Functions - part II

October 14, 2020

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."

The return statement

return is the statement used to pass values from the function back to the main program (or back to the calling function)

The syntax is

```
return variable_name # you can optionally put the variable name in ()
```

For this class, a function will return one value.

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 = fact(7) + fact(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**(¼)
        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 = ‘’
    while i < len(word):
        reversed_word = reversed_word + word [i]
    return(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
```

Program control in a function

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**(¼)
        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”)
```

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

None and NoneType

The original reverse_word function was:

```
#  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)
```

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 = sqrt(sqrt(num))
                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"))
```

More on calling by value

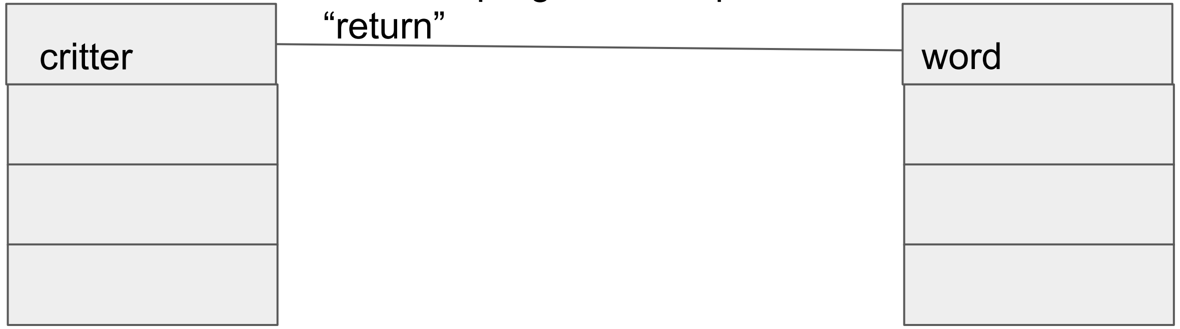
```
#  First the function definition
def reverse_word (word)
# now the code. DON’T FORGET TO INDENT!!
        i = 0
        reversed_word = ‘’
        while i < len(word):
                reversed_word = reversed_word + word [i]
#       return(reversed_word)
…       word = reversed_word
```

```
# 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?

These are NOT the same locations in memory. The value is copied over when the function call is made. Nothing is copied back to the main program, except what’s in “return”



Memory for main program

Memory for reverse_word

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."

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.”