KHWARIZMI SCIENCE SOCIETY

CODING WITH PYTHON

A COURSE FOR ABSOLUTE BEGINNERS

iDRACK

KHWARIZMI
SCIENCE SOCIETY

iDRACK

https://idrack.org/

https://idrack.org/forum/community/

contact@idrack.org

COURSE OUTLINE

KHWARIZMI SCIENCE SOCIETY

5 WEEKS, SATURDAYS, 2-5 PM

Week 1:

- An introduction to Python
- Installing Python on Windows
- Python Shell

Week 2:

- Saving & Running scripts
- Operators & Variables
- Working with Strings

Week 3:

- Python Collections
- Condition Blocks

Week 4:

- Writing Loops
- Functions in Python

Week 5:

- Introducing Modules
- In-class Assessment

iDRACK

INSTRUCTOR CONTACT

ROOP OMAR

(roop.omar@gmail.com)

We will try to cater to all queries within class timings, but if you feel there is something you need help with later, or were not able to ask during the session, please feel free to drop me an email, and I will get back to you as soon as I can.

LOOPS IN PYTHON

KHWARIZMI SCIENCE SOCIETY

iDRACK

WHY DO WE NEED LOOPS ?

KHWARIZMI
SCIENCE SOCIETY

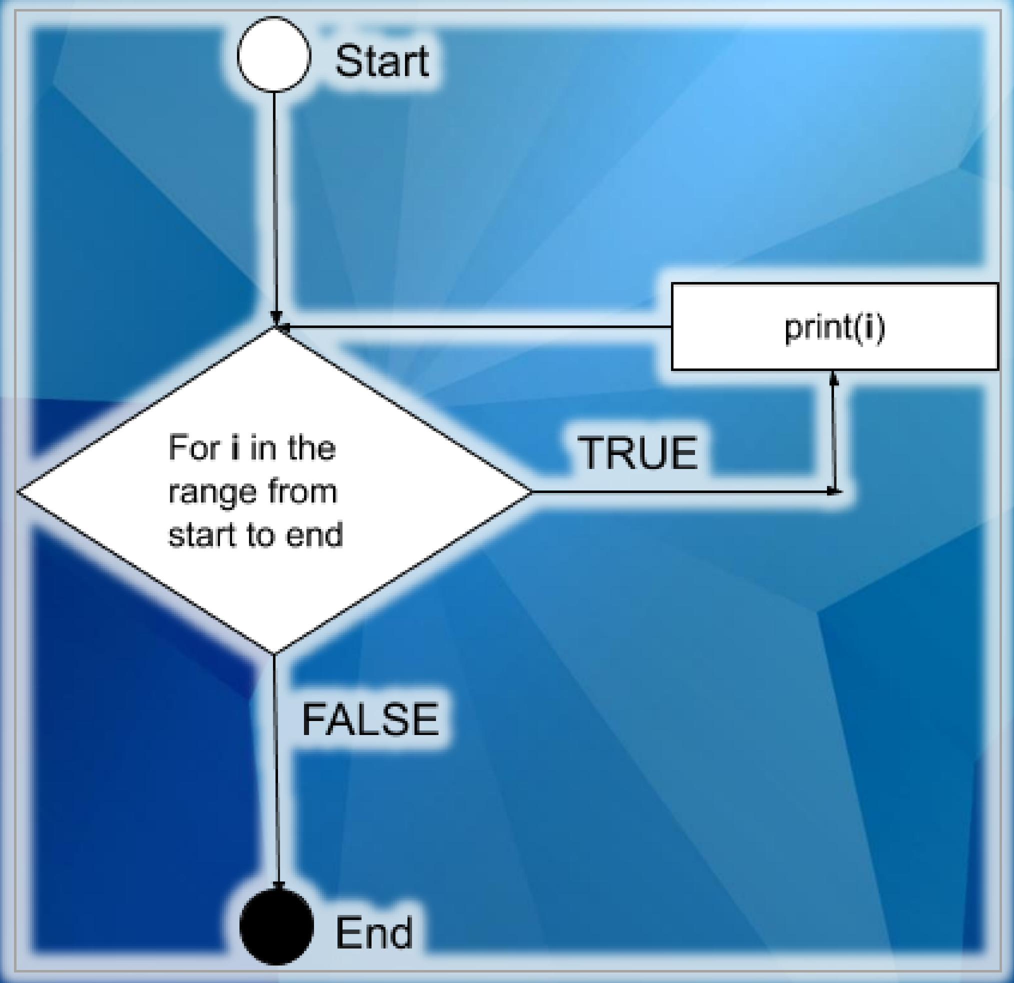
- All programming languages need ways of doing similar things many time.
- Loops are hence an essential feature of any programming or scripting language
- These are also called *iterations*.
- In Python, looping is achieved via the use of *for* and *while*.

#	Loop type	Description
1	for loop	Is an iterator based loop, which steps through the items of iterable objects like lists, tuples, string and executes a piece of code repeatedly for a number of times, based on the number of items in that iterable object.
2	while loop	Executes a block of statements repeatedly as long as the condition is TRUE.

iDRACK

PYTHON FOR LOOP

- The *for* loop in Python is an iterating function.
- If you have a sequence object like a list, you can use the for loop to iterate over the items contained within the list.


Start
For i in the range from start to end
TRUE
print(i)
FALSE
End

PRINT INDIVIDUAL LETTERS OF A STRING USING THE *FOR* LOOP

- Python string is a sequence of characters.
- If you need to go over the characters of a string individually, you can use the *for* loop.

```
word="anaconda"
for letter in word:
    print (letter)
```

USING THE *FOR* LOOP TO ITERATE OVER A PYTHON LIST OR TUPLE

- Lists and tuples are iterable objects.
- You can loop over the elements within these objects using the *for* loop.

```
words= ["Apple", "Banana", "Car", "Dolphin" ]
for word in words:
    print (word)
```

PYTHON *FOR* LOOP WITH *RANGE()* FUNCTION

- *range()* is one of the built-in functions.
- When you want the for loop to run for a specific number of times, or you need to specify a range of objects to print out, the range function works really well.
- *range()* function also takes another parameter apart from the start and the stop.
- This is the step parameter.
- It tells the range function how many numbers to skip between each count.

```
for x in range(3):
    print("Printing:", x)


for n in range(1, 10, 3):
    print("Printing with step:", n)
```

KHWARIZMI SCIENCE SOCIETY

iDRACK

PYTHON *FOR* LOOP WITH *RANGE()* FUNCTION

Parameters	Value
start	1
end	20
step	1 (default value)

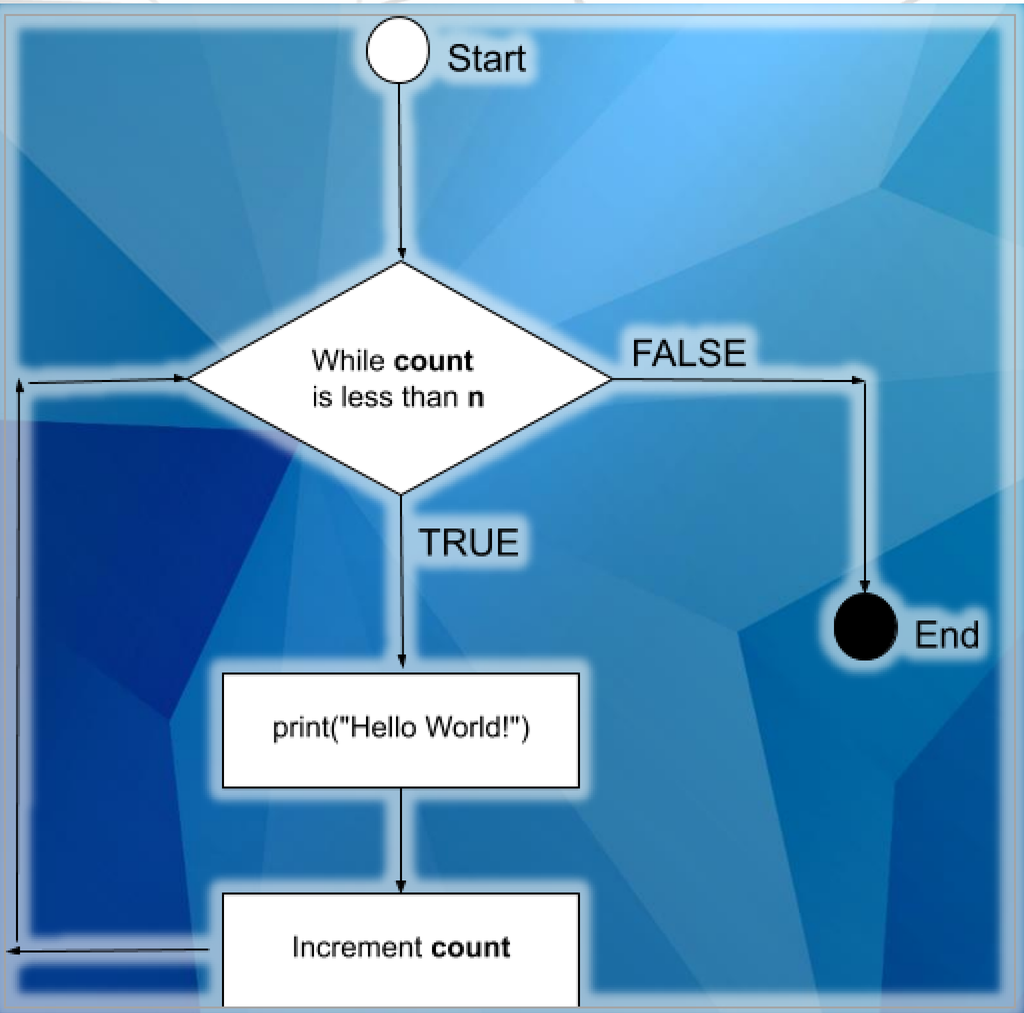
```
for x in range(3):
    print("Printing:", x)


for n in range(1, 10, 3):
    print("Printing with step:", n)
```

PYTHON WHILE LOOP

KHWARIZMI SCIENCE SOCIETY

- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
- We generally use this loop when we don't know the number of times to iterate beforehand.
- In the while loop, condition is checked first. The body of the loop is entered only if the condition evaluates to *True*.
- After one iteration, the condition is checked again. This process continues until the condition evaluates to *False*.
- Also, it is important to note that if the condition is failed in the very first attempt then the loop will not be started.

iDRACK

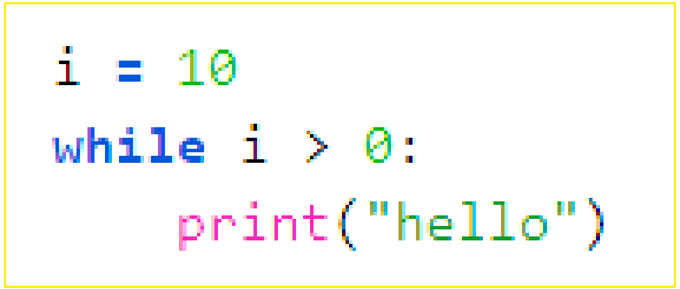

Start
While **count** is less than **n**
FALSE
TRUE
End
print("Hello World!")
Increment **count**

PYTHON WHILE LOOP

- The while loop consists of the following:
  - ✓ The ***while*** keyword
  - ✓ An expression that evaluates to True or False, also known as the ***condition***
  - ✓ A colon **:** character
  - ✓ The ***body***, or clause, of the while loop which is indented

AN INFINITE LOOP

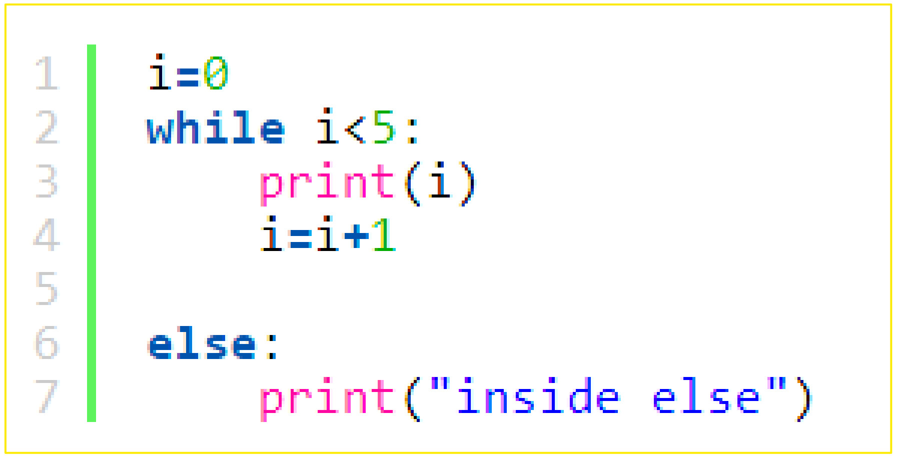
- Be careful while using a ***while*** loop.
- If you forget to increment the counter variable in python, or write flawed logic, the condition may never become false.
- In such a case, the loop will run infinitely.
- To stop execution, press **Ctrl+C** or **Ctrl+Z**.



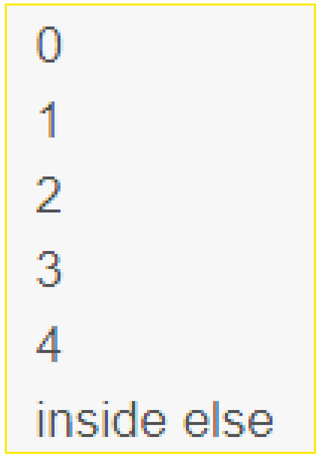
```
i = 10
while i > 0:
    print("hello")
```

THE ELSE STATEMENT FOR WHILE LOOP

- A *while* loop may have an *else* statement after it.
- When the condition becomes false, the block under the else statement is executed.
- However, it doesn't execute if you break out of the loop.



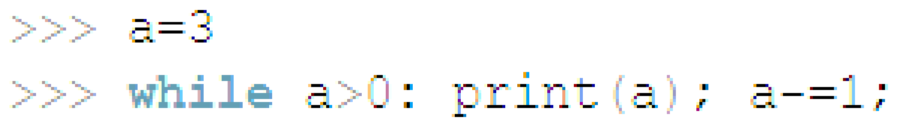
```python
i=0
while i<5:
    print(i)
    i=i+1

else:
    print("inside else")
```



```
0
1
2
3
4
inside else
```

SINGLE STATEMENT WHILE LOOP

- Like an *if* statement, if we have only one statement in ***while's*** body, we can write it all in one line.



```
>>> a=3
>>> while a>0: print(a); a-=1;
```

- You can see that there were two statements in ***while's*** body, but we used semi-colons to separate them.
- Without the second statement, it would form an infinite loop

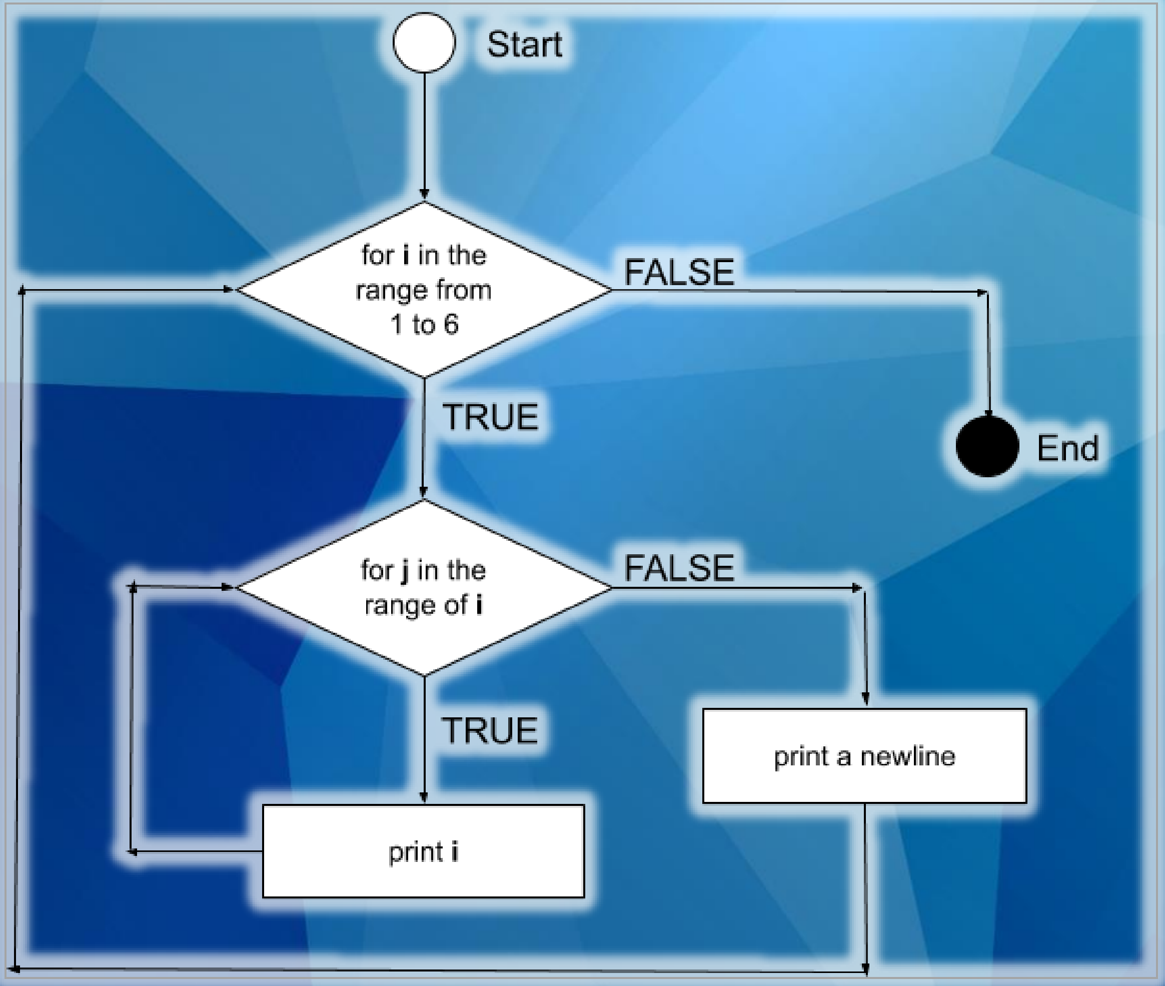
KHWARIZMI SCIENCE SOCIETY

NESTED LOOPS IN PYTHON

iDRACK

NESTED LOOPS IN PYTHON

- You can also nest a loop inside another.
- You can put a *for* loop inside a *while*,
- or a *while* inside a *for*,
- or a *for* inside a *for*,
- or a *while* inside a *while*.
- Or you can put a loop inside a loop inside a loop. You can go as far as you want.


Start
for i in the range from 1 to 6
FALSE
End
TRUE
for j in the range of i
FALSE
TRUE
print a newline
print i

```
words= ["Apple", "Banana", "Car", "Dolphin" ]
for word in words:
        #This loop is fetching word from the list
        print ("The following lines will print each letters of "+word)
        for letter in word:
                #This loop is fetching letter for the word
                print (letter)
        print("") #This print is used to print a blank line
```

```
The following lines will print each letters of Apple
A
p
p
l
e

The following lines will print each letters of Banana
B
a
n
a
n
a

The following lines will print each letters of Car
C
a
r

The following lines will print each letters of Dolphin
D
o
l
p
h
i
n

>>>
```

KHWARIZMI SCIENCE SOCIETY

LOOP CONTROL STATEMENTS IN PYTHON

iDRACK

- Sometimes, you may want to break out of normal execution in a loop.
- For this, we have three keywords in Python:

  ✓ break

  ✓ continue

  ✓ pass

BREAK STATEMENT

- The break statement is used to exit the *for* loop prematurely.
- It is used to break when a specific condition is met.
- Let's say we have a list of numbers and we want to check if a number is present or not.
- We can iterate over the list of numbers and if the number is found, break out of the loop
- This is because we don't need to keep iterating over the remaining elements.

```python
nums = [1, 2, 3, 4, 5, 6]

n = 2

found = False
for num in nums:
    if n == num:
        found = True
        break

print(f'List contains {n}: {found}')
```

CONTINUE STATEMENT

- We can use the continue statements inside a *for* loop to skip the execution of the *for* loop body for a specific condition.
- Let's say we have a list of numbers and we want to print the sum of positive numbers.
- We can use the continue statements to skip the for loop for negative numbers.
- Hence when continue is encountered, it skips the remaining loop for that iteration only.

```python
nums = [1, 2, -3, 4, -5, 6]

sum_positives = 0

for num in nums:
    if num < 0:
        continue
    sum_positives += num

print(f'Sum of Positive Numbers: {sum_positives}')
```

PASS STATEMENT

KHWARIZMI SCIENCE SOCIETY

- The *pass* is a null statement
- The Python interpreter returns a *no-operation (NOP)* after reading the pass statement.
- Nothing happens when the *pass* statement is executed.
- The difference between a *comment* and a *pass* statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored.
- We generally use it as a placeholder.

iDRACK

PASS STATEMENT

- Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would give an error. So, we use the pass statement to construct a body that does nothing.

```
'''pass is just a placeholder for
functionality to be added later.'''
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

KHWARIZMI SCIENCE SOCIETY

LOOPING BACKWARDS IN PYTHON

iDRACK

WHILE LOOP BACKWARD

```
countdown = ['Blastoff!', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10']
start = 10
while start >= 0:
    print(countdown[start])
    start -= 1
```

```
10
9
8
7
6
5
4
3
2
1
Blastoff!
```

FOR LOOP BACKWARD

```python
countdown = ['Blastoff!', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10']
for i in range(10, -1, -1):
    print(countdown[i])
```

```
10
9
8
7
6
5
4
3
2
1
Blastoff!
```

FUNCTIONS IN PYTHON

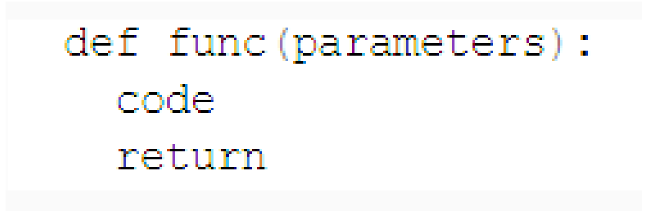
KHWARIZMI SCIENCE SOCIETY

iDRACK

WHAT IS A FUNCTION ?

- Python is object-oriented but also supports functional programming.
- A *function* is a block of code that has a name and you can call it.
- Instead of writing something 100 times, you can create a function and then call it 100 times.
- You can call it anywhere and anytime in your program.
- This adds reusability and modularity to your code.
- Functions can take arguments and return values.
- Functions can be of two types:
  - ✓ *built-in*
  - ✓ *user-defined*

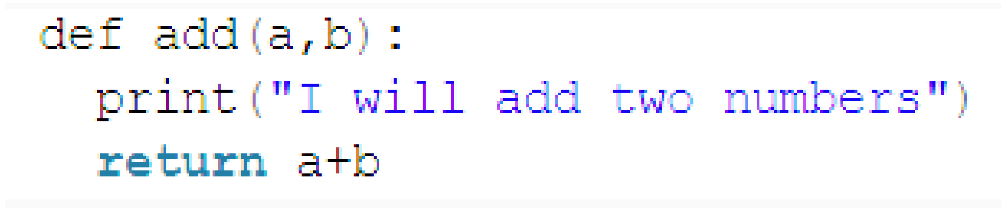
CREATING FUNCTIONS IN PYTHON



```
def func(parameters):
    code
    return
```

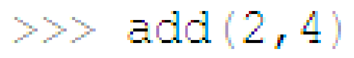
- To define a function, you use the ***def*** keyword.
- You give the function a ***name*** and it can take some ***parameters*** if you want.
- Then you specify a ***block of code*** that will execute when you call the function.
- There are no curly braces in Python and so you need to ***indent*** this code block otherwise it won't work.
- You can make the function ***return a value***.

CREATING FUNCTIONS IN PYTHON



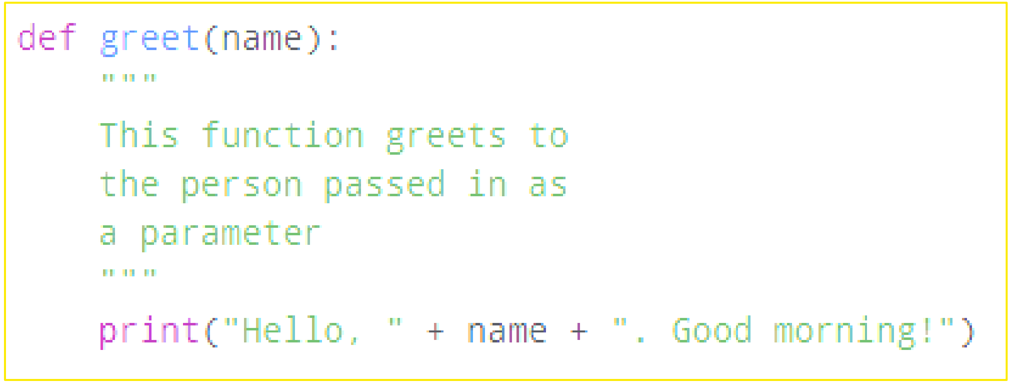
```
def add(a,b):
  print("I will add two numbers")
  return a+b
```

- This function prints “I will add two numbers” and returns the sum of two numbers.



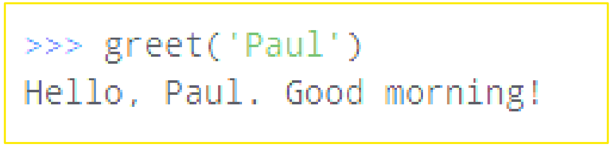
```
>>> add(2,4)
```

CREATING FUNCTIONS IN PYTHON



```python
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")
```

- This function simply displays greetings according to the given name.



```
>>> greet('Paul')
Hello, Paul. Good morning!
```

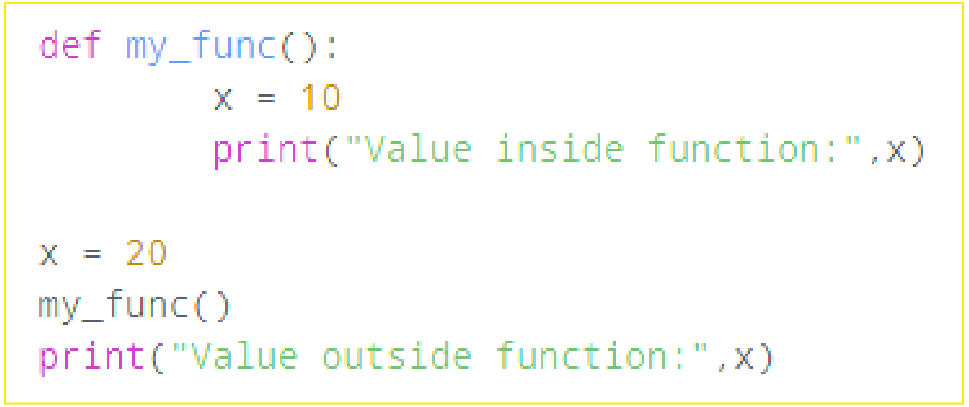
KHWARIZMI SCIENCE SOCIETY

SCOPE AND LIFETIME OF VARIABLES

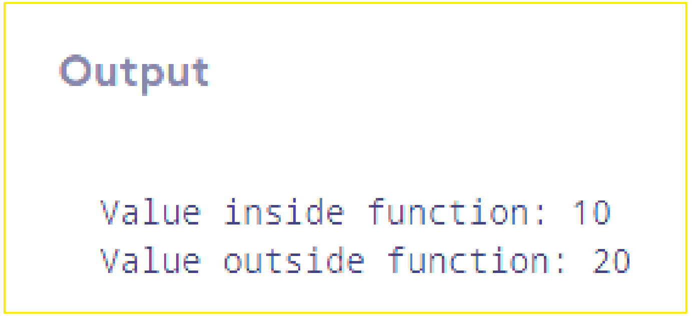
iDRACK

LOCAL & GLOBAL

- Scope of a variable is the portion of a program where the variable is recognized.
- Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a *local* scope.
- The lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.



```python
def my_func():
        x = 10
        print("Value inside function:",x)

x = 20
my_func()
print("Value outside function:",x)
```



Output

```
Value inside function: 10
Value outside function: 20
```

- Here, we can see that the value of x is 20 initially.
- Even though the function *my_func()* changed the value of x to 10, it did not affect the value outside the function.
- This is because the variable x inside the function is different (local to the function) from the one outside.
- Although they have the same names, they are two different variables with different scopes.

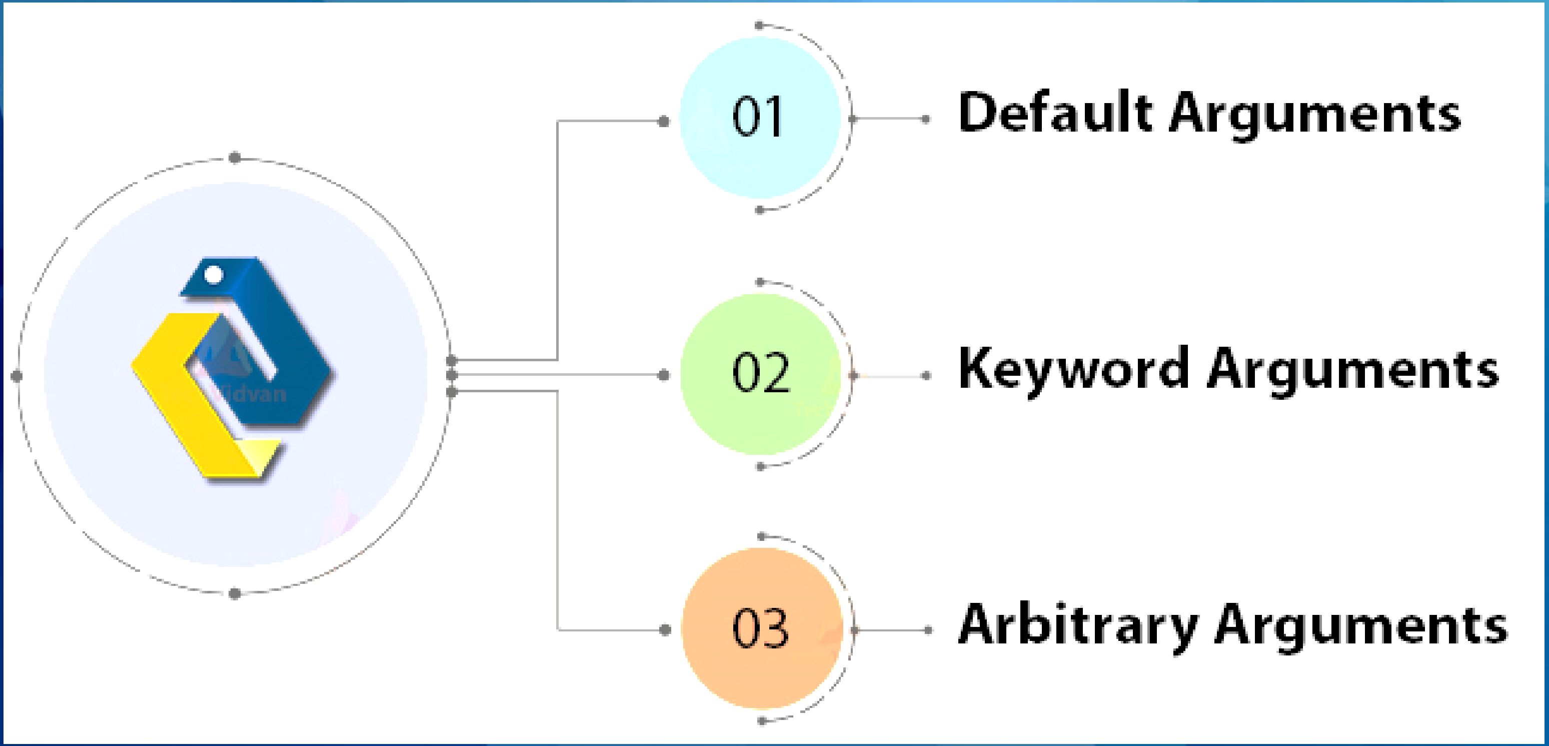
LOCAL & GLOBAL

- On the other hand, variables outside of the function are visible from inside.
- They have a *global* scope.
- We can read these values from inside the function but cannot change (write) them.
- In order to modify the value of variables outside the function, they must be declared as global variables using the keyword *global*.

KHWARIZMI SCIENCE SOCIETY

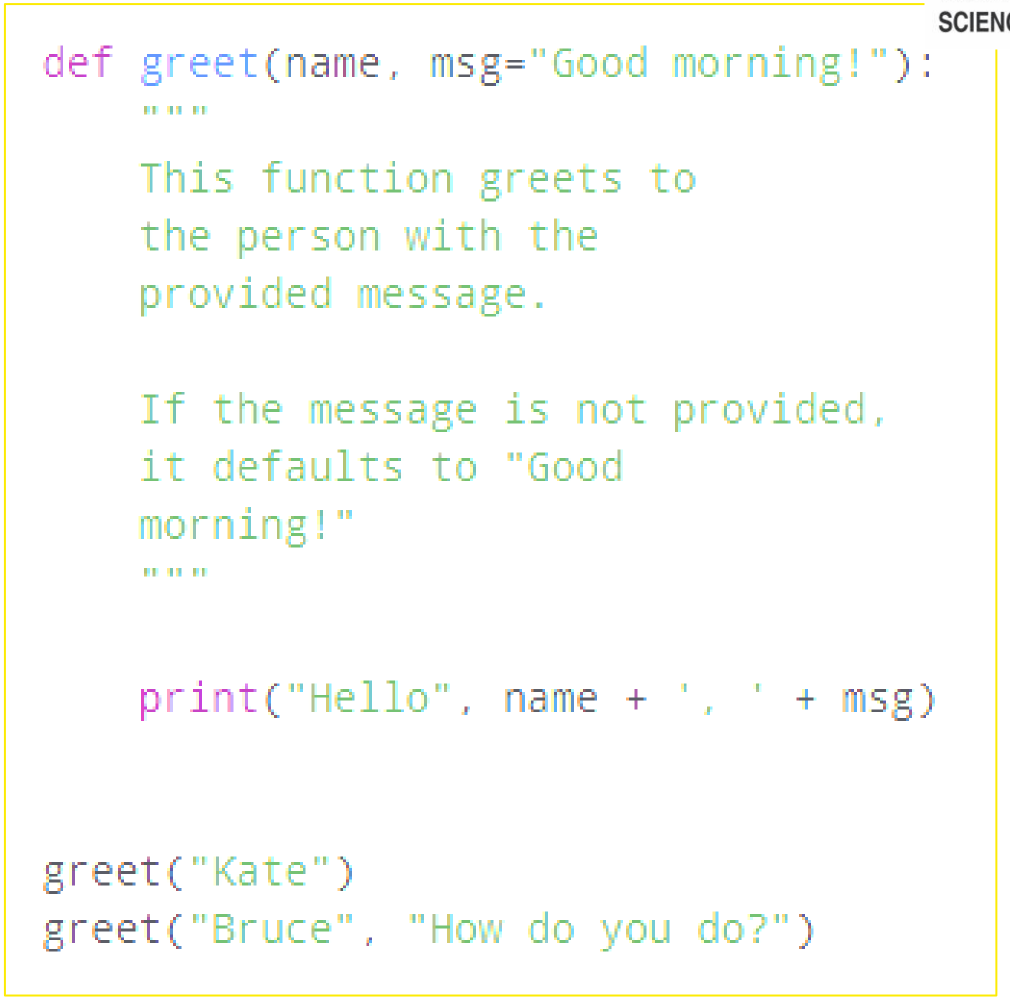
PYTHON FUNCTION PARAMETERS

iDRACK


01
Default Arguments
02
Keyword Arguments
03
Arbitrary Arguments

DEFAULT ARGUMENTS

- You can specify a default value for arguments.
- If the user calls the function, they can skip providing a value for that argument.
- Any number of arguments in a function can have a default value.



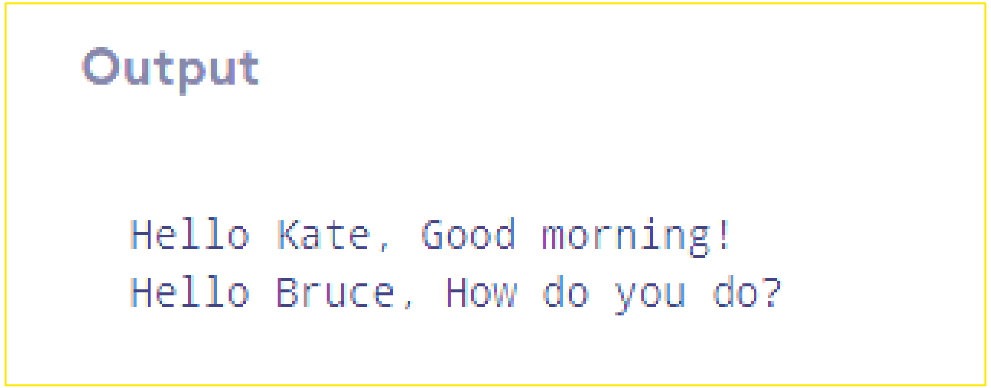
```python
def greet(name, msg="Good morning!"):
    """
    This function greets to
    the person with the
    provided message.

    If the message is not provided,
    it defaults to "Good
    morning!"
    """


    print("Hello", name + ', ' + msg)


greet("Kate")
greet("Bruce", "How do you do?")
```

KHWARIZMI SCIENCE SOCIETY

iDRACK

DEFAULT ARGUMENTS

- In this function, the parameter ***name*** does not have a default value and is required (mandatory) during a call.
- On the other hand, the parameter ***msg*** has a default value of "Good morning!".
- So, it is optional during a call. If a value is provided, it will overwrite the default value.



Output

```
Hello Kate, Good morning!
Hello Bruce, How do you do?
```

DEFAULT ARGUMENTS

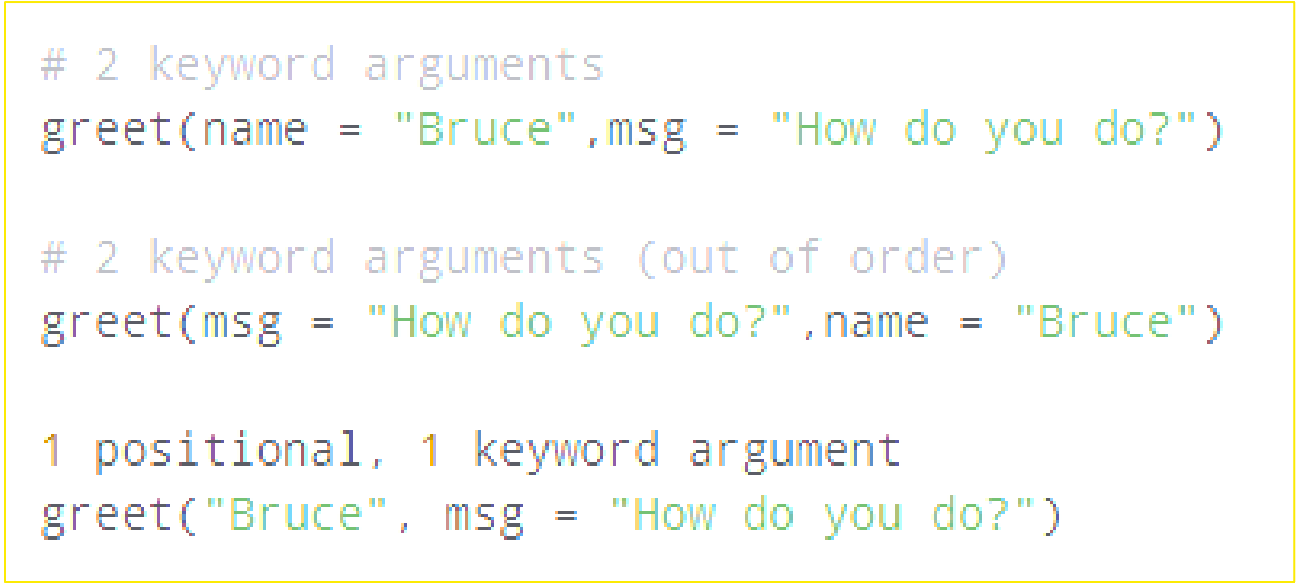
- However, once we have a default argument, all the arguments to its right must also have default values.
- This means to say, *non-default arguments* cannot follow *default arguments*.

```
def greet(msg = "Good morning!", name):
```

```
SyntaxError: non-default argument follows default argument
```

KEYWORD ARGUMENTS

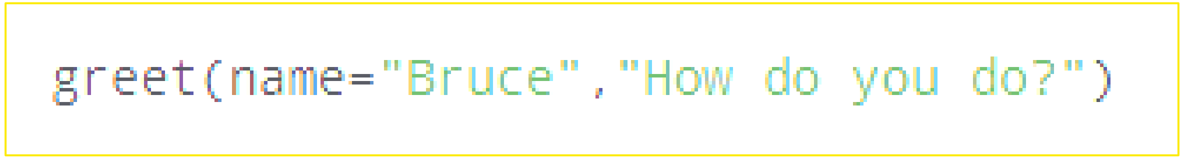
- Python allows functions to be called using keyword arguments.
- When we call functions in this way, the order (position) of the arguments can be changed.
- Following calls to the last function are all valid and produce the same result.



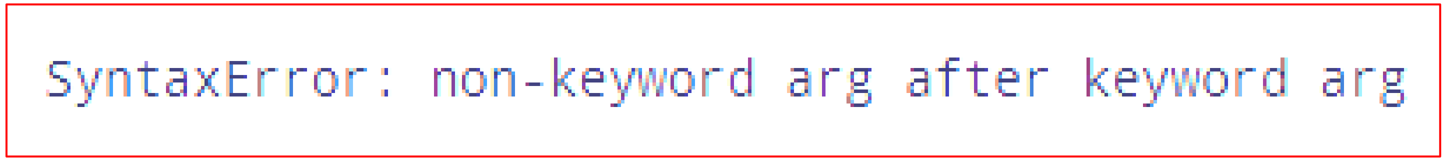
```
# 2 keyword arguments
greet(name = "Bruce",msg = "How do you do?")

# 2 keyword arguments (out of order)
greet(msg = "How do you do?",name = "Bruce")

1 positional, 1 keyword argument
greet("Bruce", msg = "How do you do?")
```

KEYWORD ARGUMENTS

- We can mix *positional* arguments with *keyword* arguments during a function call.
- But we must keep in mind that keyword arguments must follow positional arguments.
- Having a positional argument after keyword arguments will result in errors.



```
greet(name="Bruce","How do you do?")
```

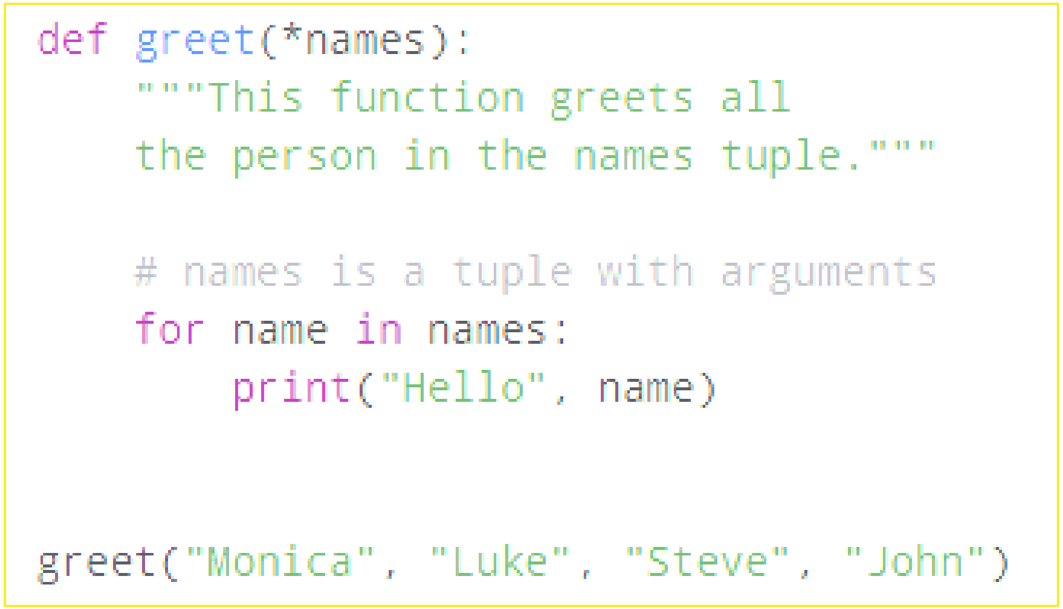


```
SyntaxError: non-keyword arg after keyword arg
```

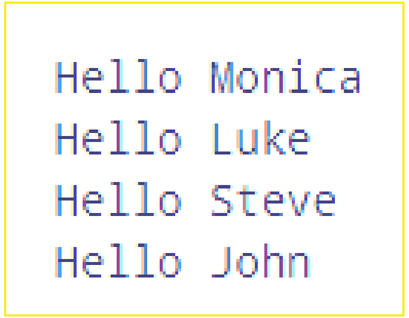
ARBITRARY ARGUMENTS

- 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 an ***arbitrary*** number of arguments.
- In the function definition, we use an asterisk (*) before the parameter name to denote this kind of argument.

ARBITRARY ARGUMENTS



```python
def greet(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
    for name in names:
        print("Hello", name)


greet("Monica", "Luke", "Steve", "John")
```



```
Hello Monica
Hello Luke
Hello Steve
Hello John
```

- Here we called the function with multiple arguments.
- These arguments get wrapped up into a tuple before being passed into the function.
- Inside the function, we use a *for* loop to retrieve all the arguments back.

KHWARIZMI SCIENCE SOCIETY

THE RETURN STATEMENT IN PYTHON

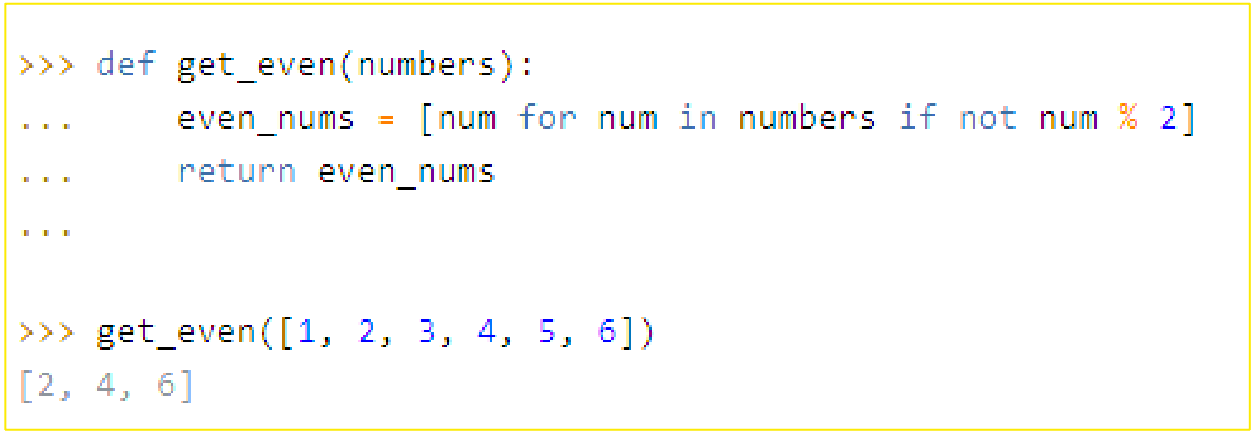
iDRACK

- In general, a function takes ***arguments*** (if any), performs some ***operations***, and ***returns*** a value.
- The value that a function returns to the caller is generally known as the function's ***return value***.
- A return statement consists of the ***return*** keyword followed by an optional return value.
- If the return statement is without any expression, then the special value ***None*** is returned.
- The statements after the return statements are not executed.
- Note: Return statements can not be used outside the function.
- You can use any Python object as a return value.
- Since everything in Python is an object, you can return strings, lists, tuples, dictionaries, functions, classes, instances, anything !

KHWARIZMI
SCIENCE SOCIETY

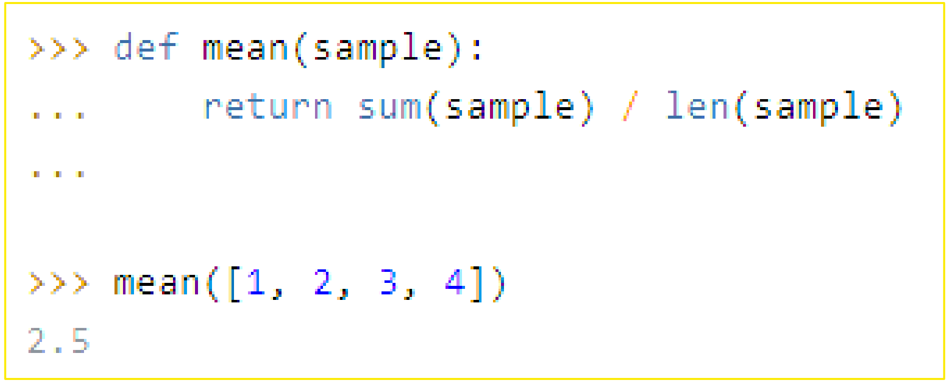
iDRACK

- *get_even()* uses a list that filters out the odd numbers in the original numbers.
- Then the function returns the resulting list, which contains only even numbers.



```
>>> def get_even(numbers):
...     even_nums = [num for num in numbers if not num % 2]
...     return even_nums
...

>>> get_even([1, 2, 3, 4, 5, 6])
[2, 4, 6]
```

- Say you need to calculate the mean of a sample of numeric values.
- To do that, you need to divide the sum of the values by the number of values.



```
>>> def mean(sample):
...     return sum(sample) / len(sample)
...

>>> mean([1, 2, 3, 4])
2.5
```

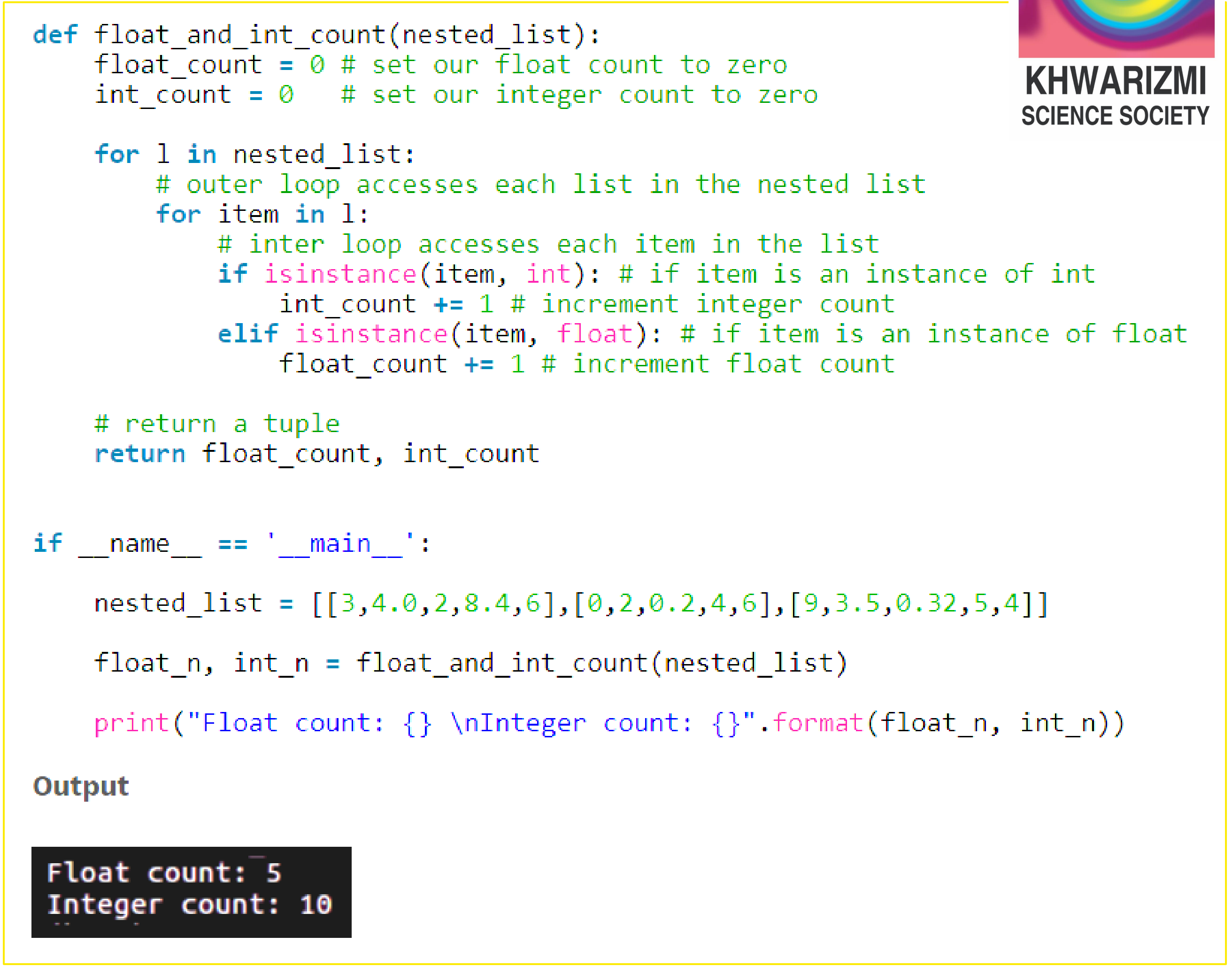
KHWARIZMI SCIENCE SOCIETY

LET'S LOOK AT ANOTHER EXAMPLE

iDRACK

```
>>> [[3,4.0,2,8.4,6],[0,2,0.2,4,6],[9,3.5,0.32,5,4]]
```

- The *outer loop* accesses the first inner list (3,4.0,2,8.4,6) in the nested list.
- The *inner loop* accesses each item in this first inner list, and checks if it is a float or integer.
- If it is an integer, it increments the integer count (*int_count*). Else if it is a float, it increments the float count (*float_count*).
- Once it has finished iterating through this first inner list, it then moves back to the outer loop and accesses the second list (0,2,0.2,4,6)
- The same process continues until it has accessed all the inner lists.



```python
def float_and_int_count(nested_list):
    float_count = 0 # set our float count to zero
    int_count = 0   # set our integer count to zero

    for l in nested_list:
        # outer loop accesses each list in the nested list
        for item in l:
            # inter loop accesses each item in the list
            if isinstance(item, int): # if item is an instance of int
                int_count += 1 # increment integer count
            elif isinstance(item, float): # if item is an instance of float
                float_count += 1 # increment float count

    # return a tuple
    return float_count, int_count


if __name__ == '__main__':

    nested_list = [[3,4.0,2,8.4,6],[0,2,0.2,4,6],[9,3.5,0.32,5,4]]

    float_n, int_n = float_and_int_count(nested_list)

    print("Float count: {} \nInteger count: {}".format(float_n, int_n))
```

Output

```
Float count: 5
Integer count: 10
```

KHWARIZMI SCIENCE SOCIETY

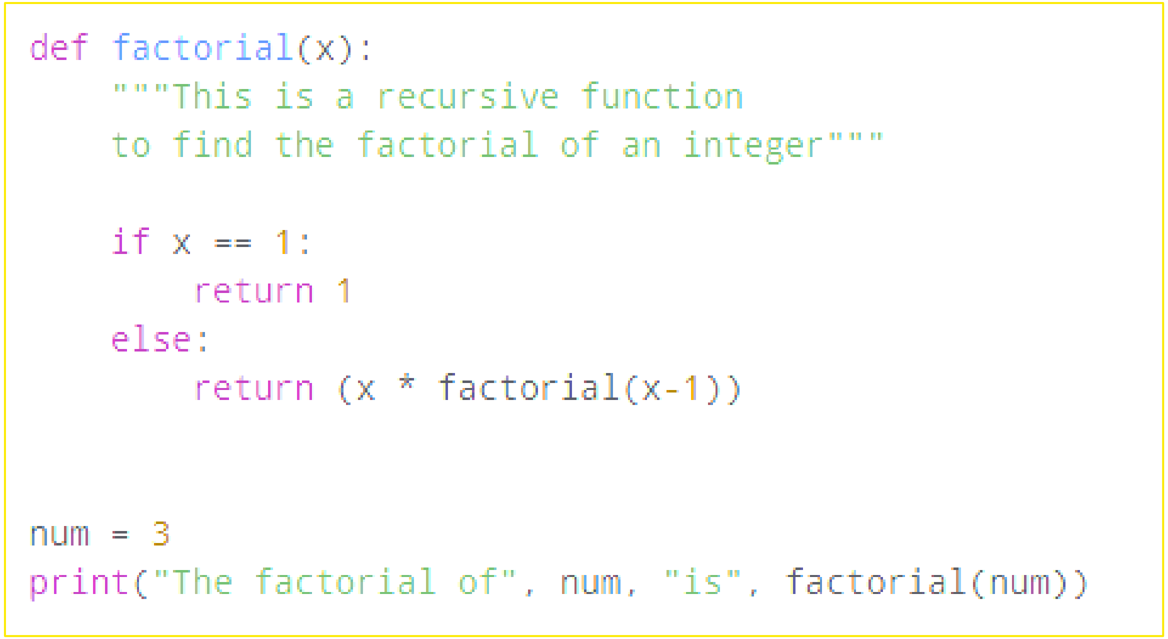
RECURSION IN PYTHON

iDRACK

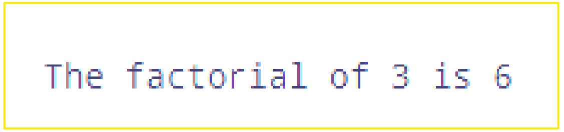
RECURSIVE FUNCTIONS

- When a function calls itself, it is ***recursion***.
- In other words, when a function body has calls to the function itself, it is recursion.
- For recursion, you should specify a ***base condition*** otherwise the function can execute infinitely.
- The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.
- By default, the maximum depth of recursion is 1000.

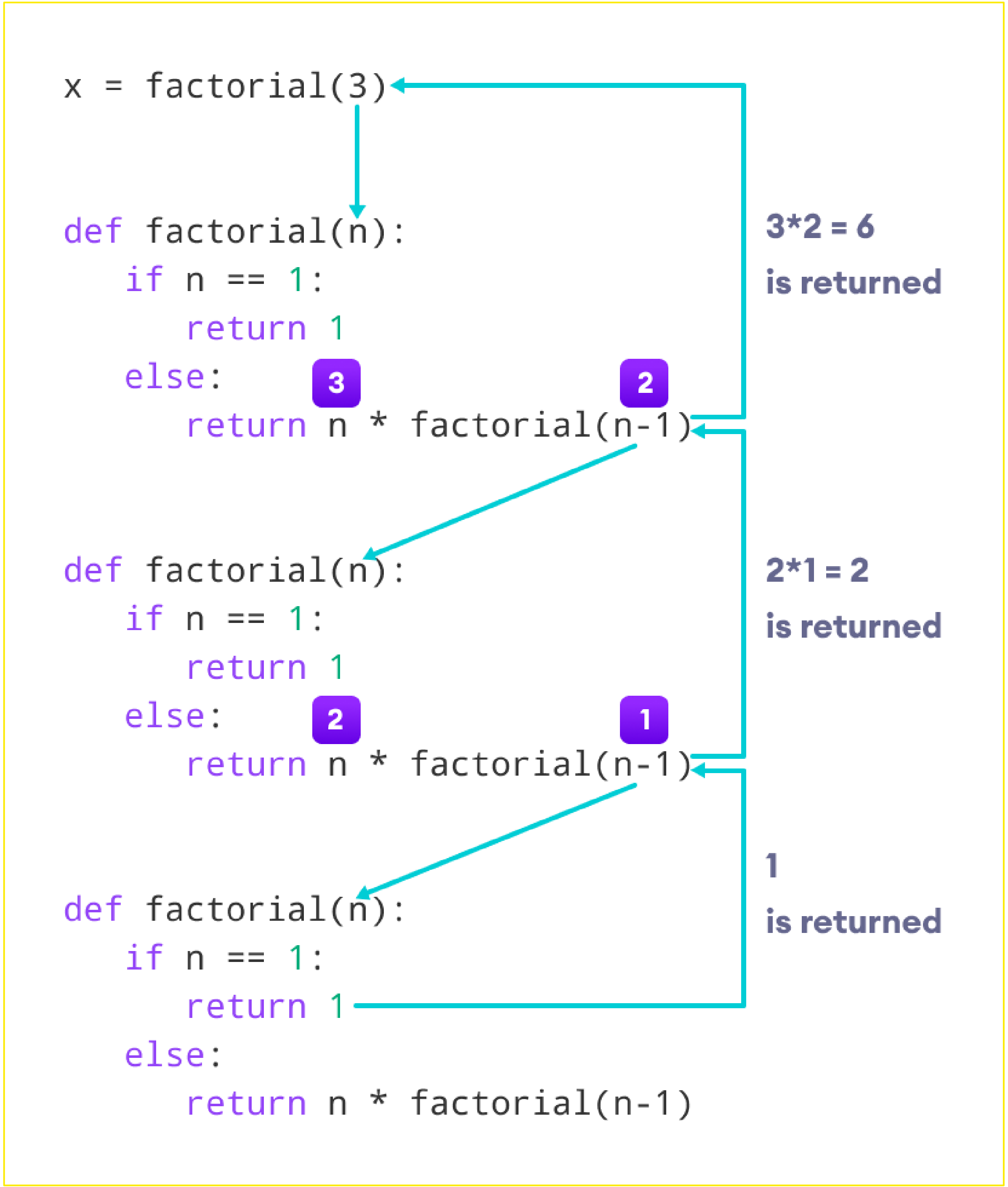
- Following is an example of a recursive function to find the factorial of an integer.
- Factorial of a number is the product of all the integers from 1 to that number.
- For example, the factorial of 6 (denoted as 6!) is **1*2*3*4*5*6 = 720.**



```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```



```
The factorial of 3 is 6
```


```
x = factorial(3)

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- Our recursion ends when the number reduces to 1.
- This is called the *base condition*.

ADVANTAGES OF RECURSION

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

DISADVANTAGES OF RECURSION

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.