CptS 111 — Lab #5

Practice with Functions

Learning Objectives:

- Be able to define void and non-void functions
- Be able to call functions
- Be able to use keyword arguments and default parameters
- Practice using docstrings

Prerequisites:

- Able to compose programs using good programming habits, that have headers, and that make use of comments, variables, basic arithmetic, data types, and basic I/O statements
- Understand the basics of functions

Task 1: Using the IDLE Shell window to practice functions

There are a number of reasons functions are used in programming. We can think of functions as a way of allowing us to write one part of a program while ignoring everything else. It's like, for example, remodeling a kitchen. We can have someone build the cabinets, someone else construct the countertops, someone else do the flooring, and so on. Of course, everything has to be put together in a certain order, and everyone has to know certain parameters, e.g., sizes and colors, to make sure everything fits together. The same is true for programming functions.

Start by opening an IDLE Shell window, and we'll practice a few basics. Please enter everything in boldface font.

```
>>> # Let's start with the simplest function: a void function with
>>> # no parameters.
>>> def greet_person():                # Void function, no params
        print('Hello.  How are you?')
        return
>>>
>>> # This is called a void function because we aren't returning
>>> # anything from the function, i.e., the return statement is empty
>>> # (void = empty).  It has no parameters because there's nothing
>>> # in the ()'s.
>>>
>>> # When we define a function, we have to use a function call to run
>>> # it.  Here's the function call for the function above.
>>> greet_person()                     # Function call
Hello.  How are you?
>>>
>>> # Pretty simple, eh?  Now let's consider a void function with one
>>> # parameter.  We'll modify the function we just wrote.
```

```
>>> def greet_student(name):          # Void function w/ param
        print(f'Hello, {name}.  How are you?')
        return
>>>
>>> # The difference between the two functions is that we've used a
>>> # parameter, i.e., something in the ()'s.  A parameter in the
>>> # function definition corresponds to an argument in the function
>>> # call.
>>> student = 'Mohammed'
>>> greet_student(student)            # Function call w/ arg
Hello, Mohammed.  How are you?
>>>
>>> # Function arguments and parameters don't have to have the same
>>> # name, but they can.  Sometimes we can pass an argument directly.
>>> greet_student('Maria')
Hello, Maria.  How are you?
>>>
>>> # Both examples above are void functions.  Next let's write a
>>> # non-void function without parameters.
>>> def get_name():                   # Non-void function, no params
        name = input('Enter your first name: ')
        return name
>>>
>>> # In this example THERE'S SOMETHING NEXT TO THE RETURN STATEMENT,
>>> # I.E., IT ISN'T EMPTY.  BECAUSE WE'RE RETURNING SOMETHING FROM
>>> # THE FUNCTION, WE HAVE TO ASSIGN AN LVALUE TO THE FUNCTION CALL.
>>> name = get_name()                 # Function call w/ lvalue
Enter your first name: Sam
>>> name    # What was returned by the function!
'Sam'
>>>
>>> # NOTE THAT THE LVALUE CAN HAVE ANY NAME.
>>> student = get_name()
Enter your first name: Ann
>>> student
'Ann'
>>> # Let's now move on to a non-void function with parameters.
>>> def calc_bmi(wt, ht):
        return 703 * wt / (ht ** 2)
>>>
>>> # What have we done here?  Well, as you can see, calc_bmi() has
>>> # two parameters, and we can actually calculate the user's BMI and
>>> # return it all in one line.  However, my personal preference is
>>> # to use one extra line for clarity.
>>> def calc_bmi(wt, ht):             # Non-void function w/ params
        bmi = 703 * wt / (ht ** 2)
        return bmi
>>>
>>> # This allows us to see what's returned with a quick glance.  We
```

```
>>> # must include two arguments in the function call, e.g.,
>>> wt = float(input('Enter weight [pounds]: '))
Enter weight [pounds]: 124
>>> ht = float(input('Enter height [inches]: '))
Enter height [inches]: 65
>>> bmi = calc_bmi(wt, ht)            # Function call w/ args
>>> print(f'Your bmi is {bmi:.1f}.')
Your bmi is 20.6.
>>>
>>> # Again, notice that because this is a non-void function, we have
>>> # to assign an lvalue to the function call.
```

So those are the four basic function configurations. Anything else we do will simply be a variation of one of these four. When you've completed this task, show your work to your TA for credit.

Task 2: More practice with functions

We'll practice with functions a bit more in the IDLE Shell window.

```
>>> # First consider another way of commenting Python programs using
>>> # docstrings.  Docstrings are typically used to describe
>>> # functions.  For example,

>>> def calc_bmi(wt, ht):
        '''
        Non-void function to calculate a user's BMI given their weight
        and height.  The BMI is returned to the calling program.
        '''
        bmi = 703 * wt / (ht ** 2)
        return bmi
>>>
>>> # We use a triple set of single or double quotes, which can be on
>>> # the same line, to create a docstring.  A docstring is written in
>>> # the first line of the function body under the function header.
>>> # A docstring is used by Python's help() function as shown below.
>>> help(calc_bmi)
Help on function calc_bmi in module __main__:

calc_bmi(wt, ht)
    Non-void function to calculate a user's BMI given their weight
    and height.  The BMI is returned to the calling program.
>>>
>>> # When we use the help() function with the name of our function,
>>> # Python displays the docstring we included in our function
>>> # definition.  The help() function can be used with both
>>> # user-defined functions and builtin functions such as print().
>>> # Let's try another one.
```

```
>>> def get_input():
        '''
        Void function to prompt user for their weight and height.
        These are returned to the calling function.
        '''
        wt = float(input('Enter weight [pounds]: '))
        ht = float(input('Enter height [inches]: '))
        return wt, ht
>>> help(get_input)
Help on function get_input in module __main__:

get_input()
    Void function to prompt user for their weight and height.
    These are returned to the calling function.
>>>
>>> # Pretty neat!  Notice that the get_input() function has no
>>> # parameters, but it returns two values.  It actually returns a
>>> # single tuple, but we can treat it like two values.  Then we also
>>> # need to use two lvalues with our function call.
>>> pounds, inches = get_input()
Enter weight [pounds]: 124
Enter height [inches]: 65
>>> print(f'The user weighs {pounds} pounds and is {inches} inches tall.')
The user weighs 124.0 pounds and is 65.0 inches tall.
>>>
>>> # In the function call above, wt was assigned to the lvalue pounds
>>> # and ht was assigned to the lvalue inches.
>>>
>>> # Next let's consider the use of keywords and default arguments.
>>> # Recall that we can use the keyword sep in the builtin function
>>> # print().  Try the following.
>>> print('Hello', 'World!')
Hello World!
>>> print('Hello', 'World!', sep='')
HelloWorld!
>>>
>>> # In the first print() command, we use the default value, a space,
>>> # for sep.  In the second, we change its default value to no
>>> # space.  We can also write our own functions with keyword
>>> # arguments and default values.  Consider the following
>>> def cheer(team='Cougs'):
        '''
        This function allows the user to override 'Cougs' by
        passing it another team name.  It prints a cheer for the
        team.
        '''
        print(f'Go, {team}!!!')
        return
>>>
```

```
>>> # Let's call this function two different ways.
>>> cheer()
Go, Cougs!!!
>>> cheer('Mariners')
Go, Mariners!!!
>>>
>>> # It's as simple as that!
```

After completing this task, show it to your TA to get credit.

Task 3: From forward to reverse

For this task, you're going to write a program in an IDLE Editor window that prints a four-digit, positive integer in reverse order. First, write a void function called `reverse_digits()` with a single integer parameter, which is the four-digit, positive integer passed to the function. Your function will print this integer in reverse order (see examples below) on one line with no spaces between digits. To do this, you need to use floor division and the modulo function, and you need to change the default value of **`end`**. Also, don't forget to include a program header and a docstring in the function. After you've written the function, add code that prompts the user for a four-digit, positive integer and then calls the **`reverse_digits()`** function with the integer as its argument. Remember that a call to a void function doesn't include an lvalue. Examples are given below.

```
Enter a four-digit, positive integer: 1234
4321
```

If the input has trailing zeros, the reversed number will look a bit odd, but that's fine.

```
Enter a four-digit integer: 1200
0021
```

After your program is running correctly, show it to your TA to get credit. Be sure to save your program in a Lab5 folder as `lab5_t3.py`.

Task 4: The doomsday algorithm

The doomsday algorithm https://en.wikipedia.org/wiki/Doomsday_rule is used to tell what day of the week it is for any given year using the doomsday for that year. A year's doomsday is given by the equation:

$$doomsday = (((yr \;//\; 12) + (yr \;\%\; 12) + (yr \;\%\; 12) \;//\; 4) \;\%\; 7 + anchor) \;\%\; 7$$

where yr is the last two digits of the year and the anchor day is an integer between 0 and 6 with 0 corresponding to Sunday and 6 to Saturday. In this lab, you'll return the doomsday as a number representing the day of the week, but in the next lab, you'll return the doomsday as an actual day of the week.

For this task, write a program with two functions as described below in the correct order in which they should appear. Don't forget to include a header in your program and docstrings in your functions.

- **`det_anchor()`**: A non-void function with one parameter, the first two digits of the input year, that returns the anchor day determined using an `if-elif-else` construct. If the digits are `19`, the function should return `3`; if they're `20`, it should return `2`; if they're something else, it should print an error message. Note that you can return a value to the calling program after a condition has been tested, i.e., right after the test statement.
- **`calc_doom()`**: A non-void function with two parameters, the year and the anchor day (yr and $anchor$ in the equation above), that calculates the doomsday and returns the result to the calling program.

The remainder of your program should do the following:

- Prompt the user for a year, a 4-digit integer (see examples below).
- Separate the input year into two pieces with the first two digits used to determine the anchor and the last two digits used as the year (yr) in the doomsday equation. `divmod(year, 100)` works well for this.
- Call `det_anchor()` using the appropriate part of the input year as the argument, and don't forget to assign the result to an lvalue.
- Call `calc_doom()` with the appropriate part of the input year as one argument and the result obtained from `det_anchor()` as the second, and don't forget to assign the result to an lvalue.
- Print the result (an integer) as demonstrated in the examples below.

Here are some examples

```
Enter a year [4-digit integer]: 1966
The doomsday for 1966 is 1.

Enter a year [4-digit integer]: 2023
The doomsday for 2023 is 2.
```

In January, the doomsday falls on Jan. 3rd (Jan. 4th in leap years) and in February and March it falls on the last day of February. It then falls on the day of the month for even months (e.g., on the 4th of April and 12th of December), and for odd months we use the mnemonic, 9-to-5 at the 7-11 (a commercial from the 20th century). Thus, for May the doomsday falls on 5/9, for July on 7/11, for September on 9/5, and for November on 11/7. What is next year's doomsday?

Save your program in Lab5 as `lab5_t4.py`. Run the program for your TA to get credit.

I hope you feel good about functions now (or at least better); we'll practice them again in our next lab.