Computational Thinking with Programming

Lecture - 9

Functions: Program routes, Calling Value Returning Functions, Calling Non- value Returning Functions

@cse_bennett @csebennett

Today…

- Last Session:
  - Lists in Python.
- Today's Session:
  - Functions:
    - Why using functions?
    - Formal definition, parameters, local and global scopes of variables, return values, and pass-by-value
- Hands on Session with Jupyter Notebook:
  - We will practice on the user defined function in Jupyter Notebook.

Multiple-line Snippets and Functions

- So far, we have been using only one-line snippets in the interactive mode, but we may want to go beyond that and execute an entire *sequence of statements*

- Python allows putting a sequence of statements together to create a brand-new command or *function*

These *indentations* are necessary to indicate that these two statements belong to the same *block of code*, which belongs to this function

```
>>> def hello():
...     print("Hello")
...     print("Programming is fun!")
...
>>>
```

Indentations Are Mandatory

- If indentations are not provided, an error will be generated

```
>>> def hello():
... print("Hello")
  File "<stdin>", line 2
    print("Hello")
        ^
IndentationError: expected an indented block
>>>
```

Invoking Functions

- After defining a function, we can call (or *invoke*) it by typing its name followed by parentheses

This is how we invoke our defined function *hello()*; notice that the two print statements (which form one code *block*) were executed in sequence!

```
>>> def hello():
...     print("Hello")
...     print("Programming is fun!")
...
>>> hello()
Hello
Programming is fun!
>>>
```

Invoking Functions

- Invoking a function without parentheses will NOT generate an error, but rather the location (*address*) in computer memory where the function definition has been stored

```
>>> def hello():
...     print("Hello")
...     print("Programming is fun!")
...
>>> hello
<function hello at 0x101f1e268>
>>>
```

Parameters

- We can also add *parameters* (or *arguments*) to our defined functions

person is a parameter; it acts as an *input* to the function *hello(…)*

```
>>> def hello(person):
...     print("Hello " + person)
...     print("Programming is fun!")
...
>>> hello(“Bennett")
Hello Bennett
Programming is fun!
>>>
```

Multiple Parameters

- We can add multiple parameters and not only one

```
>>> def hello(person, course):
...     print("Hello " + person + “ from “ + course)
...     print("Programming is fun!")
...
>>> hello(“Bennett“, “15-110”)
Hello Bennett from 15-110
Programming is fun!
>>>
```

Parameters with Default Values

- In addition, parameters can be assigned default values

```
def print_func(i, j = 100):
    print(i, j)

print_func(10, 20)
```

Run

```
10 20
```

```
def print_func(i, j = 100):
    print(i, j)

print_func(10)
```

Run

```
10 100
```

```
def print_func(i, j = 100):
    print(i, j)

print_func()
```

Run

ERROR

Modularity and Maintenance

- Consider the following code

```
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear Fred")
print("Happy birthday to you!")
```

Can we write this program with *ONLY two* prints?

```
def happy():
    print("Happy birthday to you!")

def singFred():
    happy()
    happy()
    print("Happy birthday, dear Fred")
    happy()

singFred()
```

More ***modular*** & ***maintainable***– changing anything in the lyric “Happy birthday to you!” requires making a change at only one place in happy(); thanks to the happy function!

Extensibility and Readability

- Consider the following code

```
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear Fred")
print("Happy birthday to you!")
```

What if we want to sing a verse for Lucy right after Fred?

```
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear Fred")
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear Lucy")
print("Happy birthday to you!")
```

What if we utilize functions?

Extensibility and Readability

- Consider the following code

```
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear Fred")
print("Happy birthday to you!")
```

What if we want to sing a verse for Lucy right after Fred?

```
def happy():
    print("Happy birthday to you!")


def sing(name):
    happy()
    happy()
    print("Happy birthday, dear " + name)
    happy()

sing("Fred")
sing("Lucy")
```

Easy to ***extend***, more ***readable***, and necessitates ***less typing!***

Formal Definition of Functions

- Formally, a function can be defined as follows:

```
def <name>(<formal-parameters>):
        <body>
```

- The **<name>** of a function should be an identifier and **<formal-parameters>** is a (possibly empty) list of variable names (also identifiers)

- **<formal-parameters>** and all *local* variables declared in a function are *only* accessible in the **<body>** of this function

- Variables with *identical* names declared elsewhere in a program are distinct from **<formal-parameters>** and local variables inside a function's <body>

Local Variables

- Consider the following code

```
def func1(x, y):
        #local scope
        z = 4
        print(x, y, z)

func1(2, 3)
print(x, y, z)
```

Run

```
2 3 4
Traceback (most recent call last):
  File "func1.py", line 6, in <module>
    print(x, y, z)
NameError: name 'x' is not defined
```

x, y, and z belong solely to the ***scope*** of func1(...) and can <u>*only*</u> be accessed inside func1(…); z is said to be *local* to func1(…), hence, referred to as a ***local variable***

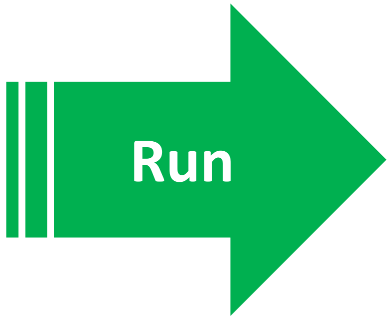
Global Variables

- Consider the following code

```
#global scope
x = 100

def func2():
        print(x)

func2()
print(x)
```



```
100
100
```

x is said to be a ***global variable*** since it is defined within the *global scope* of the program and can be, subsequently, accessed inside and outside func2()

Local vs. Global Variables

- Consider the following code

```
x = 100

def func3():
        x = 20
        print(x)

func3()
print(x)
```

Run

```
20
100
```

The global variable x is distinct from the local variable x inside func3()

Parameters vs. Global Variables

- Consider the following code

```
x = 100

def func4(x):
        print(x)

func4(20)
print(x)
```

Run

```
20
100
```

The global variable x is distinct from the parameter x of func4(…)

The global Keyword

- Consider the following code

```
def func5():
        global x
        x = 20
        print(x)

func5()
print(x)
```

Run

```
20
20
```

The ***global*** keyword *binds* variable x in the global scope; hence, can be accessed inside and outside func5()

Getting Results From Functions

- We can get information from a function by having it *return* a value

```
>>> def square(x):
...     return x * x
...
>>> square(3)
9
>>>
```

```
>>> def cube(x):
...     return x * x * x
...
>>> cube(3)
27
>>>
```

```
>>> def power(a, b):
...     return a ** b
...
>>> power(2, 3)
8
>>>
```

Pass By Value

- Consider the following code

```
>>> def addInterest(balance, rate):
...     newBalance = balance * (1+rate)
...     return newBalance
...
>>> def test():
...     amount = 1000
...     rate = 0.05
...     nb = addInterest(amount, rate)
...     print(nb)
...
>>> test()
1050.0
>>>
```

Pass By Value

- Is there a way for a function to communicate back its result without returning it?

```
>>> def addInterest(balance, rate):
...     newBalance = balance * rate
...     balance = newBalance
...
>>> def test():
...     amount = 1000
...     rate = 0.05
...     addInterest(amount, rate)
...     print(amount)
...
>>> test()

>>>
```

What will be the result?

Pass By Value

- Is there a way for a function to communicate back its result without returning it?

```
>>> def addInterest(balance, rate):
...     newBalance = balance * rate
...     balance = newBalance
...
>>> def test():
...     amount = 1000
...     rate = 0.05
...     addInterest(amount, rate)
...     print(amount)
...
>>> test()
1000
>>>
```

Why 1000 and NOT 1050.0?

Pass By Value

- The function only receives the *values* of the parameters

1000

addInterest(…) gets ONLY the *value* of amount (i.e., 1000)

```
>>> def addInterest(balance, rate):
...     newBalance = balance * rate
...     balance = newBalance
...
>>> def test():
...     amount = 1000
...     rate = 0.05
...     addInterest(amount, rate)
...     print(amount)
...
>>> test()
1000
>>>
```

Python is said to *pass parameters by value*!

Pass By Value vs. Returning a Value

- Consider the following code

```
def increment_func(x):
      x = x + 1

x = 1
increment_func(x)
print(x)
```

```
1
```

```
def increment_func(x):
      x = x + 1
      return x

x = 1
x = increment_func(x)
print(x)
```

```
2
```

More on the Print Function

- There are different forms of the print function
  1) print(), which produces a blank line of output

```
>>> print()


>>>
```

More on the Print Function

- There are different forms of the print function
  2) print(<expr>, <expr>, ..., <expr>), which indicates that the print function can take a sequence of expressions, separated by commas

```
>>> print(3+4)
7
>>> print(3, 4, 3+4)
3 4 7
>>> print("The answer is ", 3 + 4)
The answer is  7
>>> print("The answer is", 3 + 4)
The answer is 7
```

More on the Print Function

- There are different forms of the print function
  3) print(<expr>, <expr>, ..., <expr>, end = "\n"), which indicates that the print function can be modified to have an *ending text* other than the default one (i.e., \n or a new line) after all the supplied expressions are printed

Notice how we used the end parameter to allow multiple prints to build up a single line of output!

```
>>> def answer():
...     print("The answer is:", end = " ")
...     print(3 + 4)
...
>>> answer()
The answer is: 7
>>>
```

Thank You

?