INST0002 PROGRAMMING 1

LECTURE 4- FUNCTIONS

Dr Daniel Onah
Lecturer
Module Leader

Copyright Note:

Recap

Re: Cap☺

- We discussed about conditional statements – IF, ELIF, ELSE
- We looked at for loop iteration
- We looked at while loop
- We discussed about nested statements
- We discussed about operators' precedence
- We discussed about Python indentation and good code structure

Outline/Learning outcomes

1. To be able to implement functions
2. Use of predefined and user-defined function
3. To be familiar with the concept of function parameter passing
4. Local scope and global scope
5. Type of functions (lambda and recursive functions)
6. To develop the skills of solving complex tasks into simpler ones
7. To understand the scope of a function variable
8. To understand recursive nature of function call

Functions

Introduction to Function

A function is made up of block of statements (codes) that perform specific task. There are some conventional rules to adhered to when defining a function. These are:

- All function blocks start with the **def** keyword.
- After the definition of the function, the function name and parentheses () are used.
- Then an argument or parameter should be pass inside the parentheses or a default parameter list (i.e., leaving the parentheses empty).
- All function code block begins with a colon (:) and this helps in structuring the indentation of the body of the function.

Functions

A function is a sequence of instructions with a name. For example, the round() function is a function that round a floating-point value to a specified number of decimal places indicated as a passing argument in the function call.

- *A function is a named sequence of instructions*
- *Arguments are supplied when a function is called*

A function accepts **data** as a parameter or an argument and returns the outcome. The return values are the result that the function computes. i.e., the **output** for which the function computes is called the return value.

Function? Features!

- Function is like a variable, but just that it hold reusable codes.
- Function program is created to avoid code repetition.
- Function can be recursive in nature
- When a function is called, the compiler remembers the line for which the function is called
- Function is defined by using the def keyword
- Functions are a useful way of dividing program codes into manageable chunks or pieces to allow proper organization, reusability of codes and time management.
- Function is a block of code(s) that executes when it is called.

Python Built-in Functions

There are two kinds of functions in Python

- **Built-in functions** : - this are functions that are provided as part of Python programming such as: - **print() , input(), type(), float(), int()** … etc
- **User-defined functions**: - this are Functions that we define ourselves for our programs

We treat the built-in function names as "new" **reserved words** (i.e., we avoid using them as variable names)

Function Definition

In Python a **function** is some reusable code that takes argument(s) as input, does some computation, and then returns a result or results.

- We define a function using the def reserved word
- We call/invoke the function by using the function name, parentheses, and arguments in an expression.

Why Use Functions

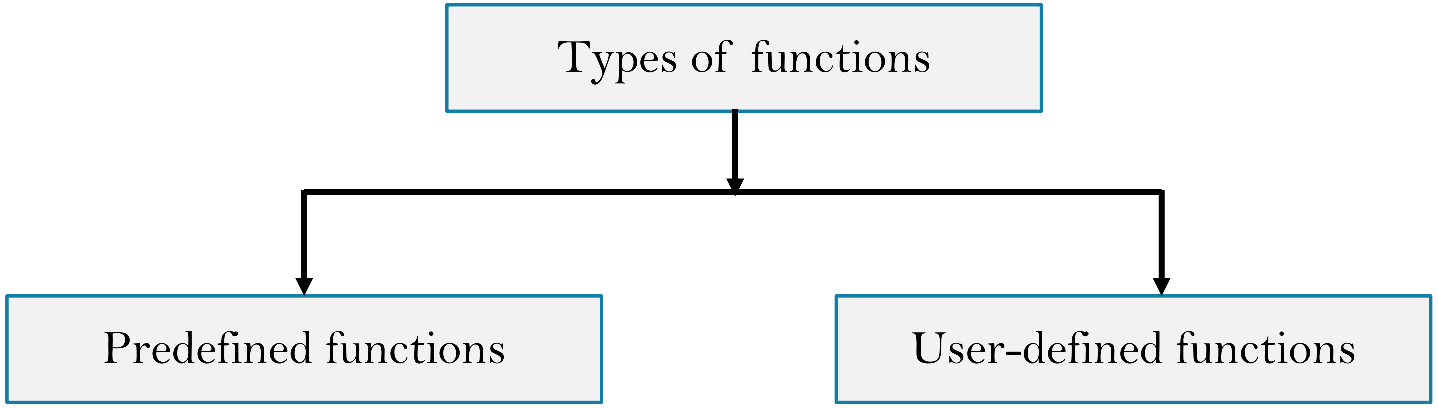
Once we have **defined** a function, we can call (or invoke) it as many times as we like. This is known as the **store** and **reuse** pattern programming code constructs.

Syntax of a Function

```
def function_name(parameters):
    "Statement 1"
    "Statement 2"
    "Statement 3"
     .
     .
     .
    "Statement n"
    return [expression]
```

Types of Functions

There are two basic types of functions: (1) the predefined functions and (2) the user-defined functions.

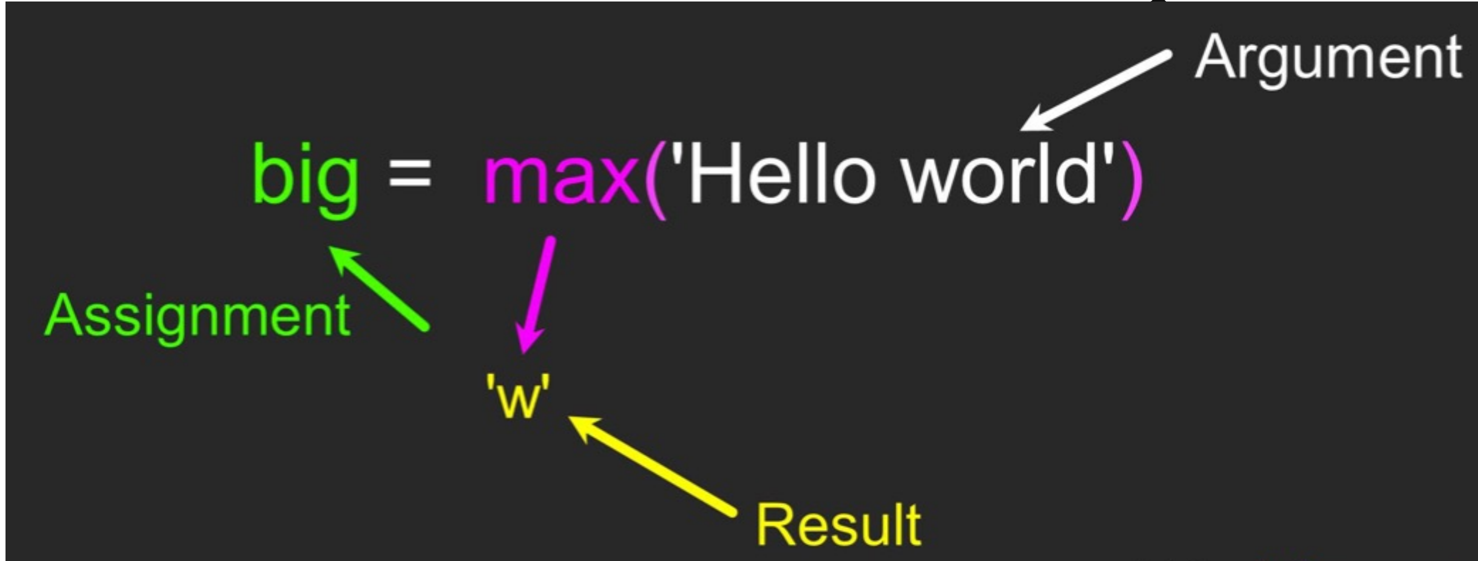


Predefined Functions

The predefined function is a Python built-in function. The functionality of a predefined function is regulated or established or predefined by Python interpreter. These built-in functions are mostly represented using lower case letters. Examples of predefined or Python built-in functions are:

Built-in function	Functionality	Built-in function	Functionality
abs()	It returns the absolute value of a number	max()	It returns the largest item in an iterable
bin()	It returns the binary version of a number	min()	It returns the smallest item in an iterable
float()	It returns a floating-point number	oct()	It converts a number into an octal (octal decimal number base 8)
hex()	It converts a number into a hexadecimal value	pow()	It returns the value of x to the power of y
int()	It returns an integer number	print()	It prints to the standard output devices
len()	It returns the length of an object	range()	It returns a sequence of numbers, starting from 0 and increments by 1 (by default)
list()	It returns a list	round()	It rounds a give number to the nearest decimal

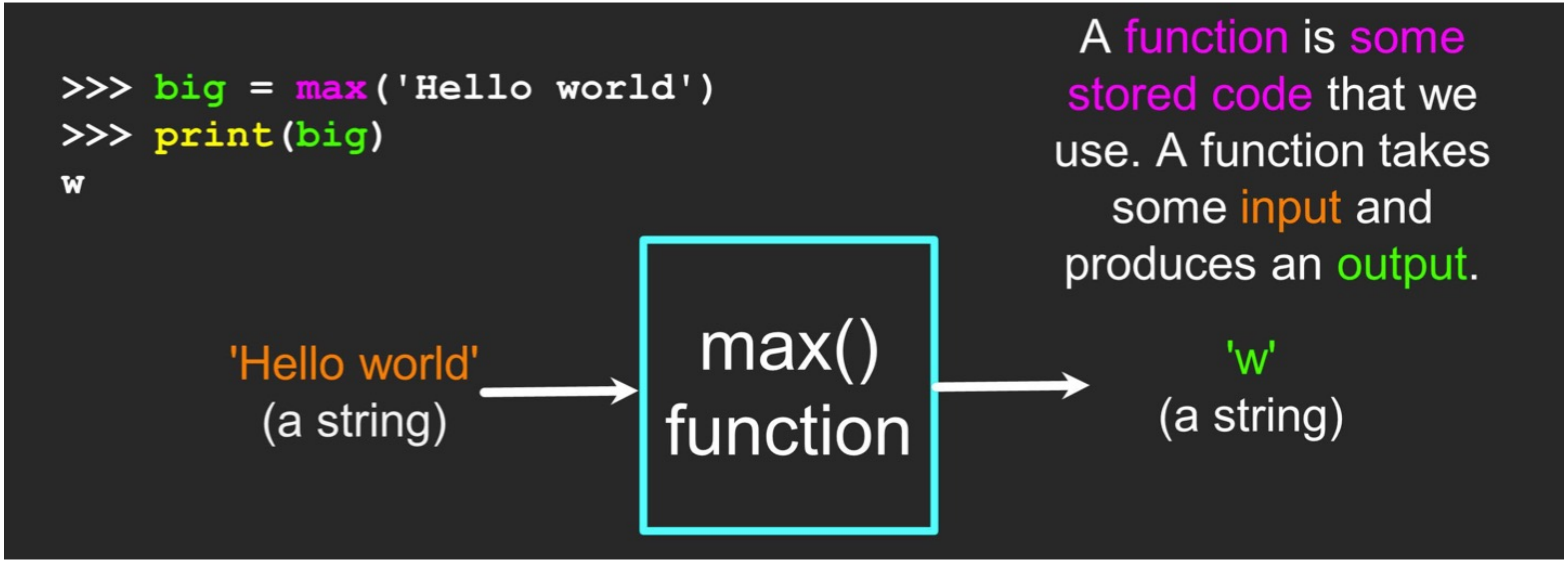
Max Function Example



```
>>> big = max('Hello world')
>>> print(big)
w
>>> tiny = min('Hello world')
>>> print(tiny)

>>>
```

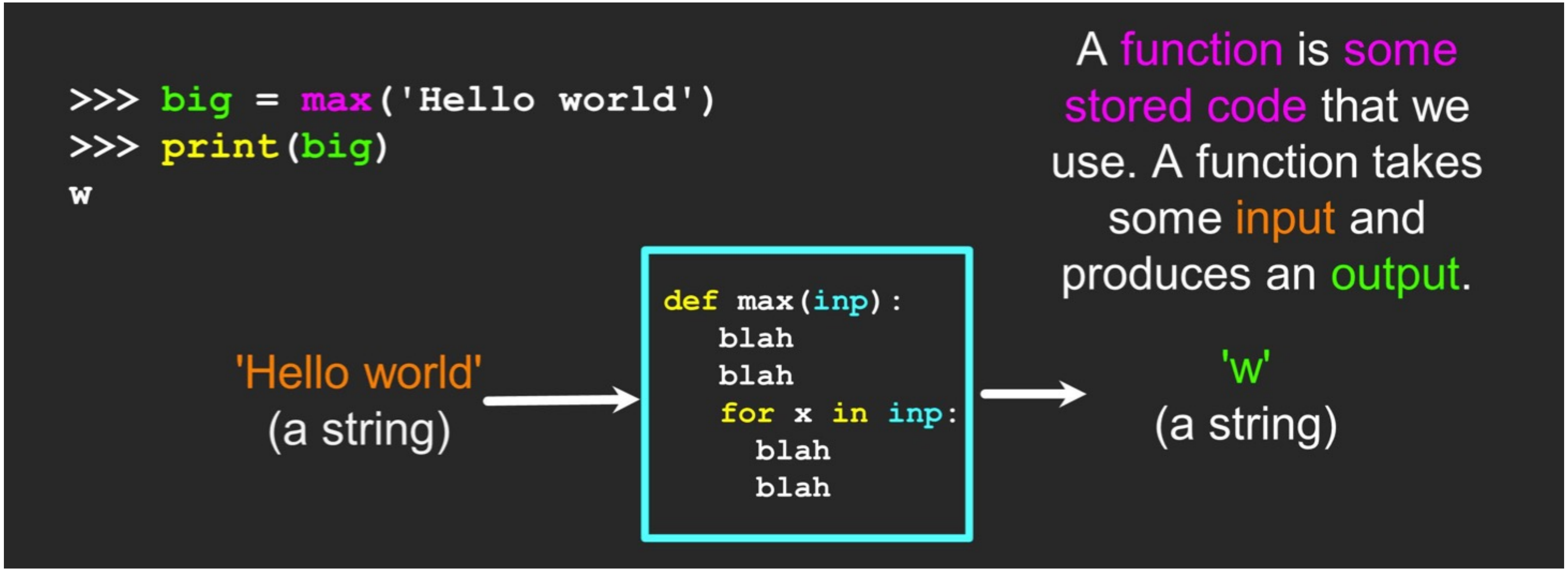
Max Function



```
>>> big = max('Hello world')
>>> print(big)
w
```

A function is some stored code that we use. A function takes some input and produces an output.

Max Function



```
>>> big = max('Hello world')
>>> print(big)
w
```

A function is some stored code that we use. A function takes some input and produces an output.

```
def max(inp):
    blah
    blah
    for x in inp:
      blah
      blah
```

User-defined Functions

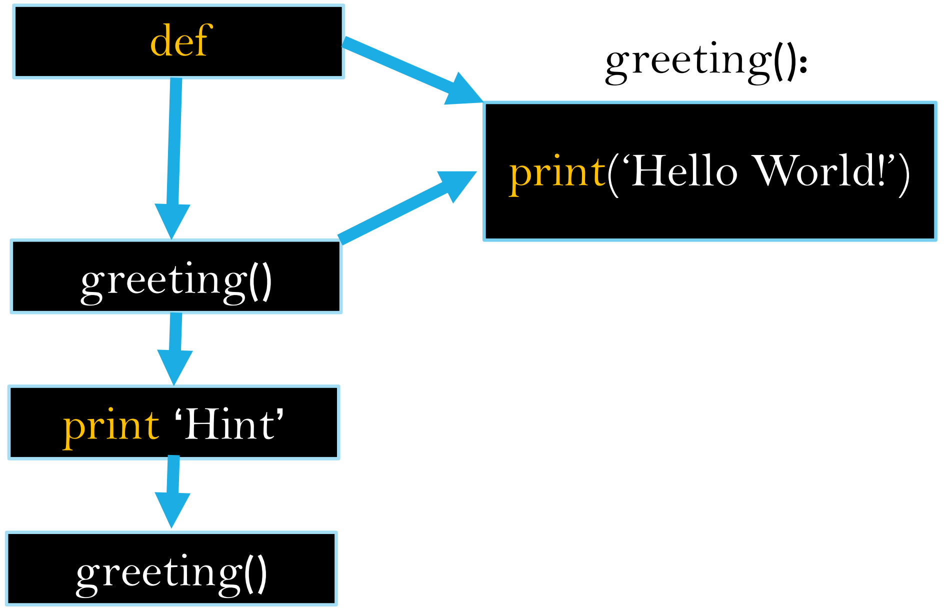
When we write a function to perform specific task, those functions are called user-defined functions. User-defined function can also use the Python predefined or built-in functions to support the program performance or operations. User-defined function just as predefined functions could also be **stored** and **reused** when needed in a program. Let's look at an example of user-defined function to help us understand the working procedure.

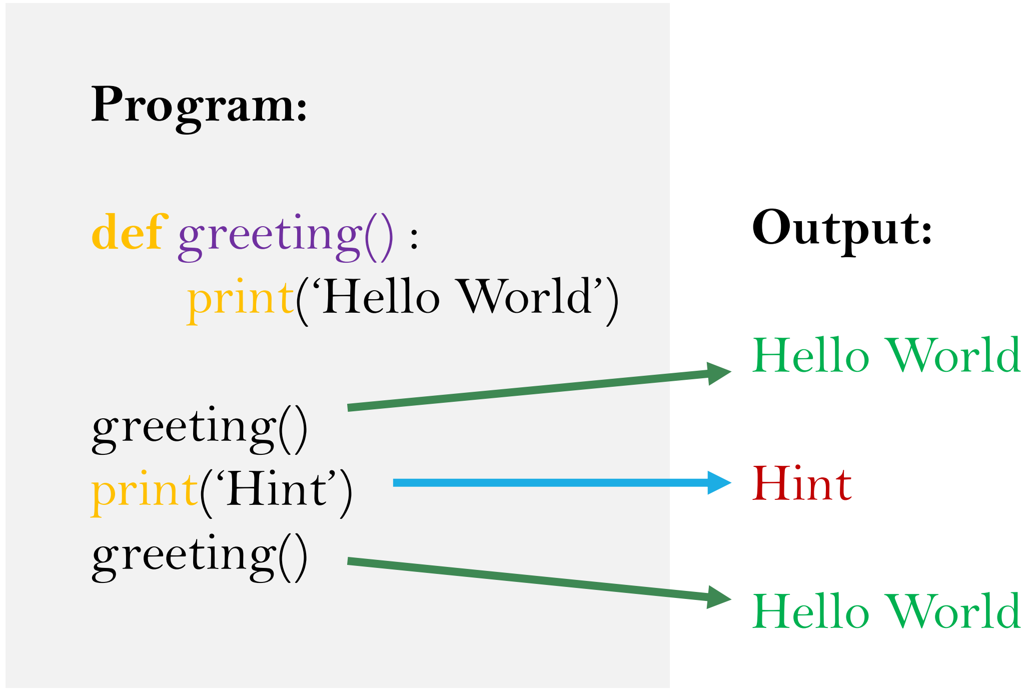
```
# Declaring a greeting function
def greeting():
    print("Inside function body")
# calling the function
greeting()
```

Output:

```
Inside function body
```

Stored and Reused




Program:

```
def greeting() :
      print('Hello World')

greeting()
print('Hint')
greeting()
```

Output:

```
Hello World
Hint
Hello World
```

Example

Let's look at a simple Python function for checking whether a given number is even or odd.

```python
"""
A program to check whether a given number(s) is even or odd.
"""
def evenOddNumber(number):
  if(number % 2 == 0): # This check if there is no remainder, this means the number will be even
    print("The number {} is an even".format(number))
# otherwise if the condition is false or not met, means the number is odd
  else:
    print("The number {} is odd".format(number))

# Now we should call the function
evenOddNumber(13)
evenOddNumber(56)
```

Output:

```
The number 13 is odd
The number 56 is an even
```

Example

Let's look at a simple Python function for checking whether a given number is even or odd. Checking whether any of the input argument has a zero value.

```
"""
A program to check whether a given number(s) is even or odd.
Check for zero input

"""
def evenOddNumber(number):
    if(number == 0):
        print("The number {} is not accepted. ".format(number))
    elif(number % 2 == 0): # This check if there is no remainder,
this means the number will be even
        print("The number {} is an even".format(number))
# otherwise if the condition is false or not met, means the number is odd
    else:
        print("The number {} is odd".format(number))

# Now we should call the function
evenOddNumber(0)
evenOddNumber(56)
```

Output:

```
The number 0 is not
accepted
The number 56 is an even
```

Function Parameters & Arguments

Parameters and Arguments

A piece of data or information can be passed into a function as arguments and parameters. Arguments are specific values with data type (e.g. int, float, string etc.) which we passed into the function to perform specific operation. We can pass *several arguments* into the function separate with a **comma**.

Note that the term parameter and argument can be used interchangeably when describing the input of a function call. But there is a slight distinction between the function parameters and arguments.

Let's clarify this point: The variable defined within the parentheses when we define our function is referred to as a **parameter list**. While the value we passed into the function when it is called is referred to as an **argument**.

Parameters and Arguments

Let's look at an example to help you understand the meaning of function arguments and parameters.

```
 # Two parameters (F_name and L_name)

def name_function(F_name, L_name):
    print(F_name + " " + L_name)

# Two arguments value (Danny, Onah)

name_function("Danny", "Onah") # Function calling
```

Output:

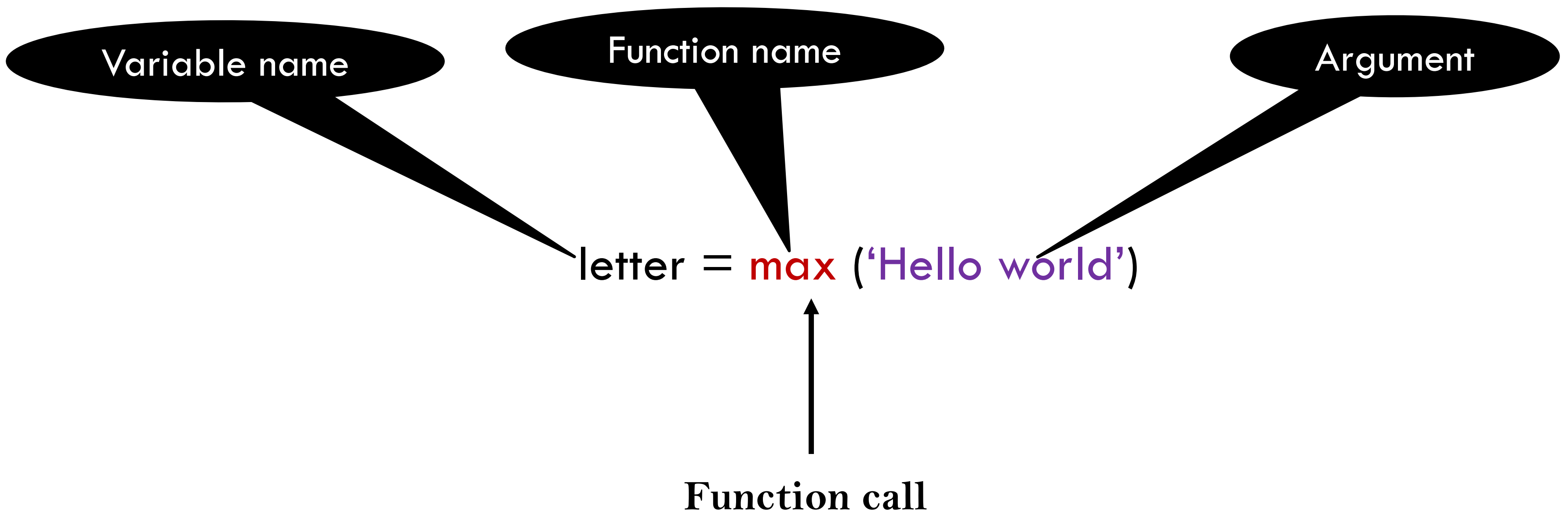
```
Danny Onah
```

Arguments

An argument is a value we pass into the function as its input when we call the function.

We use arguments so we can instruct the function to perform different kinds of operation when we call it at different times.

Arguments are passed as input in the parentheses after the name of the function is called.



```
letter = max ('Hello world')
```

Default Arguments

In Python default values for function arguments are allowed. The assignment operator is used to provide a default value for an argument.

Let’s look at an example:

```
# Function definition
def my_name(name, msg= "hope you are doing well!"):
    print("Hello ", name + ', ' + msg)

# Function calling
my_name("Danny Onah")
my_name("Danny", "good to see you!")
```

Output:

```
Hello  Danny Onah, hope you are doing well!
Hello  Danny, good to see you!
```

Arbitrary Arguments

In Python, arbitrary arguments are use as a place-holder to hold so many values passed in as arguments in a function parameter list. Sometimes you might not know **how many arguments** to use. In this case, we use the asterisk (*) to denote this **arbitrary method** before the *parameter list* in the function.

Let's look at an arbitrary arguments:

```python
# Python Arbitrary Arguments
def my_name(*names):
    print("Hello", names)

my_name("Danny", "Onah", "Dan")
```

Output:

```
Hello ('Danny', 'Onah', 'Dan')
```

Example- Function with For Loop

We can also use loops and conditional statements in a function.

Let’s look at how this is done with an example.

```
# arguments passing
def fruit(food):
    for n in food:
    print(n)
fruits = ["apple", "cherry", "banana"]
fruit(fruits)
```

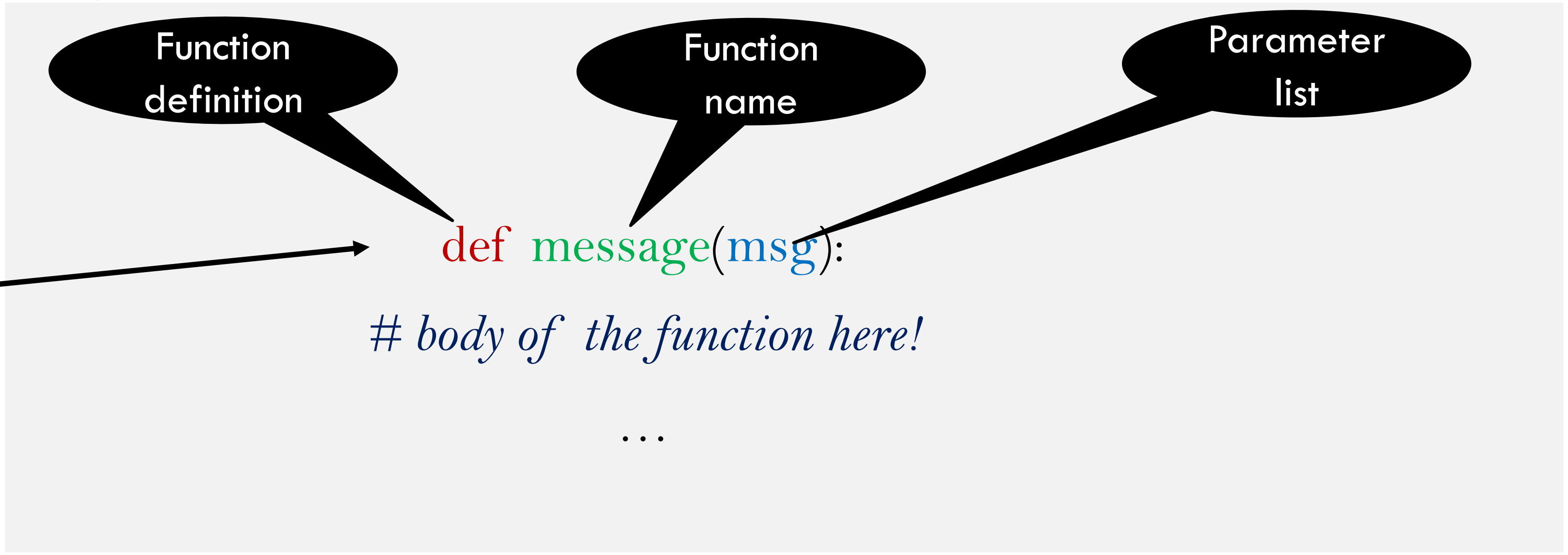
Output:

```
apple
cherry
banana
```

Parameters

A parameter is a variable which we use in the function definition. It is a "**handle**" or "**placeholder**" or "**alias**" that allows the code in the function to access the arguments for a particular function invocation.

Function definition or creation



```
def message(msg):
    # body of the function here!
    ...
```

Function overloading

This a feature of object-oriented programming where two or more functions have the same name but are ***differentiated*** by the *parameter list*.

```
def add():
def add(int x):
def add(float x):
def add(int x, int y):
def add(int x, float y):
def add(float y, int x):
```

$\left.\begin{matrix} \\ \\ \end{matrix}\right\}$ Different order parameters

Implementing a function

Let's see how we could implement a function call by computing the volume of the cube.

- Here you will learn how to create a simple function to compute the volume of a cube with a given length.
- We will then explore how to implement a function from a given specification and call it with test inputs.

Hint & Convention

When writing this function, we need to consider the following factors:

- Define the name of the function e.g. **volumeOfCube**
- Define a variable for each of argument e.g. (**length**)
- These variables are called the input parameter list or variables
- All this information together with the **def** reserved keyword formed the first line of the function, in other words this is the header of the function.

```
def volumeOfCube(length):
```

Function Definition : Header

All this information together with the def reserved keyword formed the first line of the function, in other words this is the **header of the function**.

```
def volumeOfCube(length):
```

Body of a Function

The body of a function contains the statements that are executed when the function is called.

For example; the volume of a cube with side length L:

$$V = L \times L \times L = L^3$$

Note our parameter list is called **length**, so we will use length.

We will then store the value in a variable called **volume**:

```
volume = length ** 3
```

Return Statement

Return statement is very essential in functions. It is used to **finish the execution of a function call** and **return the result of the value in an expression**. There are some essential points to note when it comes to return statements:

- **Return None:** This is same as a return statement with no arguments.
- The statements following the return statements are skipped
- If there is no expression in the return declaration, the unique value None is returned.
- You cannot use a return statement outside of a function declaration. All return statement should be inside the body or block of a function definition.

Return Syntax:

```
def  function_name():
        # function body
        .
        .
        .
        return [expression]
```

Return statement inside the function

Example 1

Let's look at a few examples on how the return statement works.

```
 # A program function to return the sum of two integer numbers

def addition():
    return 15 + 46 # adding two whole numbers and returning the value
print(addition()) # Calling the addition() function to print the result
```

Output: 61

Example 2

Let's look at a few examples on how the return statement works.

```
 # A program function to add two parameters and return the value
def addition(number1, number2):
    total = number1 + number2
    return total
total = addition(59, 82)
print("The sum is =",total) # displaying the result
 # Note that the comma (,) create the space between the value and the
 assignment operator
 # which allows us to display the result properly. If you forgot this, it
 would lead to SyntaxError.
```

Output: The sum is = 141

Example 2 – Interactive Mode

Let's look at a few examples on how the return statement works with interactive mode. Remember once you finish declaring the function block or body using the tab-key, press **enter** to start the assignment section and print statement (*last two interactive lines below*).

```
>>> def addition(number1, number2):
...     total = number1 + number2
...     return total
...
>>> total = addition(59, 82)
>>> print('The sum is = ', total)
The sum is =  141
```

Create the body with tab key

Press enter key to complete the last two lines

Create the assignment and print statement

Returning Function Values

In order for us to return the outcome or result of the function, we use the **return statement**. Often a function will take its arguments, do some computation, and return a value to be used as the value of the ***function call*** in the calling expression.

The return keyword is used for returning the value stored in the variable.

```
return volume
```

The return statement does two things:

- *Stops the function*
- *Return and determine the residual value*

Example: Residual Return Value

```
def  greeting():
        return “Hello”
print(greeting(), “Mike”)
print(greeting(), “Lucy”)
```

Residual value

Sample output:

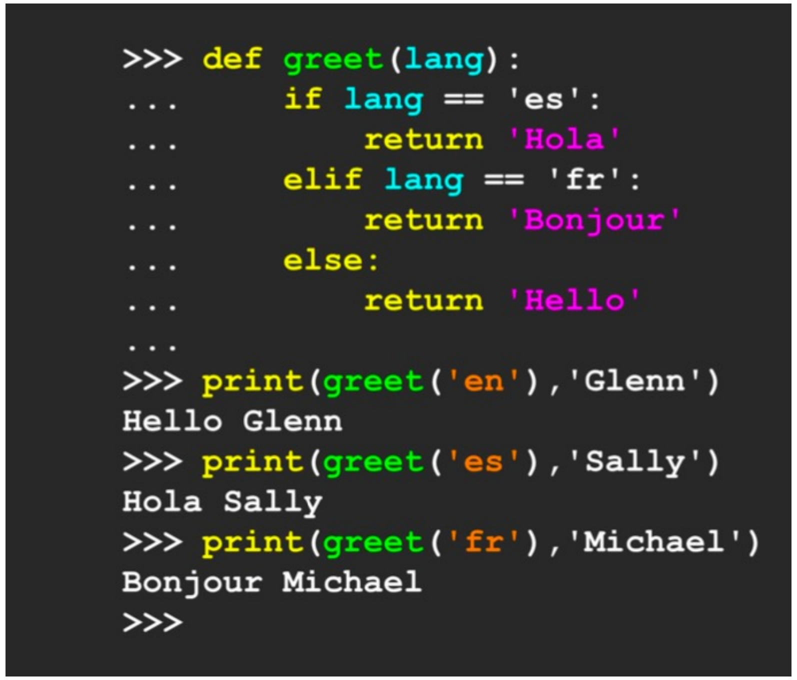
```
Hello Mike
Hello Lucy
```

Return Value

A “**fruitful**” function is one that produces a result (or return value)

The return statement ends the function execution and “***sends back***” the result of the function.

Some functions does not return values, they are said to be “non-fruitful” functions.



```
>>> def greet(lang):
...     if lang == 'es':
...         return 'Hola'
...     elif lang == 'fr':
...         return 'Bonjour'
...     else:
...         return 'Hello'
...
>>> print(greet('en'),'Glenn')
Hello Glenn
>>> print(greet('es'),'Sally')
Hola Sally
>>> print(greet('fr'),'Michael')
Bonjour Michael
>>>
```

Return & Print() Method

In order to print out the output to the screen, we have to use the return statement together with the print() method or function.

For example:

```
return print(volume)
```

Invoking or Calling a Function

Before we could print the operation of the execution of the body, we have to **invoke** the **function call**.

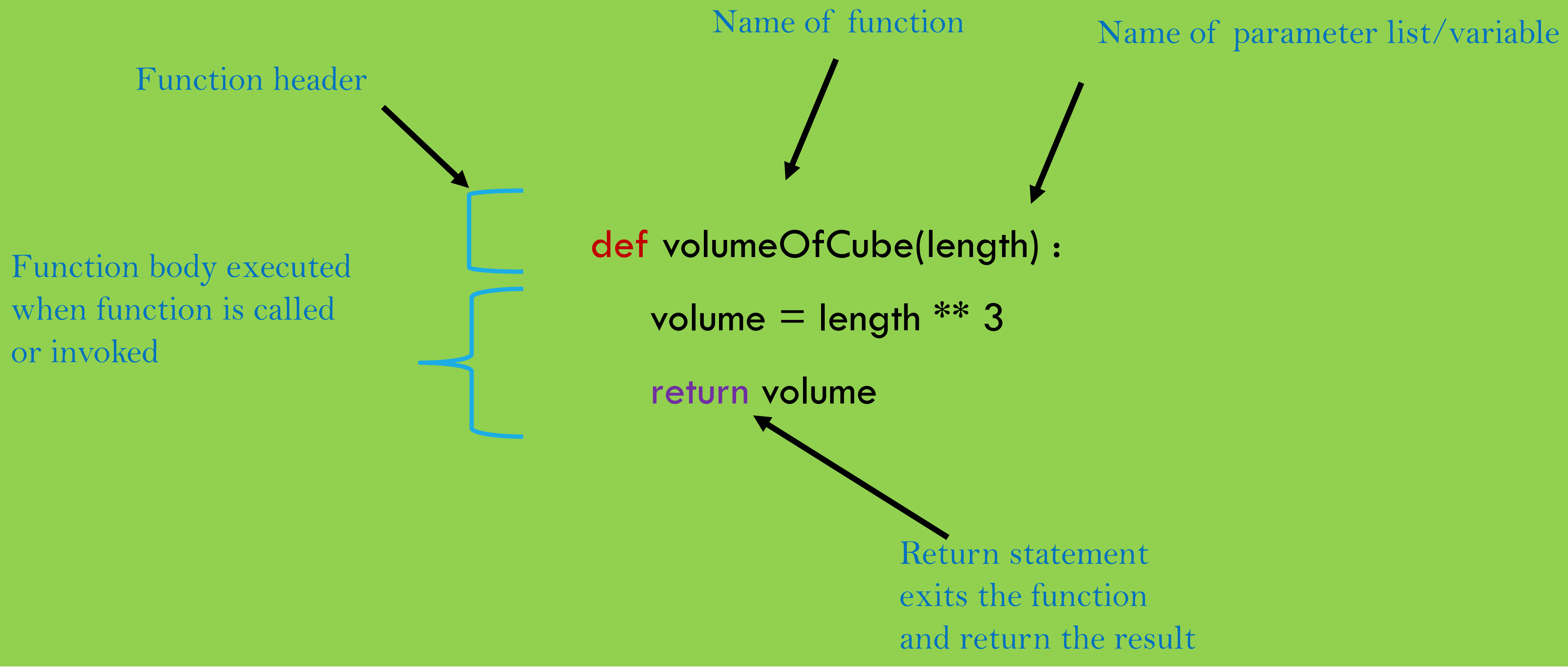
```
volumeOfCube(length)
```

Complete Function

Note: A function is a compound statement, which requires the statement in the body to be indented to the same level.

```
def volumeOfCube(length) :
        volume = length ** 3
        return volume
```

Function Construct/Structure


Name of function
Name of parameter list/variable
Function header
Function body executed when function is called or invoked
Return statement exits the function and return the result

```
def volumeOfCube(length) :
    volume = length ** 3
    return volume
```

Function Call & Execution

Function Call

This is very important in the use and reuse of functions. We call a function to perform a specific task within our program. To call a function, we type the name of the function and the parameters or arguments.

Syntax of calling a function:

```
function_name(argument_1, argument_2)
```

Function Call

This is very important in the use and reuse of functions. We call a function to perform a specific task within our program. To call a function, we type the name of the function and the parameters or arguments.

Syntax of calling a function:

```
# Function definition
def my_name(name):
        print(name)
my_name(‘Danny’) # Calling a function with an argument
```

Output: Danny

Function Call

This is very important in the use and reuse of functions. We call a function to perform a specific task within our program. To call a function, we type the name of the function and the parameters or arguments.

Syntax of calling a function:

```
# Function to sum three integers
def sum_three_numbers(num1, num2, num3):
    return (num1 + num2 + num3)

# Function calling with three arguments
number_add = sum_three_numbers(10,20,30)
print(number_add)
```

Output:

```
60
```

Testing a Function

If we run the program containing only the function definition, then nothing will happen. This is because we have not call or invoke the function.

In order to test the function, your program should contain the following constructs;

- The **definition** of the function
- Statements that **call the function** and **print the result**

Example: Arguments, Parameters, and Results

Here is sample of a full program

```
def volumeOfCube(length) :
        volume = length ** 3
        return volume
result1 = volumeOfCube(3)
result2 = volumeOfCube(16)
print(“A cube with length 3 has volume”, result1)
print(“A cube with length 16 has volume”, result2)
```

ERROR!

A function cannot be called before its declaration.

For example, this will lead to compilation error:

```
print(volumeOfCube(23)) # name compilation error
def  volumeOfCube(length) :
        Volume = length ** 3
        return volume
```

Here the interpreter does not know that the ***volumeOfCube*** *function will be defined later in the program.*

Invoking a Function in Another Function

A function can be called from within another function before the former has been defined or declared.

For example, the following is perfectly legal:

```
def  main():
        result = volumeOfCube(3)
        print("A cube with length 3 has volume", result)


def  volumeOfCube(length) :
        volume = length ** 3
        return volume
main() # calls the main function
```

The statement in the last line (**main()**) does not contained or involved in any of the functions. Therefore, it is executed directly and separately. This statement is used to call the main function (calls the Main function).

Program Illustration

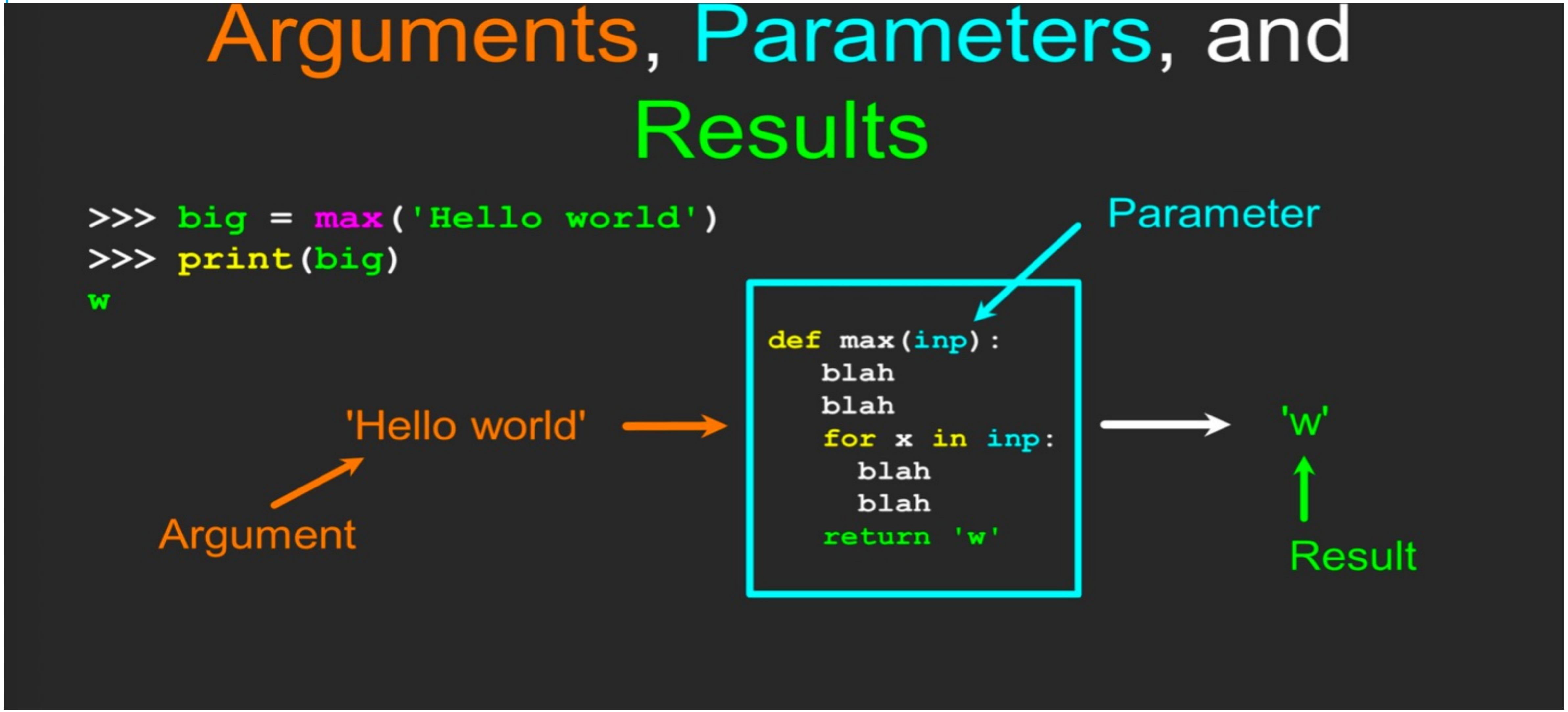
By convention main is the starting point of the program

The volumeOfCube Function is defined below

```
def   main():
          result = volumeOfCube(3)
          print(“A cube with length 3 has volume”, result)


 def volumeOfCube(length) :
        volume = length ** 3
        return volume
 main()
```

This statement is outside any function definition



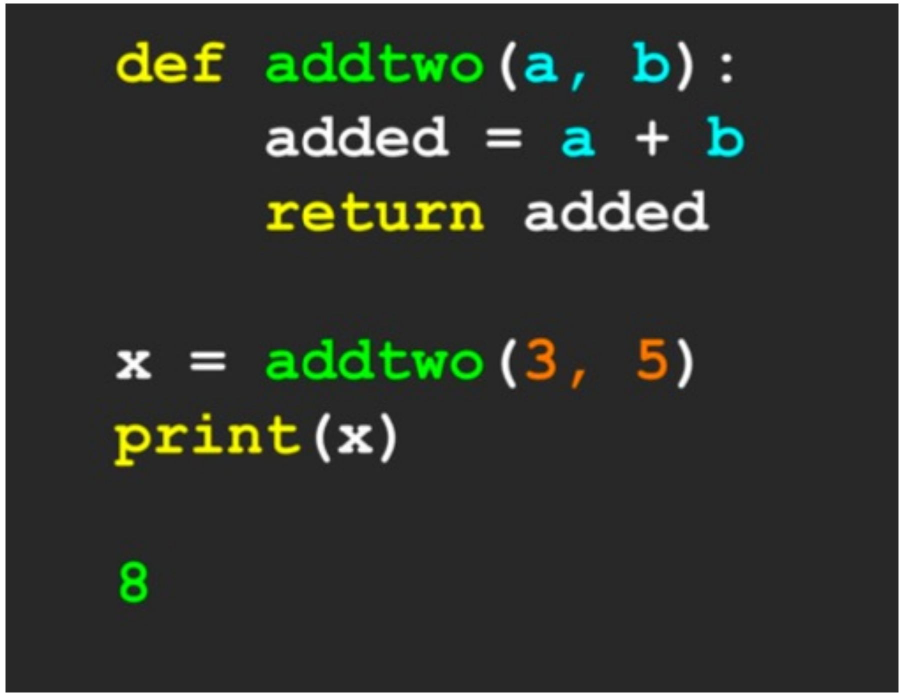
Arguments, Parameters, and Results

```
>>> big = max('Hello world')
>>> print(big)
w
```

```
def max(inp):
    blah
    blah
    for x in inp:
      blah
      blah
    return 'w'
```

Multiple Parameters/Arguments

- We can define more than one parameter in a function definition
- We simply add more arguments when we call the function
- We match the number and order of arguments and parameters



```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)

8
```

Void (Non-fruitful) Functions

When a function does **not return a value**, we call it a **“void”** function.

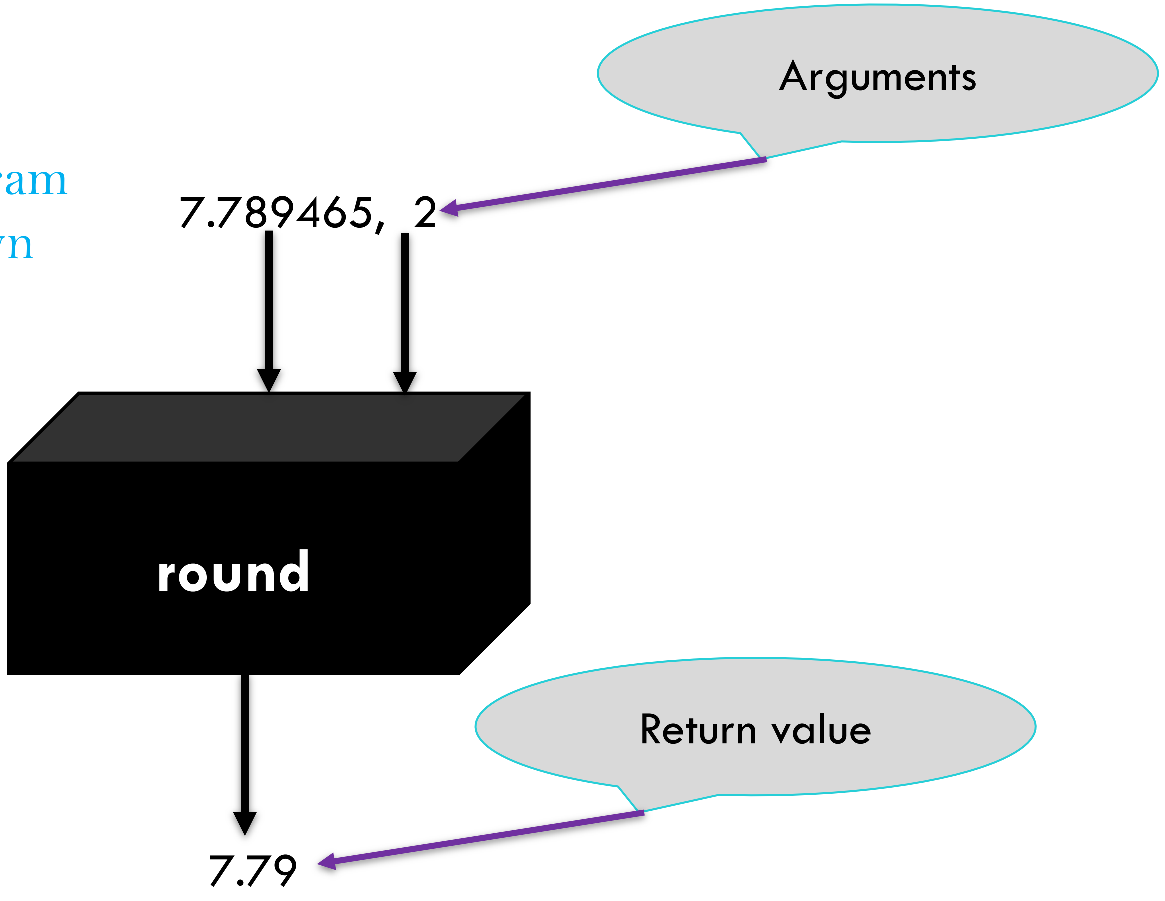
- Functions that return values are “fruitful” functions
- **Void functions** are **”not fruitful”**

To Function or Not to Function …

- Organise your code into "paragraphs" – to capture a complete thought and "name it"
- Don't repeat yourself – make it work once and then reuse it
- If something gets too long or complex, break it up into small logical chunks and put those chunks in functions
- Make a library of common stuff that you do over and over – perhaps share this with your peers …

Round() Function as a Black Box

The term black box is used in program with given specification but unknown implementation or outcome.



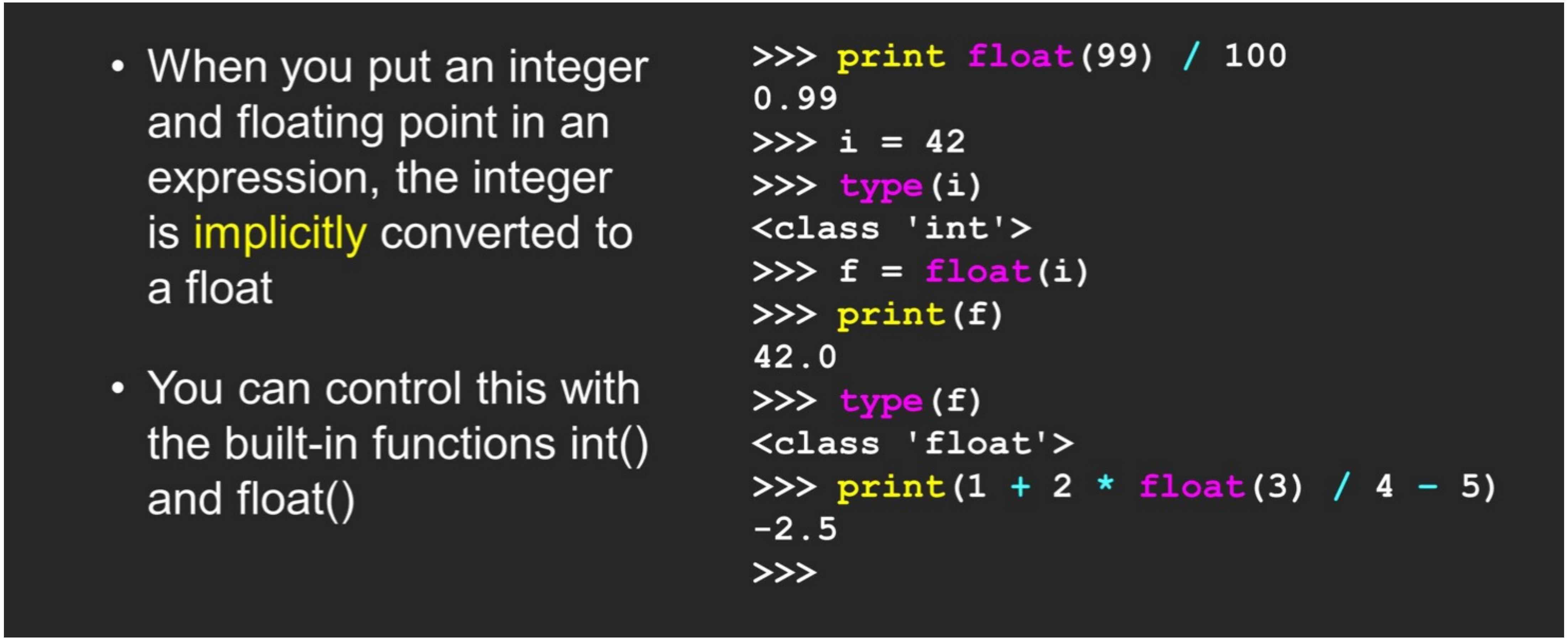
Revisiting Type Conversion

Type Conversion : Casting

Let's revisit the type conversion from the previous lectures.

Based on previous studies, we know that casting is the process of converting from one data type to another. Changing the value of one primitive type e.g., from *int* to *float* or *float* to *int* etc.

Type Conversions

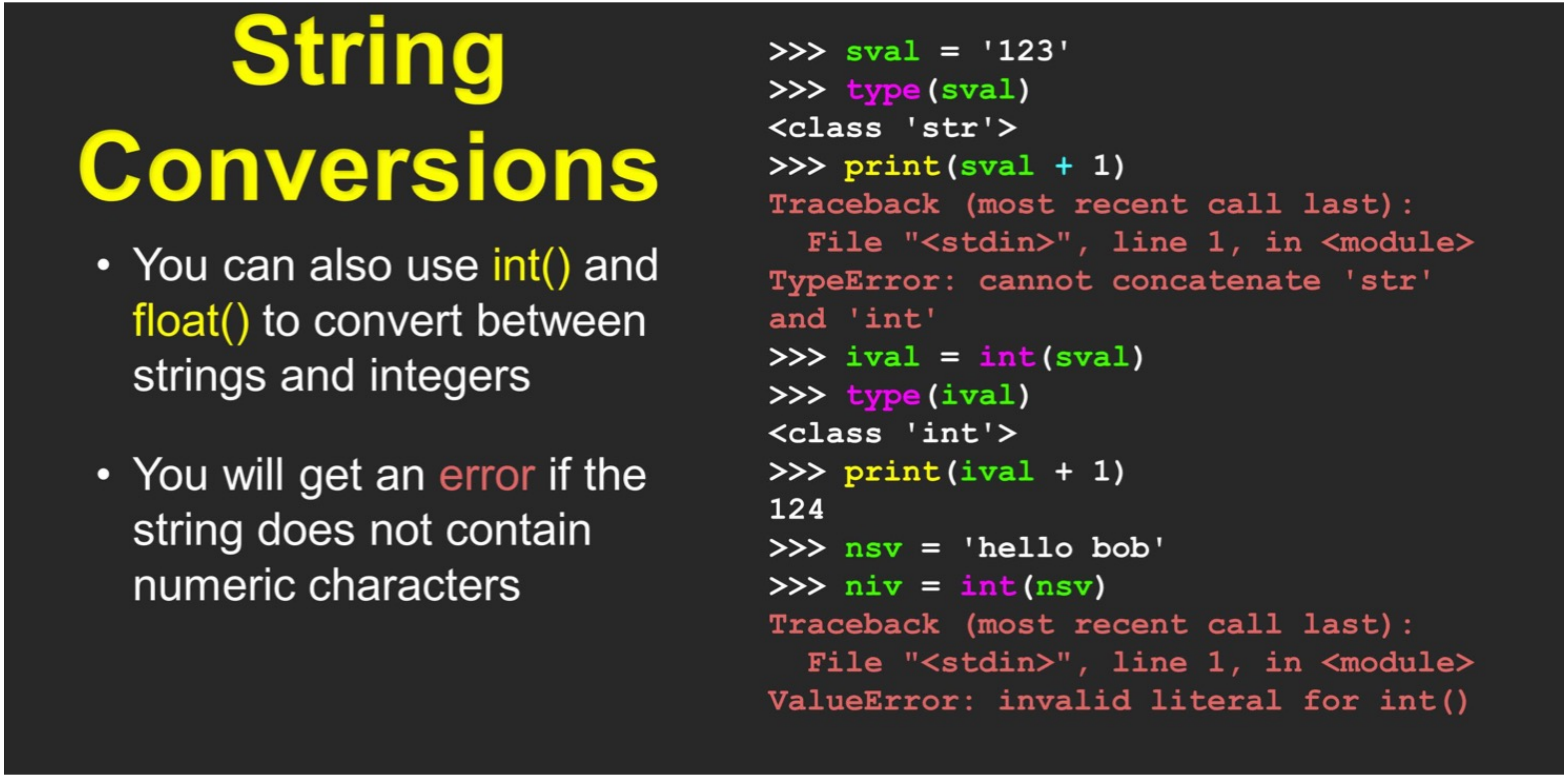


- When you put an integer and floating point in an expression, the integer is implicitly converted to a float

- You can control this with the built-in functions int() and float()

```
>>> print float(99) / 100
0.99
>>> i = 42
>>> type(i)
<class 'int'>
>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class 'float'>
>>> print(1 + 2 * float(3) / 4 - 5)
-2.5
>>>
```

Another Conversion



String Conversions

- You can also use int() and float() to convert between strings and integers
- You will get an error if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str'
and 'int'
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
```

Exercise

Rewrite your pay computation with time-and-a-half for overtime and create a function called computepay which takes two parameters (hours and rate).

```
Enter Hours: 45
Enter Rate: 10

Pay: 475.0
```

475 = 40 * 10 + 5 * 15

Local scope & Global scope

Variable Scope

As your programs get larger and contain more variables, you may encounter problems where you cannot access a variable that is defined in a different part of your program, or where two variable definitions conflict with each other. In order to resolve these problems, you need to be familiar with the concept of *variable scope*.

The **scope** of a variable is the part of the program in which you can access it. For example, the scope of a function's parameter variable is the entire function. In the following code segment, the scope of the parameter variable *sideLength* is the entire **cubeVolume** function but *not* the main function.

Example

The scope of a variable is the part of the program in which a variable is accessible.
There are two types of scope of a variable: (1) Local scope and (2) Global scope

```
def main() :
    print(cubeVolume(10))

def cubeVolume(sideLength) :
    return sideLength ** 3

main()
```

Scope of variable

Local Scope

A variable created in the nested function indicated that the scope exists in and centric to that function. Assuming we have a variable declared at the innermost function, then the scope of that variable is only in the innermost function. Local scope are variables created inside a function and which value are only bound within the scope in which it is defined in.

```
# A variable created inside a function. i.e. local scope or local variable
def display():
    x = 123  # Local variable
display()
```

Not - Local Scope

Local variables are declared inside the function or inside block (or body) of the function. Local variables can only be accessible or evaluated only within the declared function/method or block and not outside that function or block (or body of the function).

```
# A variable created inside a function. i.e. local scope or local variable
def display():
      x = 123  # Local variable
      print(x) # This will print because x is a local variable
display()
print(x) # This will not print because, the print statement is outside the function, therefore the x in this case is not part of the local variable x declared inside the function.
```

Declared Outside

Output: NameError: name ‘x’ is not defined

Local Variable

A variable that is defined within a function is called a **local variable**. When a local variable is defined in a block, it becomes available from that point until the end of the function in which it is defined. For example, in the code segment below, the scope of the square variable is highlighted.

```
def main() :
    sum = 0
    for i in range(11) :
        square = i * i
        sum = sum + square
    print(square, sum)
```

Local Variable in Loops

A loop variable in a for statement is a local variable (e.g. ‘**i**’). As with any local variable, its scope extends to the end of the function in which it was defined:

```
def main() :
    sum = 0
    for i in range(11) :
        square = i * i
        sum = sum + square
  print(i, sum)
```

Loop variable

Global Scope

A global variable is a variable that is generated in the **main body** of the program. They are available in any scope and could be used by any function, method or code. The **global variable or global scope** are declared outside of a function. *This type of variable is defined in the main body of Python program*. It is available **throughout the program**. All functions or blocks can easily access and use a global variable.

```
# A variable created outside of a function. i.e. global scope or global variable
 x = 123  # Global variable
def display():
     print(x) # This print statement will also access and display the value of x i.e.123
display()
print(x) # Printing & displaying global variable outside the function
```

Local and Global Variables

We can declare both local and global variables within a Python program. The evaluation will be done by Python interpreter separately and according to the declaration.

```
# A program to evaluate local and global variables

x = 450 # Global scope and global variable outside my_evaluation() fucntion
def my_evaluation():
    x = 250 # Local scope with my_evaluation() function
    print("Printing the value of local variable: ",x) # Printing local variable
my_evaluation()
print("Printing the value of global variable:",x) # Printing global variable
```

Output:

```
Printing the value of  local variable:  250
Printing the value of  global variable: 450
```

Scope Error!!

Here is an example of a scope problem:

```
def main() :
    sideLength = 10
    result = cubeVolume()
    print(result)
def cubeVolume() :
    # This is where the error is
    return sideLength ** 3 # Error
main()
```

Note the scope of the variable sideLength. The cubeVolume function attempts to read the variable, but it cannot—the scope of sideLength does not extend outside the main function. The remedy is to pass it as an argument to the cubeVolume() function.

Global Statement

The local variable's scope is limited to the block in which it is defined. *Furthermore, when we construct a variable within a function, that variable is local to that function.* This means, it can only be used and accessible within that function. However, we can construct a **global** variable within a function by using the Python *global* keyword. Let's look at an example of a global statement using the *global* keyword to help us better understand the definition properly.

```
# Program with a global keyword

def my_function():
    global x
    x = 199 # x is not a local variable in this case, its is a global variable
# because it has been initially declared global using the global keyword
(global x)
my_function()
print(x) # Now we can print x as a global variable
```

Output:

```
199
```

Types of Functions

Recursive Functions

A recursive function is a function that calls itself. This is not as unusual as it sounds at first. Suppose you face the arduous task of cleaning up an entire house. You may well say to yourself, **"I'll pick a room and clean it, and then I'll clean the other rooms."** In other words, the clean-up task calls itself, but with a simpler input. Eventually, all the rooms will be cleaned.

In Python, a recursive function uses the same principle.

Recursive function has two very important cases:

- **Base case –** When the condition within the base case reaches the end of the iteration (or zero etc) then the standard case is triggered to return the result.
- **Standard case –** The standard case is triggered as soon as the condition of the base case is no longer **True** and the iterative condition becomes **False**.

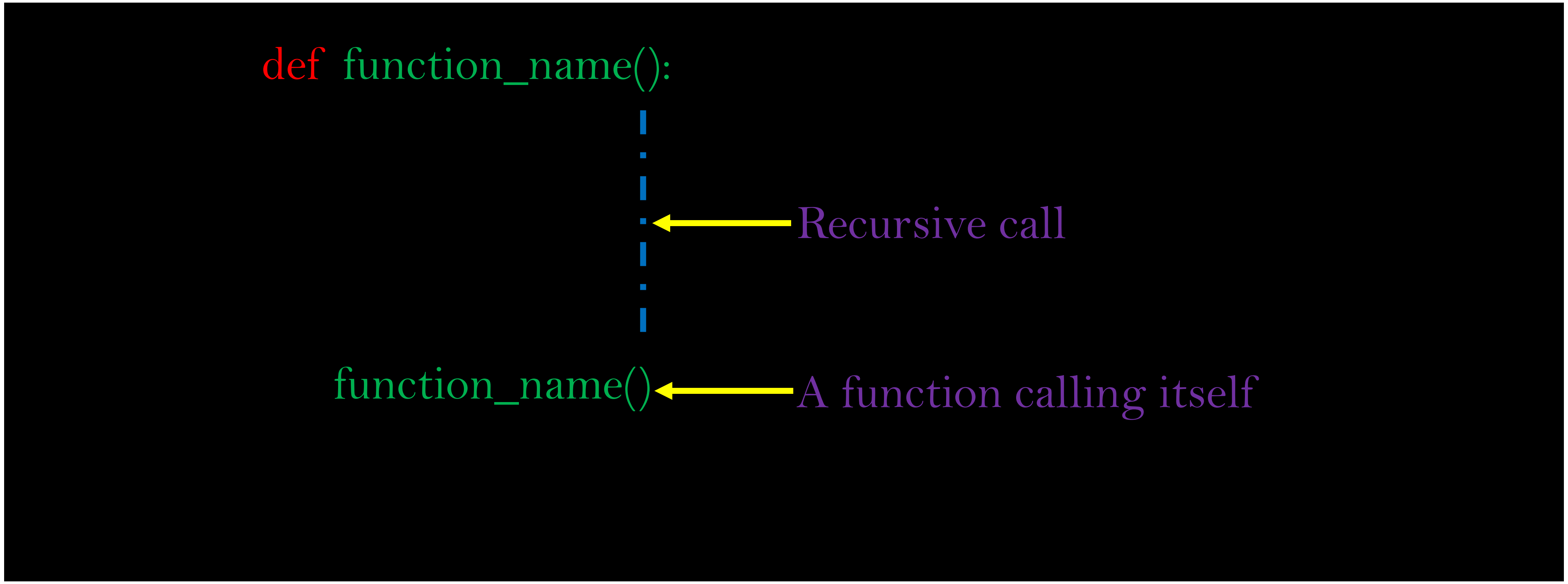
Recursive Functions

Let's describe this in a simpler language. When a function called itself, we call this recursion. We will define a function and call the same function name in their own definition (or body of the function block), that means the code block is recursive in nature.

Recursion is a very essential and important construct in Python. It is an essential technique in programming. Key fact we should know about recursive function is that:

- The recursive function should be terminated once it has finished its task
- If the terminating condition set within the code is **True**, this means our recursive code is working properly.
- However, if we forgot to set the terminating condition, then our program will run into infinite execution.

Recursive Functions - Syntax



```
def function_name():
    .
    .
    .
    function_name()
```

Advantages

The following are the advantages of a recursive function:

- *Recursive functions can be broken down into smaller chunks of problems (sub-problem) that could easily be manageable within a program.*
- *Creating a recursive sequence is a lot simpler than using a nested conditional or loop iteration.*
- *The implementation of recursive functions makes the program code to appear simple and efficient.*

Disadvantages

The following are the disadvantages of a recursive function:

- *Recursive call uses a lot of our computer memory time. This is why they are costly to use.*
- *Debugging recursive functions is a difficult task.*
- *The logic behind the recursive function (or recursion) can be difficult to understand at times.*

Example 1 – Fibonacci series

```python
"""
A program to compute Fibonacci series to n terms
Fibonacci sequence is made up of the sum of the initial number in the series and the next to
arrive at the Fibonacci number series.
"""

def fibonacci(n): # Recursive function
    # base case
    if n <= 1:
        return n
    # standard case (recursive call)
    else:
        return(fibonacci(n-1) + fibonacci(n -2))


term = int(input(" Please enter a positive number: "))
if term <= 0:
    print("Invalid input! Please enter only positive value.")
else:
    print("*** Fibonacci series *** ")
    for n in range(term):
       print(fibonacci(n))
```

Example2 – Factorial Numbers

```python
"""
A recursive program to compute the Factorial of a number
"""

def factorial(n): # recursvie function
# base case
  if n == 1:
    return n
# standard case ( recursive case)
  else:
    return n * factorial(n - 1)


n = int(input("Please enter the number: "))

# a condition to check whether the input is valid or not
if n < 0:
    print("Invalid input! Please enter a positive number.")

elif n == 0:
  print("Factorial of number 0 is 1")
else:
  print("factorial of number", n, "=", factorial(n))
```

Lambda functions

An anonymous function or a function defined without a name is called **lambda functions**. We have seen throughout the lecture that ideally functions are define using the def **keyword**. In the case of lambda or anonymous functions, they are defined using the **lambda** **keyword**.

Syntax of a lambda function:

```
lambda arguments: expression
```

Lambda Function

There are some essential points you should know about lambda functions:

- The number of arguments for a lambda function is infinite, but there is only one expression that should be declared
- If we need to use an anonymous function to perform a short operation, then we should use the lambda functions. It is very useful for short time task (e.g. to compute a sum of numbers).
- Lambda function does not necessary include a "return" statement. Instead, it always includes an expression that is returned. **The print() function is used to display the result.**

Lambda Function: Example 1

Let's look at a few examples of a lambda function declaration.

```python
"""
A lambda function program to add 78 to an argument n,
and return the result of the sum.
"""

addition = lambda n: n + 78 # lambda argument and expression
print(addition(25)) # display the result of the sum
# Note the value of 'n' is passed as an argument into the addition(n=25)
```

Output: **103**

Lambda Function: Example 2

Let's look at a few examples of a lambda function declaration.

```
"""
A lambda function program to find the product of 78 to an argument n,
and return the result of the product.
"""

product = lambda n: n * 78 # lambda argument and expression

# display the result of the product of the two numbers
print("The product is = {}".format(product(25)))
# Note the value of 'n' is passed as an argument into the product(n=25)
```

Output: The product is = **1950**

Lambda Function: Example 3

Let’s look at a few examples of a lambda function declaration.

```
"""
A lambda function program to sum three numbers (floating-point and integers).
"""
addition = lambda number1, number2, number3 : number1 + number2 + number3

# displaying the result in floating-point
print(”The sum of the three numbers (floating-point) is = %.2f"%(addition(66.15,
96.78, 60.21)))

# displaying the result in integer

print(”The sum of the three numbers (integer) is = %d"%(addition(66.15, 96.78,
60.21)))
```

Output:

```
The sum of the three numbers (floating-point) is = 223.14
The sum of the three numbers (integer) is = 223
```

Summary

Function Redefinition

Python provides a facility for the redefinition of functions because Python is dynamic in nature. Redefinition of function means redefining an already defined function. Let's take an example to understand the concept of the function redefinition.

Homework – Run This Program

```python
"""
A program for function redefinition
using string from time (strftime) function. This is used to convert date and time
objects to their string representation
-- gmtime to convert the time in seconds
"""

from time import gmtime, strftime

def show(msg): # function definition
    print(msg)
show("Ready ... to print current GMT time")

def show(msg): # function redefinition
    print(strftime("%H:%M:%S", gmtime()))
    print(msg)
show("Processing ...")
```

Summary

In this lecture, *we have discussed function and how they are important for modular programming. We looked at the parameter passing method into a function*. We have also discussed local and global variables, and their scope. We discussed two types of functions: recursive and lambda functions. We also discussed functions with arbitrary arguments.

<u>**Points about functions:**</u>

- Function blocks start with a **def** keyword after the function name and parentheses.
- Each function has an indented code block that begins with a colon (**:**)
- Return None is equivalent to a return statement with no arguments.
- The number of arguments in a lambda function is unlimited, but there is only one expression.

Summary

- Python interpreter reads the source code , it reads each function and each statement
- The statements in a function definition are not executed until the function is called
- Any statement not in a function, on the other hand, is executed as it is encountered.
- Therefore, it is imperative that you define each function before you call it.

Summary

- By convention, when defining and using function in Python, it is good programming practice to place all statements into functions, and to specify one function as the main function.
- Any legal name could be use, but **main** is commonly used with most programming languages.
- Remember that both functions are in the same source code file.

Key Facts – Python Function

- We looked at arguments
- We looked at parameters.
- Built-In Functions
  - Type conversion (int, float)
  - String conversions
- Fruitful functions and void (non-fruitful) functions
- Why do we need to use functions? (breaking programs into smaller units/chunks)

Further Reading

You can read further this week's lecture from the following chapters:

- Python for Everyone (3/e) : By **Cay Horstmann** & **Rance Necaise**
  - Chapter 5 Functions
- Learning Python (5th Edition): By **Mark Lutz**
  - Part IV Functions and Generators: Chapter 17 Scopes; Chapter 18 Arguments; Chapter19 Advanced Function Topics
- Basic Core Python Programming: By **Meenu Kohli**
  - Chapter 3 Numbers, Operators and In-built Functions

Next Lecture 5

Lecture 5 - Strings