CSE4IP
Lecture 7 –Functions

Ayad Turky

Last time

- What are loops
  - A loop is a control structure that allows a program to execute one or more statements repeatedly as long as certain condition is satisfied
- Sometimes you need to run a bit of code a bunch of times
  - For that, Python provides the *iteration statements*
    - `while and for`
- Be aware of nested loops

Functions in computer programming

- A function in Python is simply a set of statements that are grouped together and given a name
  - A function can, but is not required to, return a value as a result
  - A function can, but is not required to, accept one or more parameters as input

Why use functions?

- They make code easier to read, understand, and debug
  - Rather than putting all your code into one giant program, you can break it up into smaller, more well-defined, and more manageable chunks
- They can reduce code duplication
  - If you have to do the same thing multiple times in a program, you can write it in a function and call the function, instead of writing out the code every time
- They promote code reuse
  - If you write a useful function once, you can use it in other programs

Python Functions

- There are two kinds of functions in Python.

  - Built-in functions that are provided as part of Python - print(), input(), type(), float(), int() ...

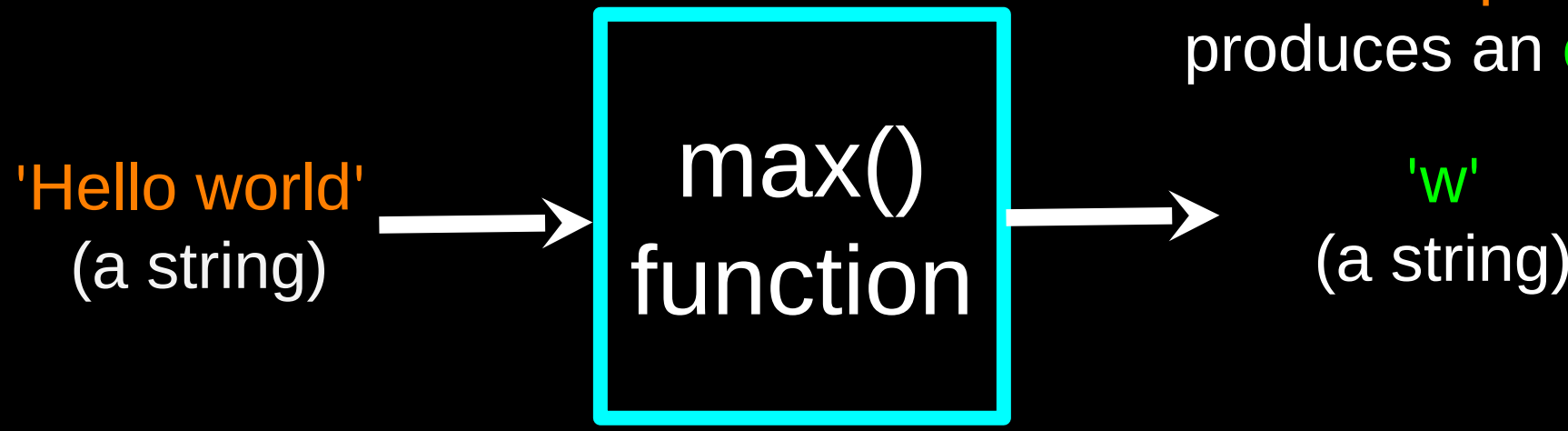
  - Functions that we define ourselves and then use

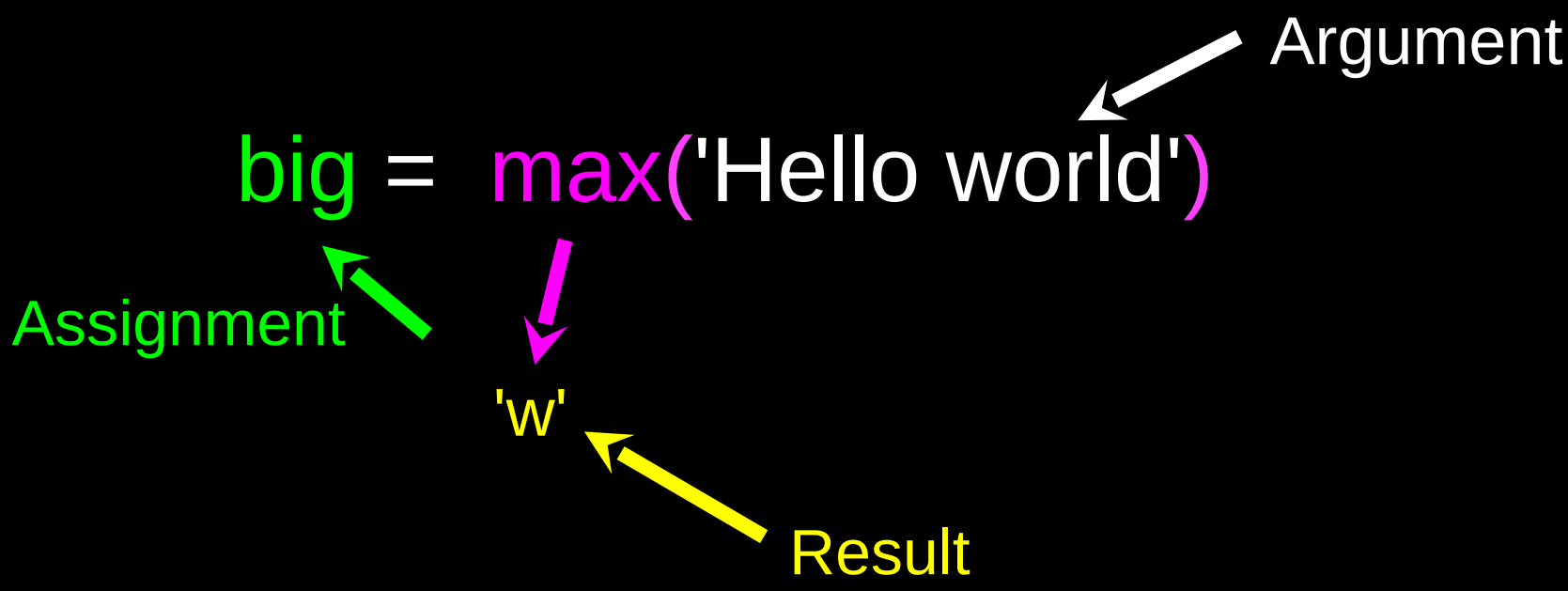
- We treat the built-in function names as “new” reserved words (i.e., we avoid them as variable names)

Built-in functions: 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.






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

>>>
```

Functions of Our Own…

Function Definition

- In Python a function is some reusable code that takes arguments(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

Function Definition

- `def <function name> (<parameters>):`

```
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

- We call a function using the function name with the general syntax

```
<function name>(<parameters>)
```

General syntax

- We define a function using the general syntax

```
def <function name> (<parameters>):
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

function header

function body

- We call a function using the general syntax

```
<function name>(<arguments>)
```

Function

- The parameters in the function definition are known as **formal parameters** or **formal arguments**
- The parameters in the function call are known as **actual parameters** or **actual arguments**
- Most people use the terms “parameter” and “argument” interchangeably

Function

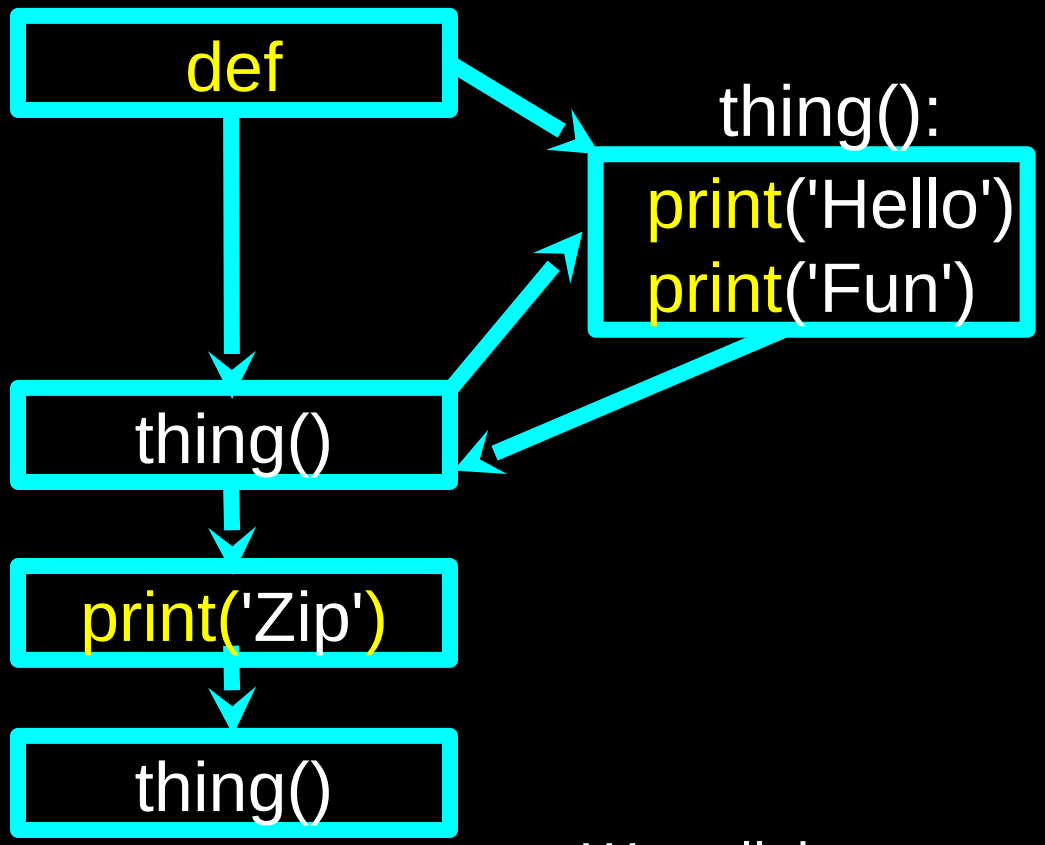
- Some use this convention

  Use “parameter” to refer to formal parameters (in function definition)

  Use “argument” to refer to the actual parameters (provided with the call)

- We will, in general, adopt this convention, for convenience

Stored (and reused) Steps



Program:

```
def thing():
    print('Hello')
    print('Fun')

thing()
print('Zip')
thing()
```

Output:

Hello
Fun
Zip
Hello
Fun

We call these reusable pieces of code “functions”

First example – Without parameters

- A function that displays “Hello” in a “box” (a line above and a line below)
- Here is the function’s definition

```
def boxHello():
    print("------")
    print("Hello!")
    print("------")
```

First example – Without parameters

- Call the function

```
boxHello()
```

- Output:

```
------
Hello!
------
```

```
def boxHello():
    print("------")
    print("Hello!")
    print("------")
```

Second Example – With Parameters

- A function that takes a message and displays it in a box

```
def boxMessage(message):
    length = len(message)
    line = "-" * length
    print(line)
    print(message)
    print(line)
```

Second Example – With Parameters

- Call the function

```
boxMessage("Hello, World!")
```

- Output:

```
------------
Hello, World!
------------
```

```
def boxMessage(message):
    length = len(message)
    line = "-" * length
    print(line)
    print(message)
    print(line)
```

Third Example – Returning a value

- A function can return a value using the **return** statement
- We can call a value-returning function in an expression

Example: Function that takes a radius of a circle and returns its area

```
def circleArea(radius):
    from math import pi
    area = pi * radius ** 2
    return area
```

Example: Function that takes a radius of a circle and returns its area

- Call the function

```
area1 = circleArea(10)
print("area of circle with radius 10:", area1)
```

```
area of circle with radius 10: 314.1592653589793
```

```
area2 = circleArea(20)
print("area of circle with radius 20:", area2)
```

```
area of circle with radius 20: 1256.6370614359173
```

```
def circleArea(radius):
    from math import pi
    area = pi * radius ** 2
    return area
```

Argument Passing and Transfer of Control

- **When we make a function call, two important things happen**
  - Passing of arguments
  - Transfer of control

Example

- A function to calculate the volume of a cylinder

```
def cylinderVolume(diameter, height):
    from math import pi
    area = pi * diameter**2 / 4
    volume = area * height
    return volume
```

Example

- Call the function in an expression

```
def cylinderVolume(diameter, height):
    from math import pi
    area = pi * diameter**2 / 4
    volume = area * height
    return volume
```

```
d = float(input("Enter the can's diameter: "))

h = float(input("Enter the can's height: "))

vol = round(cylinderVolume(d, h), 2)

print("The can’s volume is", vol)
```

Sample run

```
Enter the can's diameter: 10
Enter the can's height: 15
The can's volume is 1178.1
```

Example

Call: `vol = round(cylinderVolume(d, h), 2)`

In the sample run, when the call is made

1. The value 10 of argument **d** is passed to the parameter **diameter**
2. The value 20 of argument **h** is passed to the parameter **height**
3. The control is transferred to the function and the statements in the function are executed
4. When the execution of the function finishes,
   - ✓ the control is transferred back to the main program
   - ✓ the value returned by the function is rounded and assigned to variable **`vol`**

Advantages of Functions

- Functions provide two main advantages

➢ Task decomposition: They allow us to divide a task into smaller subtasks

➢ Program structuring : They allow us to make our programs well-structured – with meaningful components and clear relationships among them

Example – temperature conversion

- We can convert temperature in Fahrenheit into Celsius accurately with formula

$$c = (F - 32) * (5/9)$$

- We can also do that approximately with formula

$$c = (F - 30) / 2$$

- Suppose we want a table with 3 columns:
  - The first displays temperatures in F
  - The second temperatures in C, accurately
  - The third, temperatures in C, approximately

Example – temperature conversion

We can write our program using functions like this

```
def convert(f):
    c = (f - 32) * 5 / 9
    return c
def estimate(f):
    c = (f - 30 ) / 2
    return c
# main
for f in range(20, 120, 4):
    print(f, convert(f), estimate(f))
```

```
20 -6.666666666666667 -5.0
24 -4.444444444444445 -3.0
28 -2.222222222222223 -1.0
32 0.0 1.0
36 2.2222222222222223 3.0
40 4.444444444444445 5.0
44 6.666666666666667 7.0
48 8.888888888888889 9.0
52 11.111111111111111 11.0
56 13.333333333333334 13.0
60 15.555555555555555 15.0
64 17.777777777778 17.0
68 20.0 19.0
72 22.22222222222222 21.0
76 24.444444444444443 23.0
80 26.666666666666668 25.0
84 28.888888888888889 27.0
88 31.111111111111111 29.0
92 33.333333333333336 31.0
96 35.55555555555556 33.0
100 37.777777777778 35.0
104 40.0 37.0
108 42.22222222222222 39.0
112 44.444444444444 41.0
116 46.66666666666664 43.0
```

Task decomposition

- We have broken the main task into three subtasks
  - We define two functions separately
  - We use them to display a table
- We can test each of the functions separately
- If necessary, we can modify them separately

Task decomposition

- Suppose we only need the temperatures two decimal places, we can modify the two functions separately

```
def convert(f):
    c = (f - 32) * 5 / 9
    c = round(c, 2)
    return c

def estimate(f):
    c = (f - 30 ) / 2
    c = round(c, 2)
    return c

for f in range(20, 120 + 1, 4):
    print(f, convert(f), estimate(f))
```

Program structures

- The program is well-structured
- There are three clearly defined parts
  - One to perform the conversion accurately (function convert)
  - One to perform the conversion approximately (function estimate)
  - And one to use these two functions to construct and display a table

Program structures

- We can carry this structuring a little further by defining a main function and calling it:

```
def convert(f):
    c = (f - 32) * 5 / 9
    c = round(c, 2)
    return c
def estimate(f):
    c = (f - 30 ) / 2
    c = round(c, 2)
    return c
def main():
    for f in range(20, 120 + 1, 4):
        print(f, convert(f), estimate(f))
main()
```

Reuse

- The functions, once defined, can be used (called) whenever we need them

```
>>> currentTemp = 75
>>> convert(currentTemp)
23.89
```

```
def convert(f):
    c = (f - 32) * 5 / 9
    c = round(c, 2)
    return c
```

Use a module

- We can use the program as a **module**, and import it to another program to make further use of the functions
- Suppose we call the program conversion.py, we can import it as shown below

```
from conversion import convert

temp = int(input("Enter current temperature in F:"))

print("the temperature in C:", convert(temp))
```

Use a module

- Note that when we import module conversion, the main function is called and executed (this can be annoying)
- We can suppress this call when we import the module by putting the call inside a special if statement

Use a module: Example

```
def convert(f):
    c = (f - 32) * 5 / 9
    c = round(c, 2)
    return c

def estimate(f):
    c = (f - 30 ) / 2
    c = round(c, 2)
    return c

def main():
    for f in range(60, 101, 5):
        print(f, convert(f), estimate(f))

if __name__ == "__main__":
    main()
```

Use a module: Example

```
if __name__ == "__main__":
    main()
```

- is True when we run `conversion.py` as a program
- is False when we import it as a module

Documenting function

- We can document a function by providing a string after the function heading
- The string is known as a docstring
- The docstring is made use of by the help function

Documenting function: Example

```
def convert(f):
    """
    Converts temperature in F to C (accurately),
    rounds the result to 2 decimal places
    """
    c = (f - 32) * 5 / 9
    c = round(c, 2)
    return c

def estimate(f):
    """
    Converts temperature in F to C approximately,
    rounds the result to 2 decimal places
    """
    c = (f - 30 ) / 2
    c = round(c, 2)
    return c
```

Documenting function: Example

```
>>> import conversion

>>> dir(conversion)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'convert', 'estimate']

>>> help(conversion.convert)
Help on function convert in module conversion2:
convert(f)
    Converts temperature in F to C (accurately),
    rounds the result to 2 decimal places
```

Keyword Arguments

- When we call a function, we can match the actual arguments with the formal parameters by positions or by keywords

- Thus, regarding a specific call, we can have positional arguments and keyword arguments

Example

```
def f(a, b):
    print("a:", a, "b:", b)

# match arguments and parameters by positions
f(10, 20)

# match by keywords
f(b = 20, a = 10)
```

Output:

```
a: 10 b: 20
a: 10 b: 20
```

Example

- We can mix positional and keyword arguments.

```
def f(a, b, c):
    print("a:", a, "b:", b, "c:", c)


# Mix positional and keyword arguments
f(10, c = 30, b = 10)
```

Output:

```
a: 10 b: 10 c: 30
```

Default parameters

- A parameter can have a default value
- Such a parameter is called a default parameter

Example

```
# x and y are default parameters

def f(a, b, x = "XX", y = "YY"):
    print("a:", a, ", b:", b, ", x:", x, ", y:", y)

# Override all defaults
f(10, 20, "apple", "orange")

# accept all defaults
f(10, 20)

# accept some defaults
f(10, 20, y = "orange")
```

```
Output:
a: 10 , b: 20 , x: apple , y: orange
a: 10 , b: 20 , x: XX , y: YY
a: 10 , b: 20 , x: XX , y: orange
```

Rules

- When defining a function, non-default parameters must be **before** default parameters

```
def f(a = 10, b , x = "XX", y = "YY"):
    print(a, b, x, y)
```

Illegal function definition: Default parameter **a** before non-default parameter **b**

Rules

- When calling a function, positional arguments must be **before** keyword arguments

```
def f(a, b , x = "XX", y = "YY"):
    print(a, b, x, y)


f(10, x = "apple", 20)    # error
```

Illegal function call: Keyword argument `x = apple` **before** positional argument `20`

Local Variables and Global Variables

Local Variables

- A function can **define** its own variables with the assignment statement.

- Such a variable is known as a local variable

Example (with 4 local variables)

```
def canSurfaceArea(radius, height):
    """
    calculates the total surface area of a can:
     area of the top + area of the bottom
     + area of the side
    """
    from math import pi
    topArea = pi * radius**2
    bottomArea = topArea
    sideArea = 2 * pi * radius * height
    area = topArea + bottomArea + sideArea
    return area
```

Scope of variables

- The scope of a variable is the region where the variable is **available**

- The scope of a local variable of a function extends **only** to the **end** of the **function**

  Therefore, it is **not accessible** outside the function

Example

```
def canSurfaceArea(radius, height):
    from math import pi
    topArea = pi * radius**2
    bottomArea = topArea
    sideArea = 2 * pi * radius * height
    area = topArea + bottomArea + sideArea
    return area

myCanArea = canSurfaceArea(4, 10)
print("my can's area:", myCanArea)
print("its top area: ", topArea)          # error
```

Global variables

- A function can access and use a variable which is created **outside** the function

- Such a variable is known as a global variable (from the perspective of the function)

Example

```
def f(x):
    print(message)  # global variable
    print(x)

# main
message = "hello"
f(10)
```

Output:

```
hello
10
```

Notes

- When the function is executed, the global variable message must have already been defined

- Otherwise, we would have a run-time error, as in the next example.

Example

```
def f(x):
    print(message)  # global variable
    print(x)

# main program
# message = "hello"
f(10)
```

Output:

NameError: name 'message' is not defined

Rules

- If a function **defines a variable** (with the assignment statement),

  then by default, the variable is a **local variable**

- If a function wants to change a global variable with an assignment, it must use the global keyword to indicate that it wants to treat the variable as a **global variable**

Example

```
def f():
    global message
    print("Point A:", message)
    message = "Violets are blue"
    print("Point B:", message)

# main
message = "Roses are red"
f()
print("Point C:", message)
```

Output:

```
Point A: Roses are red
Point B: Violets are blue
Point C: Violets are blue
```

Example

```
def f():

    print("Point A:", message)
    message = "Violets are blue"
    print("Point B:", message)

# main
message = "Roses are red"
f()
print("Point C:", message)
```

Output:

UnboundLocalError: local variable 'message' referenced before assignment

A closer look at argument passing

The object view of Python

- To clearly understand argument passing, we need to know about how Python treats objects
- In Python everything is an object
  - A number is an object
  - A string is an object
  - A list is an object
  - etc.

The object view of Python

- An object is stored at a location (address) in memory
- When we assign an object to a variable

  <variable> = <object>

  We actually cause the variable to **point to** the object

  We also say: the variable **refers to** the object

- The address of the object that a variable pointing to can be obtained by the **`id`** function

Example

```
x = 10
print("address of x before assignment", id(x))
x = 20
print("address of x after assignment:", id(x))
y = x
print("address of y:", id(y))
```

Sample output:(which varies from run to run):

```
address of x before assignment 1697835072
address of x after assignment: 1697835232
address of y: 1697835232
```

Argument passing

- When we pass an argument, we make the parameter **point to** the **same object** as the argument

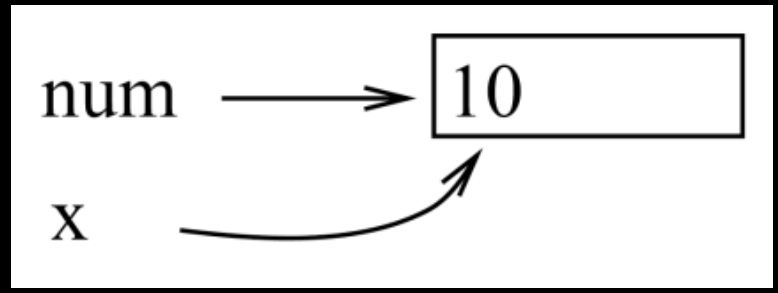
Example

```
def f(x):
    print("address of x:", id(x))
    print("x:", x)

# main
num = 10
print("num:", num)
print("address of num: ", id(num))
f(num)
Sample output:

num: 10
address of num: 1580329024
address of x: 1580329024
x: 10
```

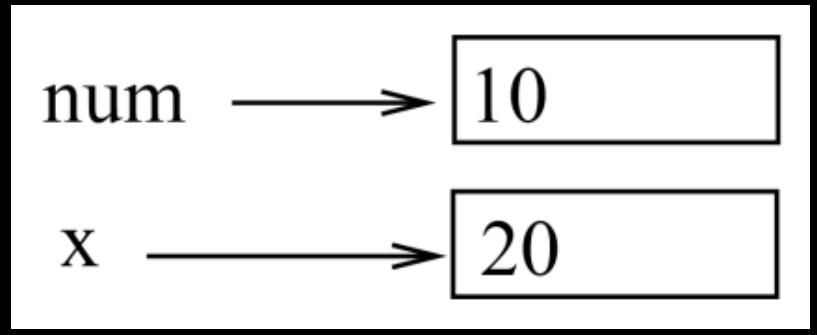
Example

```
def f(x):
    print("x before assignment:", x)
    x = 20
    print("x after assignment:", x)

# main
num = 10
print("num before call:", num)
f(num)
print("num after call:", num)

Output:
num before call: 10
x before assignment: 10
x after assignment: 20
num after call: 10
```

After argument passing



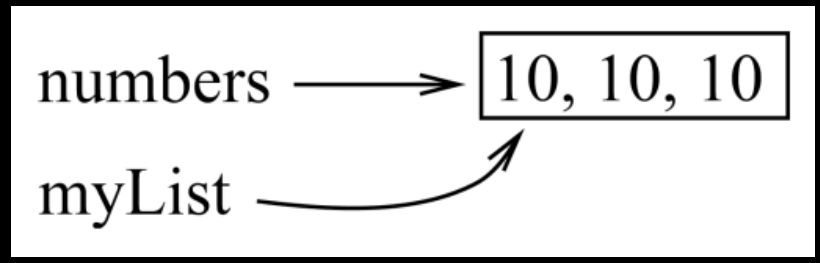
After x = 20



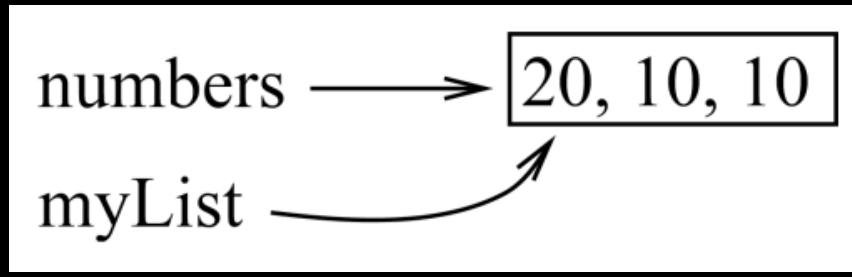
Example

```
def f(myList):
    print("myList before assignment:", myList)
    myList[0] = 20
    print("myList after assignment:", myList)

#  main
numbers = [10, 10, 10]
print("numbers before call:", numbers)
f(numbers)
print("numbers after call:", numbers)

Output:
numbers before call: [10, 10, 10]
myList before assignment: [10, 10, 10]
myList after assignment: [20, 10, 10]
numbers after call: [20, 10, 10]
```

After argument passing



After myList[0] = 20



Recap

- A *function* in Python is simply a set of statements that are grouped together and given a name

  – A function can, but is not required to, return a value as a result
  – A function can, but is not required to, accept one or more parameters as input

- Function can **define** its own variables with the assignment statement.

  – Such a variable is known as a local variable

- A function can access and use a variable which is created **outside** the function

  – Such a variable is known as a global variable (from the

Acknowledgements

- Acknowledgement to https://www.py4e.com/ and to Charles R. Severance (www.dr-chuck.com) of the University of Michigan School of Information.
- Also acknowledgement to Rick Skarbez (CSE1PES/CSE5CES).

Thank you.

Be well.

latrobe.edu.au

La Trobe University
CRICOS Provider Code Number 00115M