More on Functions

October 4, 2023

Administrative notes

Homework 3 submissions…

Homework 4

Sample Exam 1

Continuing on with functions

Using the returned value from a function

The calling function gets the returned value as if the call were a variable.

Consider the built in function **`len(some_list)`**:

```
if len(some_list) > 2:
    print(len(some_list))
    list_length = len(some_list)
```

The only place you can't use the returned value is on the left side of an assignment statement

```
len(some_list) = x + 5 # NOT PERMITTED
```

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"))
```

How parameters are passed in memory

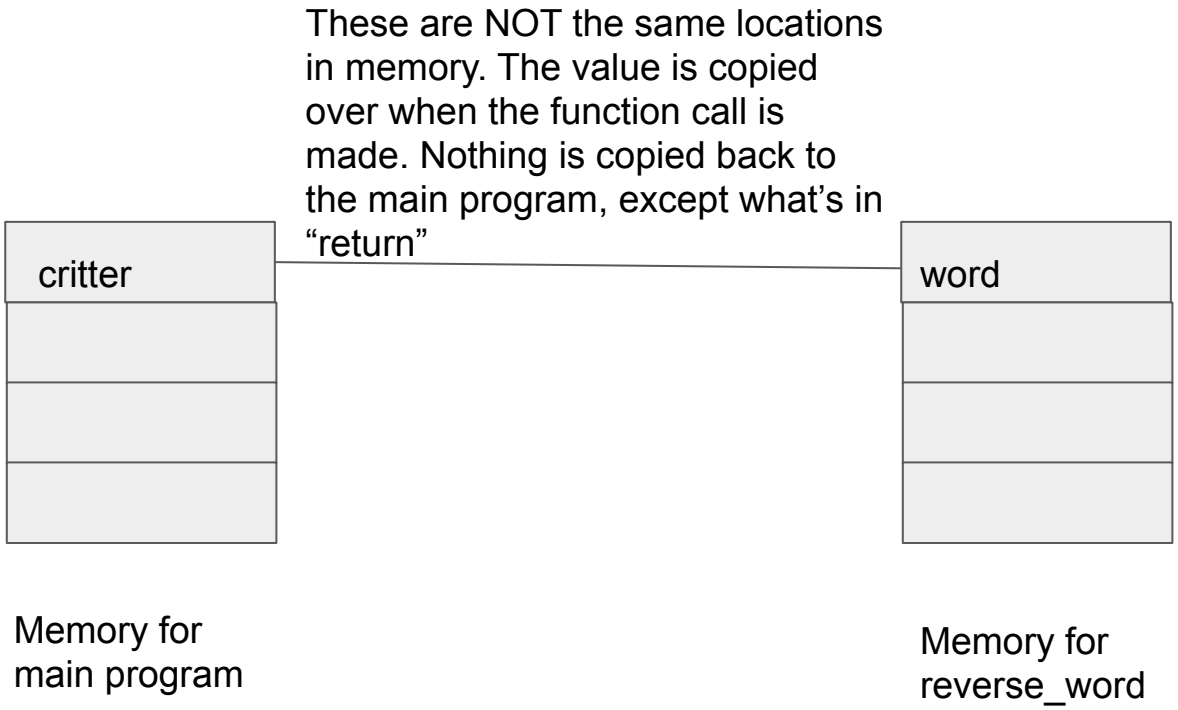
Calling by value

```
#  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?



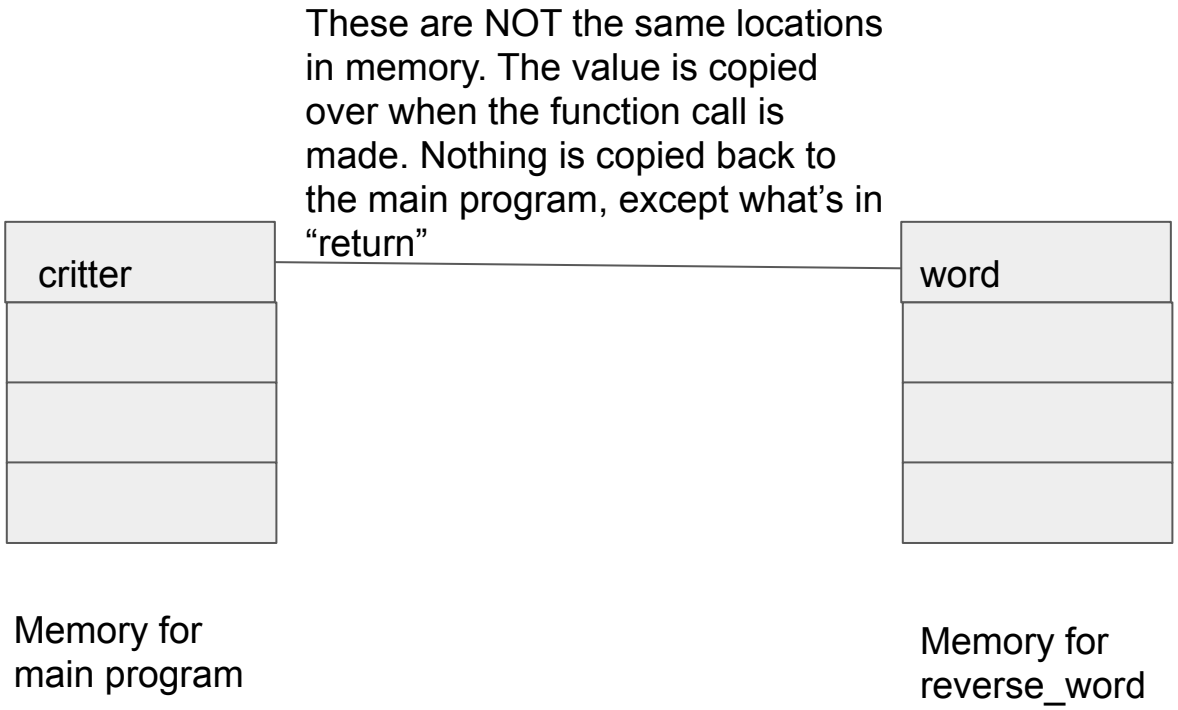
calling by value: ***immutable*** variables as parameters

```
#  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?



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!!

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)
```

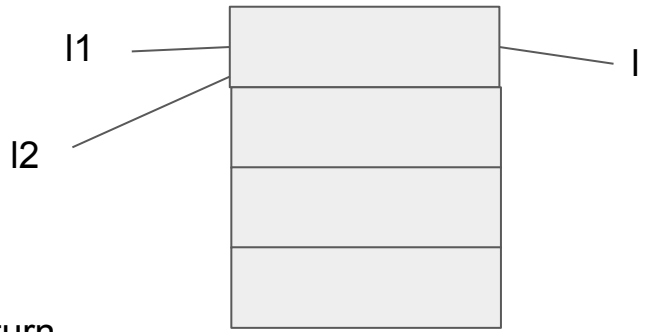
Why does this happen?

- 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

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

Scope of a variable

Scope of variables

Scope means where a variable can be directly seen and used

Variables defined in the main program can be seen everywhere in the program. BUT - you should only directly use them in the main program. If you need their values in a function, pass them as arguments. If you use a main program variable directly in a function, without passing it as an argument, that's called a "global variable" and it will cost you major points!!!

Variables defined in functions can only be seen and used in the functions where they are defined. These are called "local variables."

Examples of variable scope

Calculating someone's age in days - slight modification of Monday's code

Pseudocode - program computes someone's age in days.

- Ask the user for today's date
- Ask the user for user's birthdate
- Calculate the year and day_of_year for today, and for the user's birthdate
  - 3/10/2021 69th day of 2021
  - Born: 4/15/1989 105th day of 1989 subtract 4/15/1989 from 3/10/2021 - 3 subtractions; 2 carries - complicated; easy to get wrong. 105 1989 subtracted from 69 2021 easier to get right (2 subtractions, one carry)
- Subtract; print the answer

```
# 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

Importing functions in Python

Importing modules and functions in Python

Python comes with some “builtin” functions such as len(), print(), input(),...

There are tons of other functions that have already been written by others, and which are free to you to use in your programming career.

- There’s no need to rewrite a function if you know somebody else has already written it

You get access to that code by using the import() function

import()

import() tells the Python interpreter that you want access to a module that you know about, and the functions in that module

A “module” is Pythonic for a group of functions made available. Other languages might call this a “library” or a “package.”

Import random

Imports a module that contains a bunch of functions all related to the generation and management of “random” numbers

Note: the module must be present on your computer for “import” to work. If you get an error message saying the module does not exist, you’ll have to install it.

Using a function in a module

Once you have imported a module, you can use its functions in your program

```
random.randint(1,25)
```

Generates a random integer between 1 and 25, inclusive

You can use this just like any other function:

```
for i in range(10):
    r_num = random.randint(1,25)
    print(r_num)
```

How do you know what functions are in a module?

...and what parameters to use to call them?

This is where the ability to search the web is your friend. :-)

All the common modules are documented out there in Python-land, along with their Application Programming Interfaces (APIs)

- Which is a fancy way of saying “descriptions of how to call a function, what the parameters are, what the parameters mean and what the return values are.”