correct number and order during function call. Below is a typical syntax for a required argument function.

```
def function_name(param1, param2, param3, ..., paramN):
    statement(s)
    return expression

function_name(p1, p2, p3, ..., pN)
```

Function Arguments - Required Argument

Example:

```python
def summation(x, y, z):
    res = x + y + z
    return res

r = summation(4, 7, 2)
print(r)

r = summation(4, 7, 2, 5)

r = summation(4, 7)
```

Output:

```
13

Traceback (most recent call last):
  File "main.py", line 8, in <module>
    r = summation(4, 7, 2, 5)
TypeError: summation() takes 3 positional
           arguments but 4 were given

Traceback (most recent call last):
  File "main.py", line 12, in <module>
    r = summation(4, 7)
TypeError: summation() missing 1 required
           positional argument: 'z'
```

Function Arguments - Keyword Argument

- Keyword argument are related to the function calls. When you use keyword arguments in a function call, the caller identifier 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 keyword provided to match the value with parameters.
- When calling functions in Python, you will often have to choose between using keyword arguments or positional arguments. Keyword arguments can often be used to make function calls more explicit. You will likely see keyword arguments quite a bit in Python.
- Python has a number of functions that take an unlimited number of positional arguments. These functions sometimes have arguments that can be provided to customize their functionality. Those arguments must be provided as named arguments to distinguish them from the unlimited positional arguments.

Function Arguments - Keyword Argument

Example:

```
def print_info(name, level, gpa, department):
    print('name:', name)
    print('level:', level)
    print('gpa:', gpa)
    print('department:', department)

print_info(department='CS', name='Mark', gpa=3.7, level=3)
print_info(level=4, department='IT', name='Peter', gpa=3.4)
print_info(gpa=3.5, name='Alan', level=2, department='IS')
```

Output:

```
name: Mark
level: 3
gpa: 3.7
department: CS
```

```
name: Peter
level: 4
gpa: 3.4
department: IT
```

```
name: Alan
level: 2
gpa: 3.5
department: IS
```

Function Arguments - Default Argument

- Function arguments can have default values in Python, We can provide a default value to an argument by using the assignment operator (=).
- 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 givens an idea on default arguments, it prints default age if it is not passed.
- Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values. This means to say, non-default arguments cannot follow default arguments.

```
def function_name(param1, param2, param3=value, ..., paramN=value):
    statement(s)
    return expression

function_name(p1, p2, p3, ..., pN)
function_name(p1, p2)
```

Function Arguments - Default Argument

Example:

```
def summation(x, y=2, z=6):
    res = x + y + z
    return res

r = summation(4, 7, 2)
print(r)

r = summation(4, 7)
print(r)

r = summation(4)
print(r)
```

Output:

```
13

17

12
```

Function Arguments - Variable Length Argument

- Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.
- You may need to process a function for more arguments than you specified while defining the function. These argument are called variable-length argument and are not named in the function definition, unlike required and default argument.
- An asterisk (*) is placed before the variable name that holds the values of all non keyword variable arguments. This tuple remains empty if no additional argument are specified during the function call.

```
def function_name(*param):
    statement(s)
    return expression

function_name()
function_name(p1)
function_name(p1, p2)
function_name(p1, p2, p3)
function_name(p1, p2, p3, ..., pN)
```

Function Argument - Variable Length Argument

Example:

```
def summation(*x):
    print(len(x))
    print(type(x))
    print(x)
    res = 0
    for i in x:
        res += i
    return res

r = summation()
print(r)
r = summation(4)
print(r)
r = summation(4, 7)
print(r)
r = summation(4, 7, 2)
print(r)
```

Output:

```
<class 'tuple'>

0
4
11
13
```

Practice

Problem: Prime number

- Implement a function which checks whether a given number is prime or not using python functions, such that the input number data type will be integer and positive.
- A natural number (1, 2, 3, 4, 5, 6, etc.) is called a prime number (or a prime) if it is greater than 1 and cannot be written as the product of two smaller natural numbers.
- Test Cases:

Test Case 1	Test Case 2	Test Case 3	Test Case 4
3	8	11	9
prime	not prime	prime	not prime

Problem Solution

- A natural number (1, 2, 3, 4, 5, 6, etc.) is called a prime number (or a prime) if it is greater than 1 and cannot be written as the product of two smaller natural numbers.

```
def is_prime(x):
    if x < 2:
        return False
    for i in range(2, x):
        if x % i == 0:
            return False
    return True

n = int(input())
print('prime' if is_prime(n) else 'not prime')
```

Problem: Min and Max functions

- Implement two functions min, max which takes at least two numbers or more using python functions (variable length argument) and return the best value (minimum value in function min and maximum value in function max).

- Test Cases:

```
print(calc_min(7, 4))                    4
print(calc_min(7, 4, 8))                 4
print(calc_min(7, 4, 8, 3))              3
print(calc_min(7, 4, 8, 3, 1))           1

print(calc_max(7, 4))                    7
print(calc_max(7, 4, 8))                 8
print(calc_max(7, 4, 8, 3))              8
print(calc_max(7, 4, 8, 3, 11))          11
```

Problem Solution

- Implement two functions min, max which takes at least two numbers or more using python functions (variable length argument) and return the best value (minimum value in function min and maximum value in function max).

```python
def calc_min(x, y, *z):
    res = x if x < y else y
    for i in z:
        if res > i:
            res = i
    return res

def calc_max(x, y, *z):
    res = x if x > y else y
    for i in z:
        if res < i:
            res = i
    return res
```

Lecture Agenda

- ✓ Section 1: Function Definition
- ✓ Section 2: Calling a Function
- ✓ Section 3: Return Statement
- ✓ Section 4: Passing by Reference & Value
- ✓ Section 5: Function Arguments
- **Section 6: Anonymous Function**
- Section 7: Inner Functions
- Section 8: Global & Local Variables

Anonymous Function

- These function 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 variable other than those in their parameter list and those in the global namespace.
- In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments).
- The Syntax of lambda function contains only a single statement, which is as follows

```
function_name = lambda argument(s) : expression
```

Anonymous Function

Example:

```
summation = lambda x, y, z : x + y + z

r = summation(5, 11, 7)
print(r)
r = summation(y = 5, z = 11, x = 7)
print(r)

summation = lambda x, y = 3, z = 2 : x + y + z

r = summation(5, 11, 7)
print(r)
r = summation(5, 11)
print(r)
r = summation(5)
print(r)
```

Output:

```
23
23

23
18
10
```

Use of Lambda Function

- We use lambda functions when we require a nameless function for a short period of time. In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.
- The filter() function in Python takes in a function and a list as arguments. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.
- The map() function in Python takes in a function and a list. The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Use of a General Function

Example:

Output:

```python
# Program to filter out only the even items from a list
def filter_fun(x):
    return x % 2 == 0

y = [1, 4, 6, 5, 8]
r = list(filter(filter_fun, y))
print(r)
```

```
[4, 6, 8]
```

```python
# Program to double each item in a list using map()
def map_fun(x):
    return x * 2

y = [1, 5, 4, 6, 8]
r = list(map(map_fun, y))
print(r)
```

```
[2, 10, 8, 12, 16]
```

Use of Lambda Function

Example:

```
# Program to filter out only the even items from a list

y = [1, 4, 6, 5, 8]
r = list(filter(lambda x: (x%2 == 0) , y))
print(r)
```

Output:

```
[4, 6, 8]
```

```
# Program to double each item in a list using map()

y = [1, 5, 4, 6, 8]
r = list(map(lambda x: x * 2 , y))
print(r)
```

```
[2, 10, 8, 12, 16]
```

Quiz

- What is the output of the following program according to python functions?

```python
def greetPerson(*name):
    print('Hello', name)

greetPerson('Frodo', 'Sauron')
```

```python
result = lambda x: x * x

print(result(5))
```

- [ ] Hello Frodo
  Hello Sauron
- [ ] Hello ('Frodo', 'Sauron')
- [ ] Hello Frodo
- [ ] Syntax Error! greetPerson() can take only one argument.
- [ ] `lambda x: x*x`
- [ ] 10
- [ ] 25
- [ ] 5*5

Quiz Solution

- What is the output of the following program according to python functions?

```python
def greetPerson(*name):
    print('Hello', name)

greetPerson('Frodo', 'Sauron')
```

```python
result = lambda x: x * x

print(result(5))
```

- [ ] Hello Frodo
  Hello Sauron
- [x] Hello ('Frodo', 'Sauron')
- [ ] Hello Frodo
- [ ] Syntax Error! greetPerson() can take only one argument.

- [ ] lambda x: x*x
- [ ] 10
- [x] 25
- [ ] 5*5

Practice

Problem: Distance between two points

- Implement a function which calculates the distance between two points given x, y for each point using python functions (anonymous function), such that the input number data type will be float and real number.

Hint: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Test Case 1	Test Case 2	Test Case 3	Test Case 4
0 0	0 0	3 2	-5 7
3 4	-6 -2	-4 -1	4 -3
5.0	6.32	7.62	13.45

Problem Solution

- Implement a function which calculates the distance between two points given x, y for each point using python functions (anonymous function), such that the input number data type will be float and real number.

Hint: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Test Case 1	Test Case 2	Test Case 3	Test Case 4
0 0	0 0	3 2	-5 7
3 4	-6 -2	-4 -1	4 -3
5.0	6.32	7.62	13.45

```
dist = lambda x1, y1, x2, y2 : ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5

x1, y1 = map(float, input().split())
x2, y2 = map(float, input().split())
print(dist(x1, y1, x2, y2))
```

Problem: Summation range

- Implement a function which calculates the summation from x to y inclusive using python functions (anonymous function), such that the input number data type will be integer and positive.
- Test Cases:

Test Case 1	Test Case 2	Test Case 3	Test Case 4
1 5	1 100	3 7	8 12
15	5050	25	50

Problem Solution

- Implement a function which calculates the summation from x to y inclusive using python functions (anonymous function), such that the input number data type will be integer and positive.
- Test Cases:

Test Case 1	Test Case 2	Test Case 3	Test Case 4
1 5	1 100	3 7	8 12
15	5050	25	50

```
summation = lambda n : n * (n+1) // 2

x, y = map(int, input().split())
print(summation(y) - summation(x-1))
```

Lecture Agenda

- ✓ Section 1: Function Definition
- ✓ Section 2: Calling a Function
- ✓ Section 3: Return Statement
- ✓ Section 4: Passing by Reference & Value
- ✓ Section 5: Function Arguments
- ✓ Section 6: Anonymous Function
- **Section 7: Inner Functions**
- Section 8: Global & Local Variables

Inner Functions

- A function which is defined inside another function is known as nested function. Nested functions are able to access variables of the enclosing scope. In Python, these non-local variables can be accessed only within their scope and not outside their scope.
- Functions are one of the "first-class citizens" of Python, which means that functions are at the same level as other Python objects like integers, strings, modules, etc. They can be created and destroyed dynamically, passed to other functions, returned as values, etc.
- Python supports the concept of a "nested function" or "inner function", which is simply a function defined inside another function. In the rest of the article, we will use the word "inner function" and "nested function" interchangeably.
- There are various reasons as to why one would like to create a function inside another function. The inner function is able to access the variables within the enclosing scope. In this article, we will be exploring various aspects of inner functions in Python.

Inner Functions

- To define an inner function in Python, we simply create a function inside another function using the Python's def keyword.

Example:

```
def outer_function():
    print ("Hello from outer function")

    def inner_function():
        print ("Hello from inner function")

    inner_function()

outer_function()
```

- In the above example, inner-function() has been defined inside outer-function(), making it an inner function. To call inner-function(), we must first call outer-function(). The outer-function() will then go ahead and call inner-function() as it has been defined inside it.

Inner Functions

Iterations Trace:

1-
```python
def outer_function():
    print ("Hello from outer function")

    def inner_function():
        print ("Hello from inner function")

    inner_function()

outer_function()
```

2-
```python
def outer_function():
    print ("Hello from outer function")

    def inner_function():
        print ("Hello from inner function")

    inner_function()

outer_function()
```

3-
```python
def outer_function():
    print ("Hello from outer function")

    def inner_function():
        print ("Hello from inner function")

    inner_function()

outer_function()
```

4-
```python
def outer_function():
    print ("Hello from outer function")

    def inner_function():
        print ("Hello from inner function")

    inner_function()

outer_function()
```

Inner Functions

Iterations Trace:

1-
```python
def outer_function():
    print ("Hello from outer function")

    def inner_function():
        print ("Hello from inner function")

    inner_function()

outer_function()
```

2-
```python
def outer_function():
    print ("Hello from outer function")

    def inner_function():
        print ("Hello from inner function")

    inner_function()

outer_function()
```

3-
```python
def outer_function():
    print ("Hello from outer function")

    def inner_function():
        print ("Hello from inner function")

    inner_function()

outer_function()
```

4-
```python
def outer_function():
    print ("Hello from outer function")

    def inner_function():
        print ("Hello from inner function")

    inner_function()

outer_function()
```

When and why Inner Functions?

- A function can be created as an inner function in order to protect it from everything that is happening outside of the function. In that case, the function will be hidden from the global scope. You use inner functions to protect them from everything happening outside of the function, meaning that they are hidden from the global scope.
- As closures are used as callback functions, they provide some sort of data hiding. This helps us to reduce the use of global variables. When we have few functions in our code, closures prove to be efficient way. But if we need to have many functions.
- The use of closures and factory functions is the most common and powerful use for inner functions. In most cases, when you see a decorated function, the decorator is a factory function that takes a function as argument and returns a new function that includes the old function inside the closure. Stop. Take a deep breath. Grab a coffee. Read that again.

Inner Functions

Example:

```python
def outer_function(x):

    def secret_increment(x):
        return x + 2

    y = secret_increment(x)
    print(x)
    print(y)

secret_increment(5)
```

Output:

```
Traceback (most recent call last):
  File "main.py", line 10, in <module>
    secret_increment(5)
NameError: name 'secret_increment' is not defined
```

- In the above code, we are trying to call the secret-increment() function, but instead we got an error.

Inner Functions

Example:

```
def outer_function(x):

    def secret_increment(x):
        return x + 2

    y = secret_increment(x)
    print(x)
    print(y)

outer_function(5)
```

Output:

```
5
7
```

- The script above shows that the inner function, that is, secret-increment() is protected from what is happening outside it since the variable x inside the secret-increment function is not affected by the value passed to the parameter x of the outer function. In other words, the variables inside the inner function is not accessible outside it.

When to use closures?

- A closure simply causes the inner function to remember the state of its environment when called. Beginners often think that a closure is the inner function, but it's really caused by the inner function. The closure "closes" the local variable on the stack, and this stays around after the stack creation has finished executing.

- We have a closure in Python when a nested function references a value in its enclosing scope. The criteria that must be met to create closure in Python are summarized in the following points.

  - We must have a nested function (function inside a function).
  - The nested function must refer to a value defined in the enclosing function.
  - The enclosing function must return the nested function.

- Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem.

Closure Function

Example:

```
def power_generator(n):

    def power(p):
        return n ** p

    return power

p2 = power_generator(2)
p3 = power_generator(3)

print(p2)
print(p3)
print(type(p2))
print(type(p3))

print(p2(5))
print(p3(4))
```

Output:

```
<function power_generator.<locals>.power at 0x7fca9725c3b0>
<function power_generator.<locals>.power at 0x7fca9725c440>
<class 'function'>
<class 'function'>

32
81
```

Lecture Agenda

- ✓ Section 1: Function Definition
- ✓ Section 2: Calling a Function
- ✓ Section 3: Return Statement
- ✓ Section 4: Passing by Reference & Value
- ✓ Section 5: Function Arguments
- ✓ Section 6: Anonymous Function
- ✓ Section 7: Inner Functions
- Section 8: Global & Local Variables

Global & Local Variables

- All variables in a program may not be accessible at all locations in that program, this depend on where you have declared a variable. The scope of a variable determine the portion of the program where you can access a particular identifier.
- There are two basic scopes of variable in Python: Global Variable - Local Variable
  Variable 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, where as 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.
- In Python, global keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

Global Variables

- In Python, a variable declared outside of the function or in global scope is known as global variable. This means, global variable can be accessed inside or outside of the function.

Example:

```
x = 3

def print_info():
    print(x)

print_info()
print(x)
```

Output:

```
3

3
```

- In above code, we created x as a global variable and defined a print-info() to print the global variable x. Finally, we call the print-info() which will print the value of x.

Global Variables

Example:

```
x = 3

def print_info():
    x *= 2

    print(x)

print_info()
print(x)
```

Output:

```
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    print_info()
  File "main.py", line 4, in print_info
    x*=2
UnboundLocalError: local variable 'x' referenced before assignment
```

- The output shows an error because Python treats x as a local variable and x is also not defined inside print-info().

Global Variables

- We use global keyword to read and write a global variable inside a function.

Example:

```
x = 3

def print_info():
    global x
    x *= 2
    print(x)

print_info()
print(x)
```

Output:

```
6
6
```

- In the above program, we define x as a global keyword inside the print-info() function. Then, we update the variable x by x * 2, i.e x = x * 2. After that, we call the print-info() function. Finally, we print global variable x. As we can see, change also occurred on the global variable outside the function, x = 6.

Local Variables

- A variable declared inside the function's body or in the local scope is known as local variable.

Example:

```
def print_info():
    x = 3
    print(x)

print_info()
print(x)
```

Output:

```
3

Traceback (most recent call last):
  File "main.py", line 6, in <module>
    print(x)
NameError: name 'x' is not defined
```

- The output shows an error, because we are trying to access a local variable x in a global scope whereas the local variable only works inside print-info() or local scope.

Global variable & Local variable

Example:

```
x = 5

def print_info():
    x = 10
    print(x)

print_info()
print(x)
```

Output:

```
10

5
```

- In above code, we used same name x for both global variable and local variable. We get different result when we print same variable because the variable is declared in both scopes, i.e. the local scope inside print-info() and global scope outside print-info().
- When we print the variable inside the print-info() it outputs local x = 10, this is called local scope of variable. Similarly, when we print the variable outside the print-info(), it outputs global x = 5, this is called global scope of variable.

Nonlocal variable

- Nonlocal variable are used in nested function whose local scope is not defined. This means, the variable can be neither in the local nor the global scope.
- Let's see an example on how a global variable is created in Python. We use nonlocal keyword to create nonlocal variable.
- Example:

```
def outer():
    x = 3

    def inner():
        nonlocal x
        x = 7
        print(x)                7

    inner()
    print(x)                    7

outer()
```

Quiz

- What is the output of the following program according to python functions?

```python
def display(x):
    print(x + 1)

x = -2
x = 4
display(12)
```

```python
x = 12
def display(a, b=x):
    print(a, b)

x = 15
display(4)
```

- ☐ 13
- ☐ 10
- ☐ 2
- ☐ 5

- ☐ 12 4
- ☐ 4 12
- ☐ 4 15
- ☐ none of the above

Quiz Solution

- What is the output of the following program according to python functions?

```python
def display(x):
    print(x + 1)

x = -2
x = 4
display(12)
```

```python
x = 12
def display(a, b=x):
    print(a, b)

x = 15
display(4)
```

- [x] 13
- [ ] 10
- [ ] 2
- [ ] 5

- [ ] 12 4
- [x] 4 12
- [ ] 4 15
- [ ] none of the above

Quiz

- What is the output of the following program according to python functions?

```python
def f(p, q, r):
    global s, q
    p, q, r, s = 10, 20, 30, 40
    print(p, q, r, s)

p, q, r, s = 1, 2, 3, 4
f(5, 10, 15)
```

- [ ] 1 2 3 4
- [ ] 5 10 15 4
- [ ] 10 20 30 40
- [ ] 5 10 15 40

```python
x = 5
def f1():
    global x
    x = 4
def f2(a, b):
    global x
    return a + b + x

f1()
total = f2(1, 2)
print(total)
```

- [ ] 7
- [ ] 8
- [ ] 15
- [ ] none of the above

Quiz Solution

- What is the output of the following program according to python functions?

```python
def f(p, q, r):
    global s, q
    p, q, r, s = 10, 20, 30, 40
    print(p, q, r, s)

p, q, r, s = 1, 2, 3, 4
f(5, 10, 15)
```

```python
x = 5
def f1():
    global x
    x = 4
def f2(a, b):
    global x
    return a + b + x

f1()
total = f2(1, 2)
print(total)
```

- [ ] 1 2 3 4
- [ ] 5 10 15 4
- [x] 10 20 30 40
- [ ] 5 10 15 40

- [x] 7
- [ ] 8
- [ ] 15
- [ ] none of the above

Practice

Problem: Computing the number of days in a month

- You will write a function called days_in_month that takes two integers: a year and a month. The function should return the number of days in that month. You may assume that both inputs are valid (in other words, you do not need to write any code to check whether or not they are reasonable).
- Test Cases:

Test Case 1	Test Case 2	Test Case 3	Test Case 4
2000 3	2010 6	2100 2	2400 2
31	30	28	29

Problem Solution

- You will write a function called days_in_month that takes two integers: a year and a month. The function should return the number of days in that month. You may assume that both inputs are valid (in other words, you do not need to write any code to check whether or not they are reasonable).

```python
def days_in_month(year, month):
    if month in [1, 3, 5, 7, 8, 10, 12]:
        return 31
    elif month in [4, 6, 9, 11]:
        return 30
    else:
        if year % 4 == 0 and year % 100 != 0 or year % 400 == 0:
            return 29
        else:
            return 28

print(days_in_month(2000, 3))
print(days_in_month(2010, 6))
print(days_in_month(2100, 2))
print(days_in_month(2400, 2))
```

Problem: Checking if a date is valid

- You will write a function called is_valid_date that takes three integers: a year, a month, and a day. The function should return True if that date is valid and False otherwise. This function should not assume that the inputs are valid. Rather, this function should check that all three inputs combine to form a valid date, with a month between 1 and 12, and a day between 1 and the number of days in the given month. Notice that the function days_in_month that you wrote for Part 1 will be useful here!

- Test Cases:

Test Case 1	Test Case 2	Test Case 3	Test Case 4
2000 2 29	2010 4 31	2020 8 32	2100 2 29
True	False	False	False

Problem Solution

- You will write a function called is_valid_date that takes three integers: a year, a month, and a day. The function should return True if that date is valid and False otherwise. This function should not assume that the inputs are valid. Rather, this function should check that all three inputs combine to form a valid date, with a month between 1 and 12, and a day between 1 and the number of days in the given month. Notice that the function days_in_month that you wrote for Part 1 will be useful here!

```
def is_valid_date(year, month, day):
    return 1 <= year <= 9999 and \
           1 <= month <= 12 and \
           1 <= day <= days_in_month(year, month)


print(is_valid_date(2000, 2, 29))
print(is_valid_date(2010, 4, 31))
print(is_valid_date(2020, 8, 32))
print(is_valid_date(2100, 2, 29))
```

Problem: Comparing two dates

- You will write a function called compare_dates that takes six integers: a year, a month, and a day of two dates. The function should return -1 if that date 1 is less that date 2 and return 1 if that date 1 is greater that date 2 and 0 if that two dates are equal. This function should assume that the inputs are valid.
- Test Cases:

Test Case 1	Test Case 2	Test Case 3
2010 1 1 2011 1 1	2000 7 31 2000 4 30	2000 7 31 2000 7 31
-1	1	0

Problem Solution

- You will write a function called compare_dates that takes six integers: a year, a month, and a day of two dates. The function should return -1 if that date 1 is less that date 2 and return 1 if that date 1 is greater that date 2 and 0 if that two dates are equal. This function should assume that the inputs are valid.

```
def compare_dates(year1, month1, day1, year2, month2, day2):
    if year1 != year2:
        return 1 if year1 > year2 else -1
    if month1 != month2:
        return 1 if month1 > month2 else -1
    if day1 != day2:
        return 1 if day1 > day2 else -1
    return 0

print(compare_dates(2010, 1, 1,  2011, 1,  1 ))
print(compare_dates(2000, 7, 31, 2000, 4,  30))
print(compare_dates(2010, 1, 1,  2010, 1,  1 ))
```

Problem: Calculate the next day

- You will write a function called next_date that takes three integers: a year, a month, and a day of a date. The function should return three integers: a year, a month, and a day that represent the next date of the given data. This function should assume that the inputs are valid.
- Test Cases:

Test Case 1	Test Case 2	Test Case 3	Test Case 4
2010 1 1	2010 1 31	2100 2 28	2010 12 31
2010 1 2	2010 2 1	2100 3 1	2011 1 1

Problem Solution

- You will write a function called next_date that takes three integers: a year, a month, and a day of a date. The function should return three integers: a year, a month, and a day that represent the next date of the given data. This function should assume that the inputs are valid.

```
def next_date(year, month, day):
    day += 1
    if day > days_in_month(year, month):
        day = 1
        month += 1
    if month == 13:
        month = 1
        year += 1
    return year, month, day

print(next_date(2010, 1, 1))
print(next_date(2010, 1, 31))
print(next_date(2100, 2, 28))
print(next_date(2010, 12, 31))
```

Problem: Computing the number of days between two dates

- You will write a function called days_between that takes six integers (year1, month1, day1, year2, month2, day2) and returns the number of days from an earlier date (year1-month1-day1) to a later date (year2-month2-day2). If either date is invalid, the function should return 0. Notice that you already wrote a function to determine if a date is valid or not! If the second date is earlier than the first date, the function should also return 0.

- Test Cases:

Test Case 1

```
2010 1 1 2011 1 1
365
```

Test Case 2

```
2010 1 1 2000 12 31
0
```

Test Case 3

```
2000 4 30 2000 7 31
92
```

Test Case 4

```
2020 4 31 2020 7 31
0
```

Problem Solution

```
def days_between(y1, m1, d1, y2, m2, d2):
    if not is_valid_date(y1, m1, d1) or \
       not is_valid_date(y2, m2, d2):
       return 0

    cur_y, cur_m, cur_d = y1, m1, d1
    n_days = 0
    while compare_dates(cur_y, cur_m, cur_d, y2, m2, d2) == -1:
        n_days += 1
        cur_y, cur_m, cur_d = next_date(cur_y, cur_m, cur_d)

    return n_days

print(days_between(2010, 1, 1,  2011, 1,  1 ))
print(days_between(2000, 4, 30, 2000, 7,  31))
print(days_between(2010, 1, 1,  2000, 12, 31))
print(days_between(2020, 4, 31, 2020, 7,  31))
```

Problem: Convert decimal to binary and vice versa

- Implement two functions which convert binary to decimal and decimal to binary using python functions, such that the input number data type will be integer and positive in case of convert decimal to binary and it will be string in case of convert binary to decimal.
- Test Cases:

Test Case 1	Test Case 2	Test Case 3	Test Case 4
15	17	6	24
1111	10001	110	11000

Test Case 1	Test Case 2	Test Case 3	Test Case 4
1111	10001	110	11000
15	17	6	24

Problem Solution

- Function convert decimal to binary

```
def convert_decimal_to_binary(x):
    res = ''
    while x > 0:
        res = chr(x%2 + ord('0')) + res
        x //= 2
    return res

n = int(input())
print(convert_decimal_to_binary(n))
```

Problem Solution

- Function convert binary to decimal

```
def convert_binary_to_decimal(x):
    res = 0
    for i in range(len(x)):
        d = ord(x[len(x) - i - 1]) - ord('0')
        res += d * (1 << i)
    return res

s = input()
print(convert_binary_to_decimal(s))
```

Lecture Agenda

- ✓ Section 1: Function Definition
- ✓ Section 2: Calling a Function
- ✓ Section 3: Return Statement
- ✓ Section 4: Passing by Reference & Value
- ✓ Section 5: Function Arguments
- ✓ Section 6: Anonymous Function
- ✓ Section 7: Inner Functions
- ✓ Section 8: Global & Local Variables

Problems

1- Implement a function which converts decimal to octal

2- Implement a function which converts octal to decimal

3- Implement a function which converts decimal to hexa-decimal

4- Implement a function which converts hexa-decimal to decimal

5- Implement a function which converts octal to hexa-decimal

6- Implement a function which converts hexa-decimal to octal

7- Implement a function which gets all prime numbers between range (two given numbers)

8- Implement a function which checks the given number is even or odd

9- Implement a function which checks the given number is positive or negative or zero

10- Implement a function which calculates the factorial of a number

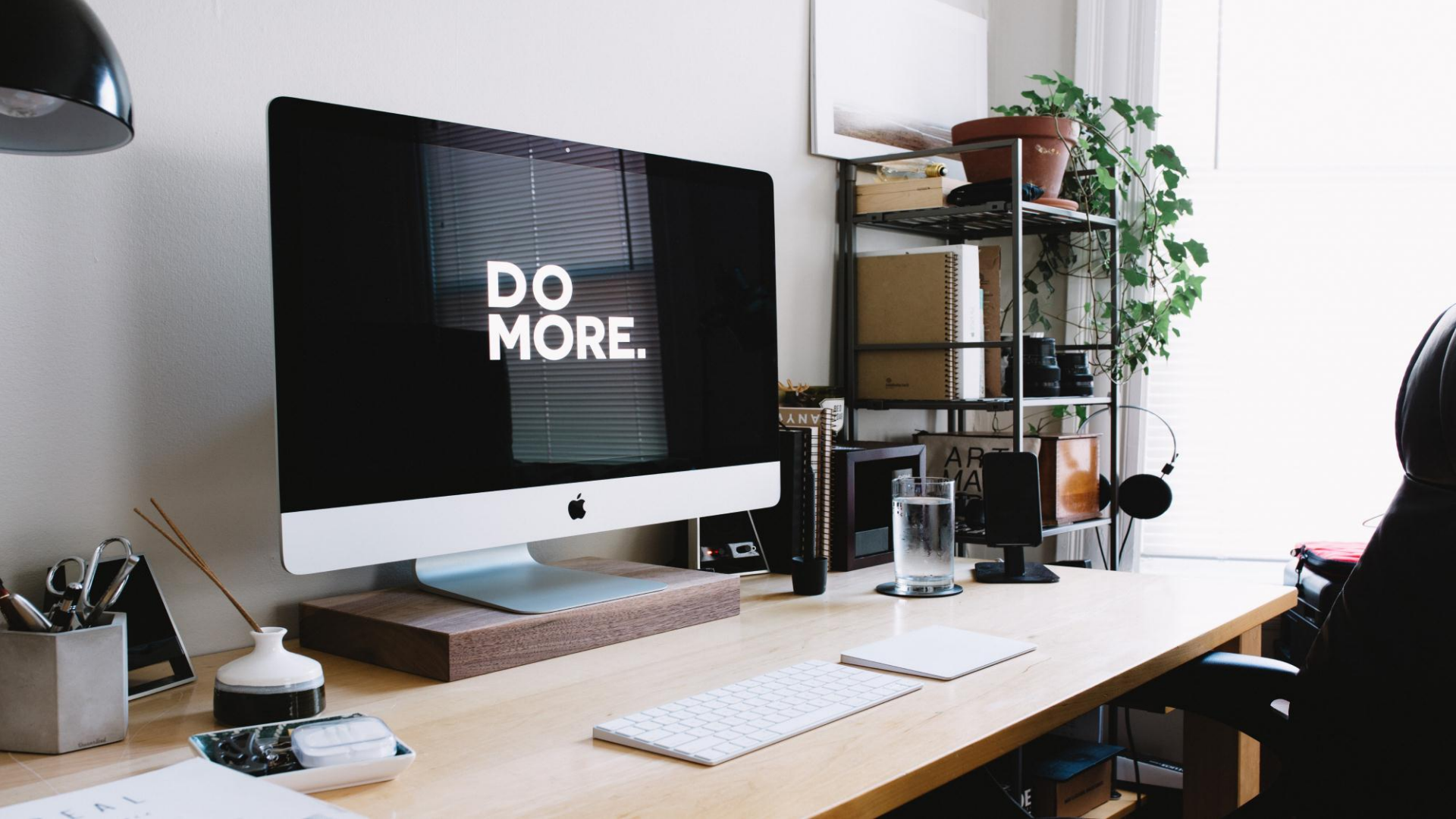
11- Implement a function which gets the factors of a number

12- Implement a function which calculates the absolute value of a number (anonymous function)

13- Implement a function which calculates the square of a number (anonymous function)

14- Implement a function which calculates the square root of a number (anonymous function)

15- Implement a function which prints the alphabet from character ‘a’ to given character.


DO
MORE.