More on Functions

October 5, 2022

Administrative notes

Monday we introduced functions

Key things from Monday's lecture

- Functions are the fourth type of control flow we have covered this semester (sequential, conditional, iterative and now functions)
- Python program execution always starts with the main program - the if __name__ == "__main__" statement
- Functions cause the program control to hop to a different location; execute the code there; and then return
- Parts of a function (definition, name, parameters, function body, function call, arguments)
- return statement - how you actually return a value to the calling main program or function

What we'll add today:

- return statements and what happens if there isn't one
- Local variables, global variables and scope
- What happens when your parameter is a list

`return` statements

return statements are the only direct way to get values back to the main program

(or the function that called this function)

return statements are optional, but the function ALWAYS returns a value

- If you DON’T have a return statement, the function returns the special value None, which is of type NoneType

The function ends the instant the first return statement is executed

A function with multiple “return” statements

```
def factorial(x):
    if x > 0:
        product = 1
        for i in range(1,x+1):
            product *= i
        return product
    else:
        return 1

if   name   == "__main__":
    num = int(input("Enter the number whose factorial will be
calculated"))
    fact = factorial(num)
    print(num, " factorial is ", fact)
```

Notes:

- Only one of these two return statements can be executed
- An integer will be returned
- That integer will be stored in “fact” in the main program

Using returned values

If you want to use the value that's returned, you have to do so in the calling program/function

If the function contains:

```
def factorial(num):
    product = 1
    for i in range(num):
        Product *= i
    return product
```

The calling program should use the value:

```
fact = factorial(5)
```

Or

```
sum = factorial(7) + factorial(2)
```

or...

Example - calculate the “fourth root” of a positive number

```
# the “fourth root” of a number is that number raised to the (¼) power
#
def fourth_root(num):
    ans = num**0.25  # or you could use 1/4
    return ans

#set the variable’s value, then call the function
original_number = 81
final_number = fourth_root(original_number)
print(final_number)
```

Another example: reverse_word

```
#  First the function definition
def reverse_word (word)
# now the code. DON'T FORGET TO INDENT!!
        i = 0
        reversed_word = ''  #Do not use the function name here!!!!
        while i < len(word):
                reversed_word = reversed_word + word [i]
        print (" The word ", word, " reversed is ", reversed_word)
```

This function does its work independently; it doesn't return any values for use in the program. It would be more useful if this function did return a value. Let's revise it.

Revised reverse_word

```
#  First the function definition
def reverse_word (word):
  # now the code. DON'T FORGET TO INDENT!!

  i = 0
  reversed_word = ''  #Do not use the function name
here!!!!
  while i < len(word):
     reversed_word = reversed_word + word[-(i+1)]
     i += 1
  print (' The word ', word, ' reversed is ',
reversed_word)


...
```

```
# Now the call is
animals = ["cat", "Australian cattle dog",
"duckbilled platypus", "ocelot", "zebra"]
for critter in animals:
        r_word = reverse_word(critter)
# now do whatever you want to with the
# reversed word in the main program
```

The function stops when it executes a return statement

A function stops executing once it executes a return statement. Let's look at fourth_root() again. This time we'll add some code after the "return" statement

```
# the "fourth root" is the number raised to the ¼ power

def fourth_root(num):
        ans = num**0.25
        return (ans)
# The following statement is not going to execute, because the function has already ended
# due to the "return" statement
        print("the code has successfully executed")
```

None and NoneType

The original reverse_word function was:

```
def reverse_word (word):
  # now the code. DON'T FORGET TO INDENT!!

  i = 0
  reversed_word = ''  #Do not use the function name here!!!!
  while i < len(word):
     reversed_word = reversed_word + word[-(i+1)]
     i += 1
  print (' The word ', word, ' reversed is ', reversed_word)
```

There's no "return()" statement, so what does this return?

A special value called "None" which is of type "NoneType"

The value “None”

Python defines a special value “None” which is of type “Nonetype”

If a function does not otherwise return a value, it returns “None”

- Functions do not have to contain a return statement. Any function without a return statement returns “None.”
- If a function’s return statement is not executed, the function returns “None.”
- You can explicitly tell the function to return “None.”

A common error is to have a function return None when you expected it to return something else.

- You’ll see an error message like:
- `Error; type ‘Nonetype’ is not iterable`

“None”

If a function returns “None” you will generally have trouble using that value in your code, unless you use it for error checking. Note: not checking whether your function returned “None” is a very common error, and can make debugging difficult. Check for that in your calling code.

In the code in your function, having

```
return None
return
```

And no return statement at all have the same effect - your function returns a value of None.

Error checking using “None”

```
# Function definition


def fourth_root(num):
        If num >= 0::
                ans = num**(1 / 4)
                return (ans)
        else:
                return None
# we could have also said
#               return
# or just omitted the entire else: clause and
# have no return statement at all. The code
# works the same
```

```
# get the original value from the user
# then call the function
original_number = float(input("enter a number"))
done = False
while not(done):
  final_number = fourth_root(original_number)
  if final_number != None:
     print("The fourth root of", original_number, end = "")
     print(" is ", final_number)
     done = True
  else:
     original_number = float(input("we were serious about needing a
non-negative number"))
```

Variable scope

global variable: a variable that is declared and used in the main program

- It CAN be used anywhere in the program - main, functions, etc. but SHOULD only be used in the main program
- Using a global variable in a function is error-prone; the bugs will drive you nuts. For your own sanity, if you need a value in a function, pass it as an argument/parameter

CONSTANTS are global, but their use in functions is allowed

Local variable; a variable declared and used in a function

- Python only lets you access that local variable within that function
- Nested functions make this “interesting”

```
# an example of scope in Python
DIGITS =
['0','1','2','3','4','5','6','7','8','9']

def get_input():

   def error check(s):
       is number = True
       for char in s:
           if not char in DIGITS:
               is number = False
       return is_number

   num = input("Enter a positive integer;
we will calculate that number's
factorial")
   is num = error check(num)
   while not is num:
       print("Error; that is not a
positive integer")
       num = input("Enter a positive
integer; we will calculate that number's
factorial")
       is num = error check(num)
   # now we know that num is a positive
integer and we can cast it
   num = int(num)
   return num
```

```
def factorial(n):
   product = 1
   for i in range(1,n+1):
       product *= i
   return product

def print answer(n, fact):
   print(n, "factorial is ", fact)

# now the main program
if   name   == "__main__":
   num = get input()
   answer = factorial(num)
   print_answer(num, answer)
```

Symbol tables

get_input

num	string	address_101
error_check	function	address_102
is_num	boolean	address_103

Local Variables

Main program

num	int	address1
answer	int	address2
get_input	function	address_3
factorial	function	address_4
print_answer	function	address_5

Global Variables

What happens if your parameter is a list

Lists are mutable variables

Python does NOT use call by value for them; it uses call by reference

- That means it passes the address in memory of the list
- The function uses the EXACT SAME locations in memory as the main program
- That means your function can have unexpected side effects
    - So be careful!!

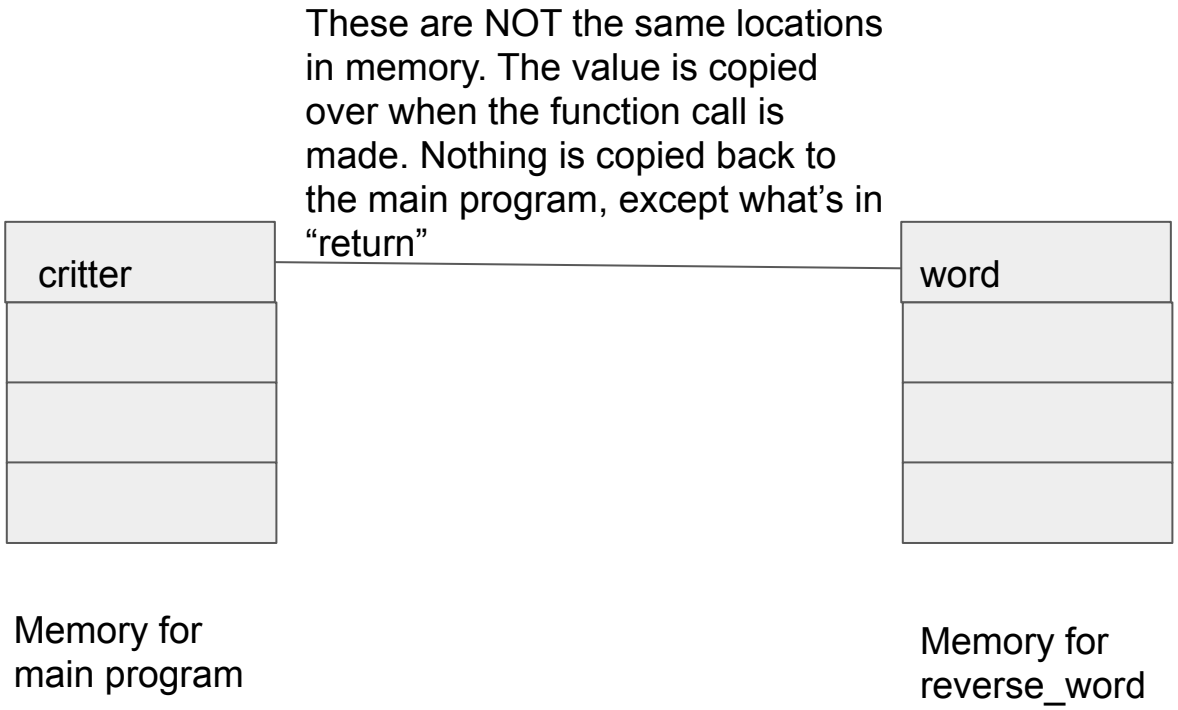
calling by value: ***immutable*** variables as parameters

```
#  First the function definition
def reverse_word (word):
# now the code. DON'T FORGET TO INDENT!!
        i = 0
        reversed_word = ‘’
        while i < len(word):
                reversed_word = reversed_word + word
[-(i+1]
#       return(reversed_word)
…       word = reversed_word
```

```
# Now the call is
animals = [“cat”, “Australian cattle dog”,
“duckbilled platypus”, “ocelot”, “zebra”]
for critter in animals:
        reverse_word(critter)
        print(critter)
# now do whatever you want to with the
# reversed word in the main program
```

This doesn't work in Python!! (It does work in some other languages, so if you've got experience with doing this, put it out of your mind for now)

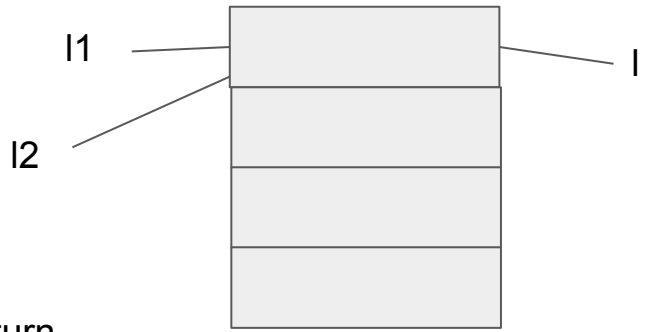
Why doesn’t this work?



Mutable variables

With mutable variables, like lists, you don't copy the value in a new location in memory - you simply set a pointer to the SAME location in memory, and modify in place

When you call the function, you get a pointer to the SAME location in memory



When you return from the function, you get a pointer to the SAME location in memory

The result is that when you modified the list in the function, you (inadvertently) modified the original list in the main program

An example - a list as a parameter

```
def add_to_list(l):
    for i in range(len(l)):
        l[i] += 10
    return


if __name__ == "__main__":
    my_list = [1,2,3,4,5]
    add_to_list(my_list)
    print(my_list)
```

Why does this happen?

- Because my_list in the main program and l in the function are the same location in memory
- When you change one, you change the other