CSC 110: Week 4, Lecture 2

Dr. Bodine

Last class…

- Introduced functions
  - “Subprograms”
  - Why we use them
- Defining functions in python
  - `def` statements
- Parameters and scope

Today

- Continue our discussion of scope
- Parameters
  - Formal vs actual
  - Modifying parameters
- Results
  - Void and value-returning
- Working examples of functions

Starting with our homework example…

```
>>> def testfunction(inputvalue):
        #Calculate square and reciprocal
        square = inputvalue**2
        reciprocal = 1/inputvalue

        #print square and reciprocal
        print("The square of",inputvalue, "is", square)
        print("The reciprocal of", inputvalue,"is",reciprocal)


>>> testfunction(3)
The square of 3 is 9
The reciprocal of 3 is 0.3333333333333333
>>>
```

Now with functions!

```
>>> def squarefunction(value):
        print("The square of",value,"is",value**2)


>>> def reciprocalfunction(value):
        print("The reciprocal of",value,"is",1/value)


>>> def testfunction(value):
        squarefunction(value)
        reciprocalfunction(value)


>>> testfunction(3)
The square of 3 is 9
The reciprocal of 3 is 0.3333333333333333
```

Now with functions!

```
>>> def squarefunction(value):
        print("The square of",value,"is",value**2)


>>> def reciprocalfunction(value):
        print("The reciprocal of",value,"is",1/value)


>>> def testfunction(value):
        squarefunction(value)
        reciprocalfunction(value)


>>> testfunction(3)
The square of 3 is 9
The reciprocal of 3 is 0.3333333333333333
```

But now, what if we want to extend this to print the square of the reciprocal? Print statements don't mean we can't compose these functions are get our result → Not modular coding…need something else

Results from functions

- Previous examples *suspiciously* focused on printing results to screen
- Returns allow us to use what we compute in functions in other places in the program, not just print to screen
  - Remember, we need a handle on this we want to use
- Void functions
  - Functions without any returns
  - When tasks are complete, return to point of call
- Return-value functions
  - Give us a handle on what comes out of computations or manipulations

Our homework example with function results

```
>>> def squarefunction(value):
        return value**2

>>> squarefunction(4)
16
>>> squarefunction(5)
25
>>>
```

```
>>> def reciprocalfunction(value):
        return 1/value

>>> reciprocalfunction(4)
0.25
>>> reciprocalfunction(5)
0.2
>>>
```

Assigning variables with functions

```
>>> def squarefunction(value):
        return value**2

>>> squarefunction(4)
16
>>> squarefunction(5)
25
>>>
```

```
>>> def reciprocalfunction(value):
        return 1/value

>>> reciprocalfunction(4)
0.25
>>> reciprocalfunction(5)
0.2
>>>
```

```
>>> x = squarefunction(4)
>>> x
16
>>> y = reciprocalfunction(4)
>>> y
0.25
>>>
```

Putting it all together…

```
>>> def squarefunction(value):
        return value**2

>>> def reciprocalfunction(value):
        return 1/value

>>> def testfunction(value):
        squaredvalue = squarefunction(value)
        reciprocalvalue = reciprocalfunction(value)
        print("The square of",value,"is", squaredvalue)
        print("The reciprocal of",value,"is",reciprocalvalue)


>>> testfunction(3)
The square of 3 is 9
The reciprocal of 3 is 0.3333333333333333
>>>
```

Now, add functionality…

```
>>> def testfunction2(value):
        squaredvalue = squarefunction(value)
        reciprocalvalue = reciprocalfunction(value)
        squaredreciprocalvalue = squarefunction(reciprocalfunction(value))

        print("The square of",value,"is", squaredvalue)
        print("The reciprocal of",value,"is",reciprocalvalue)
        print("The square of the reciprocal of",value,"is",squaredreciprocalvalue)


>>> testfunction2(3)
The square of 3 is 9
The reciprocal of 3 is 0.3333333333333333
The square of the reciprocal of 3 is 0.1111111111111111
```

What happens when we call a function?

- At a function call:

1. Suspends execution at the point of call
2. Formal parameters get assigned values based on actual parameters
3. Body of function is executed
4. Control returns to the program at the point just after function call

- What about when we define function?

Let’s examine a working set of functions

```
>>> def squarefunction(value):
        return value**2

>>> def reciprocalfunction(value):
        return 1/value

>>> def testfunction(value):
        squaredvalue = squarefunction(value)
        reciprocalvalue = reciprocalfunction(value)
        print("The square of",value,"is", squaredvalue)
        print("The reciprocal of",value,"is",reciprocalvalue)


>>> testfunction(3)
The square of 3 is 9
The reciprocal of 3 is 0.3333333333333333
>>>
```

Another example

```
>>> def myfunction1(invalue1, invalue2):
        return invalue1/invalue2

>>> def myfunction2(invalue):
        return invalue**2

>>> def myfunction3(int1, int2):
        print("Doing compuatations with", int1,"and",int2)
        result1 = myfunction1(int1,int2)
        result2 = myfunction2(result1)
        result3 = myfunction2(myfunction1(int1,int2))
        print("result1 =",result1)
        print("result2 =",result2)
        print("result3 =",result3)


>>>
>>> myfunction3(10,4)
Doing compuatations with 10 and 4
result1 = 2.5
result2 = 6.25
result3 = 6.25
```

Returns vs modifying parameters

- We have used returns to get output of functions but what if we want to modify parameters?
  - Example, want my function to increment a value based on another value
  - Requires an understanding of variable assignment in the language of interest

- Not always possible to DIRECTLY modify parameters
  - Depends on language
  - Depends on parameter type

Returns vs modifying parameters

- We have used returns to get output of functions but what if we want to modify parameters?
  - Example, want my function to increment a value
  - Requires an understanding of variable assignment in the language of interest

- Parameter passed by value:
  - Formal parameters are assigned actual parameter values at call time
  - Function does not have access to variable storing the actual parameters
  - Related to scope!!!

- Parameters passed by reference:
  - Gives direct access to the variable, not just the value
  - Allowed in some languages and not in others…

Modifying parameters in python

- We can start with a simple test
- What happens?
- Why do we get this result?

```
>>> x = 2
>>> def myfunction(mypar):
        mypar = mypar**2


>>> myfunction(x)
```

Modifying parameters in python

- We can start with a simple test
- What happens?
- Why do we get this result?

```
>>> x = 2
>>> def myfunction(mypar):
        mypar = mypar**2


>>> myfunction(x)
>>> x
2
>>>
```

Modifying parameters in python

- Could we modify it with a return?

- How would we have to call the function to make that work?

Modifying parameters in python

- Could we modify it with a return?
- How would we have to call the function to make that work?

```
>>> x = 2
>>> def myfunction(mypar):
        mypar2 = mypar**2
        return mypar2

>>> x = myfunction(x)
>>> x
4
>>>
```

Is it always the case with python parameters?

- What about lists?

```
>>> testlist = [1,2,3,4,5]
>>> def incrementlist(inList):
        for i in range(len(inList)):
                testlist[i]=testlist[i]+1


>>> incrementlist(testlist)
>>> testlist
[2, 3, 4, 5, 6]
```

- Why does this work? What is different about lists

Example:

- `def toNumbers(strList)` `strList` is a list of strings, toNumbers modifies the list by converting the entries to numbers

- Do we need a return or is a void function acceptable?

- Could we technically do it either way?

- What features do we have to use to accomplish this goal?

Example:

```
>>> def toNumbers(strList):
        for i in range(len(strList)):
                strList[i] = eval(strList[i])


>>>
>>> testlist = ['1','12','3.4']
>>> testlist
['1', '12', '3.4']
>>> toNumbers(testlist)
>>> testlist
[1, 12, 3.4]
>>> type(testlist[0])
<class 'int'>
>>>
```

Challenge problem:

- Book example: Write definitions for these functions

`sumN(n)` Returns the sum of the first n natural #s

`sumNCubes(n)` Returns the sum of the cubes of the first n natural #s

- Then use theses functions in a program that prompts the user for n and prints out the sum of the first n natural numbers and the sum fo the cubes of the first n natural numbers