Lecture 14

Function Review and Function Parameters

1 Germinal, Year CCXXX

***Song of the day:* *Feel My Rhythm* by Red Velvet (2022).**

Part 1: *The Super Entertaining and Basic (S.E.B) Calculator.*

Introduction

Our goal is to create a calculator that will function as follows:

```
Welcome to the S.E.B Calculator!

Would you like to calculate (y), or shut down (n)? y

Would you like to add (+), subtract (-), multiply (*), or divide (/)? Enter DONE
ERROR: Please enter a valid operator (i.e. +, -, *, /).
Would you like to add (+), subtract (-), multiply (*), or divide (/)? Enter DONE
ERROR: Please enter a valid operator (i.e. +, -, *, /).
Would you like to add (+), subtract (-), multiply (*), or divide (/)? Enter DONE
Enter a number: 42
Enter another number: 42
42.0 + 42.0 = 84.0
Would you like to add (+), subtract (-), multiply (*), or divide (/)? Enter DONE
Enter a number: 42
Enter another number: 0.42
42.0 / 0.42 = 100.0
Would you like to add (+), subtract (-), multiply (*), or divide (/)? Enter DONE
Enter a number: 5
Enter another number: 0.77
5.0 * 0.77 = 3.85
Would you like to add (+), subtract (-), multiply (*), or divide (/)? Enter DONE

Would you like to continue operation? [y/n] y
Would you like to add (+), subtract (-), multiply (*), or divide (/)? Enter DONE
Enter a number: 4
Enter another number: 5
4.0 + 5.0 = 9.0
Would you like to add (+), subtract (-), multiply (*), or divide (/)? Enter DONE
Would you like to continue operation? [y/n] n
```

Note: Your output need not look exactly like mine.

That is, a program that will:

1. Ask the user if they want to perform arithmetic operations, or shut down. Here, and only here, you can assume that the user will enter either `'y'` for "yes" and `'n'` for "no".
2. If the user chooses "yes", the calculator will ask them to enter the operator that they want to use between **two** numerical operands. Here, the user is **limited to** `'+'` **,** `'-'` **,** `'*'` **,** `'/'` **, and** `"DONE"` **(to stop arithmetic operations).** If the user enters anything else, the program should ask them for input again until they enter a valid operator.
3. Once a valid operator has been entered, the program will prompt the user to enter two numerical values (here, you can assume that the user will always enter real, valid numerical values). The program will then print the result of such operation.
4. The program will then ask the user if they want to perform another arithmetic operation (i.e. step 2).
5. If the user chooses not to continue (`"DONE"`), the program will ask the user if they want to continue or shut down (i.e. step 1).

As you can see, we have a couple of program loops going on here, and it super complicated, but let's break each of these steps down into small functions to help us organize better.

Step 1**:** `add()` **,** `subtract()` **,** `multiply()` **, and** `divide()`

Define four simple functions, `add()` , `subtract()` , `multiply()` , and `divide()` . Each of these functions will do the following:

1. Ask the user for two numerical inputs.
2. Perform the relevant arithmetic operation.
3. Print the result.

That's it. Make sure to get these four functions down before moving on!

Step 2**:** `do_arithmetic()`

Define a function called `do_arithmetic()`. This function will perform the instructions from step 2 above:

1. The calculator will ask them to enter the operator that they want to use between **two** numerical operands.
2. If the user enters anything other than a valid operator, the program should ask them for input again until they enter a valid operator.
3. Once they do, your program must **perform a function call** to either `add()`, `subtract()`, `multiply()`, and `divide()` depending on what operator was chosen. If `"DONE"` was chosen, this marks the end of your function.

Note: Make sure to actually use `add()`, `subtract()`, `multiply()`, and `divide()` in this step! We're trying to get used to breaking programs down into functions. Since it's our first time doing this, don't worry if you struggle with it—just follow the instructions and give it your best shot.

Step 3: Our driver function, `main()`

The function that will run this whole program will be called, by convention, `main()`. This function will perform the instructions from step 1 in our introduction:

1. Ask the user if they want to perform arithmetic operations, or shut down. Here, and only here, you can assume that the user will enter either `'y'` for "yes" and `'n'` for "no".
2. If they say "yes", run `do_arithmetic()`. If they say "no". Your `main()` function can end.

Step 4: Run the program.

At the end, make a call to your `main()` function to run the whole thing! You may use **this file**.

Solution

Step 1 involved the definition of three near-identical functions. The syntax of definiting a function, (`add()` for example) is as follows:

```
def add():
    # Function body
```

The instructions called for us to ask the user for the numerical values, calculate their sum, and then display the result. Let's put that inside of our function body:

```
def add():
    # Asking user for input
    number_one = float(input("Enter a number: "))
    number_two = float(input("Enter another number: "))

    # Calculating the sum
    addition = number_one + number_two

    # Printing result
    print("{} + {} = {}".format(number_one, number_two, addition))
```

The other three functions `add()` , `subtract()` , `multiply()` , and `divide()` will look almost identical, save for the operation that they are actually carrying out:

```
def subtract():
    number_one = float(input("Enter a number: "))
    number_two = float(input("Enter another number: "))
    difference = number_one - number_two
    print("{} - {} = {}".format(number_one, number_two, difference))


def multiply():
    number_one = float(input("Enter a number: "))
    number_two = float(input("Enter another number: "))
    product = number_one * number_two
    print("{} * {} = {}".format(number_one, number_two, product))


def divide():
    number_one = float(input("Enter a number: "))
    number_two = float(input("Enter another number: "))
    ratio = number_one / number_two
    print("{} / {} = {}".format(number_one, number_two, ratio))
```

Awesome. So now we have the four functions that our calculator will use during the operation of our program. You can text any of these functions to make sure that they work well by **calling it** anywhere below its definition. For example:

```
def divide():
    number_one = float(input("Enter a number: "))
    number_two = float(input("Enter another number: "))
    ratio = number_one / number_two
    print("{} / {} = {}".format(number_one, number_two, ratio))
```

```
divide()  # a call to the divide() function
```

Possible output using `7` and `42` as input:

```
Enter a number: 7
Enter another number: 42
7.0 / 42.0 = 0.16666666666666666
```

Next, we have to define our `do_arithmetic()` function:

```
def do_arithmetic():
    # function body here
```

We know that `do_arithmetic()` will continue to ask the user to enter any of the possible operators (`'+'` , `'-'` , `'*'` , `'/'`) **until** they enter the end code `"DONE"` . Any time we see this kind of behaviour, we know that we will need a `while` -loop.

We learned two ways of "ending" a `while` -loop. Here, I will be using the "flagging" method. Our flag, in this case, will be a variable called `is_operation_not_done` , which will start as `True` :

```
is_operation_not_done = True

while is_operation_not_done:
    # ...
```

This makes sense if you read it in English:

> **While** the operation is not done (i.e. `True`), continue running our loop.

Whenever the user enters the end code `"DONE"` , we will switch the "flag" to equal `False` , and this will end the loop.

Our first step inside our loop is to ask for the user's operator of choice:

```
def do_arithmetic():
    is_operation_not_done = True

    while is_operation_not_done:
        user_operator = input("Would you like to add (+), subtract (-), multiply 
                              "end. ")
```

Here's a cool trick with flagging. *If* the user does end up entering `"DONE"`, we don't really care if the rest of the code inside the `while`-loop executes. We would like to flip the `is_operation_not_done` to `False`, and restart the loop so that the `while`-loop can see that we are finished.

We will use the `continue` keyword for this:

```
def do_arithmetic():
    is_operation_not_done = True

    while is_operation_not_done:
        user_operator = input("Would you like to add (+), subtract (-), multiply 
                              "end. ")

        if user_operator == "DONE":
            is_operation_not_done = False
            continue
```

This is basically telling Python the following:

> If the user enters `"DONE"` as input, the operation is done (i.e. `is_operation_not_done` is `False`). Skip anything else that may come in the loop (`continue`) and restart the loop.

Note: If the use of `continue` confuses you, don't worry. You can continue using our old method of ending `while`-loops. I included a simple example **here** of `continue` inside a `for`-loop for your benefit.

Alright so, provided that the user did not enter `"DONE"`, we can check if the user entered a valid operator. If they didn't, instead of making a call to one of our arithmetic functions, we can just print an error message:

```
ef do_arithmetic():
   is_operation_not_done = True

   while is_operation_not_done:
       user_operator = input("Would you like to add (+), subtract (-), multiply (*)
                             "end. ")

       if user_operator == "DONE":
           is_operation_not_done = False
           continue
```

```
        if user_operator == '+':
            add()
        elif user_operator == '-':
            subtract()
        elif user_operator == '*':
            multiply()
        elif user_operator == '/':
            divide()
        else:
            print("ERROR: Please enter a valid operator (i.e. +, -, *, /).")
```

The loop will repeat these instructions over and over until the user enters `"DONE"`. And that is all you need for `do_arithmetic()`.

Finally, we'll move onto our `main()` function. The name "main" is a convention typically used in computer science, and we usually reserve it for the function that starts the entire app. It's also commonly referred to as the "driver" function, since it drives the entire program forward.

Our instructions tell us to simply ask the user to say whether they want to start the arithmetic program to run or not. It will continue to do so, until the user enters the "shutdown" code (i.e. `'n'`). This is a simple `while`-loop like the ones we have done before:

```
def main():
    print("Welcome to the S.E.B Calculator!")
    user_choice = input("Would you like to calculate (y), or shut down (n)? ")

    while user_choice != 'n':
        do_arithmetic()
        user_choice = input("Would you like to continue operation? [y/n] ")
```

The last step I have listed is to run the entire program. By this, I simply mean to make a call to your driver function. In our case, this is our `main()`.

```
main()  # this is typically the last line in our code
```

You can find the full solution **here** (in the `solutions` sub-folder).

Part 2: *Function Parameters*

Let's talk about the anatomy of a familiar function call:

```
name = input("What is your name?")
```

Even though this is just a single line of code, there are actually three things going on here:

1. A **function** call to the `input()` function.
2. The **assignment** of whatever value `input()` returns to the variable `name`.
3. The **passing** of the string `"What is your name?"` as part of the `input()` function call.

That's actually a lot, if we consider the functions we've written so far. Take our `add()` function, for example. When we make a function call, the only thing that is happening is step 1. There is no assignment to anything, nor is there any passing of data into the `add()` function as part of its invocation.

```
add()
```

We'll leave assignment for next lecture. What I would like to do instead is to be able to do things like what certain methods in the `math` module does. Consider the `pow()` method:

```
import math

base = float(input("Enter a number to use as the base: "))
power = float(input("Enter a number to use as the power: "))

result = math.pow(base, power)

print(result)
```

The `math.pow()` method accepts two **arguments** in its parentheses during invocation: the base, and the power of the exponentiation.

> **Arguments**: The values placed between the parentheses of a function *during a function call*.

If we were to write a `pow()` function similar to this one with our current knowledge, we would have to use `input()` *inside* its definition. Above, these values exist *outside* the `math.pow()` definition, and are completely independent from it.

So, how do we make it so that our `add()` function accepts arguments from the outside, to be used inside the function definition? Very simply:

```python
def add(number_one, number_two):
    addition = number_one + number_two
    print(addition)
```

Two things changed:

- Inside the set of parentheses of the function signature, we wrote the names of the variables (`number_one` and `number_two`) where we will store the values that we anticipate to receive from the outside.
- Since we're using these outside values, stored inside `number_one` and `number_two` , we no longer need to use `input()` to ask the user to enter them. Thus, we delete those two lines.

Now, if we wanted to use our `add()` function, we would do so the same way we use `math.pow()` :

```python
user_input_one = float(input("Enter a number to add: "))
user_input_two = float(input("Enter another number to add: "))

add(user_input_one, user_input_two)
```

What was the point of this? It will become clearer as the semester goes along, but in general, you want each of your functions to do a **single** thing. `print()` , for instance, only displays the arguments that you pass through its parentheses. It doesn't do **anything** else, like calculate the length of your argument, or its ASCII value sum, etc. It is a function defined to do one thing and one thing only: to print.

Similarly, `add()` should do one thing and one thing only: add.

Note: Right now, `add()` is technically doing two things: adding and printing. We will take care of that on Wednesday.

Something you may have noticed is that the operands passed into `add()` as arguments have different variable names than they do inside the definition of `add()` :

```python
# Function definition
def add(number_one, number_two):
    addition = number_one + number_two
    print(addition)
```

```
# Function invocation/call
user_input_one = float(input("Enter a number to add: "))
user_input_two = float(input("Enter another number to add: "))

add(user_input_one, user_input_two)
```

This, however, still works perfectly fine if we run it:

```
Enter a number to add: 42
Enter another number to add: 24
66.0
```

This is *by far* the part of functions that confuses students the most. If the first number is called `number_one` inside the function, why can I pass a variable called `user_input_one` into it?

This is because `number_one` and `number_two` are what are called **parameters**, as opposed to arguments (which we defined earlier).

> **Parameters**: The values placed between the parentheses of, and then used in, a function *definition*.

So, in the line:

```
add(user_input_one, user_input_two)
```

This is what is happening:

1. The values stored inside the variables `user_input_one` (say, `42`) and `user_input_two` (say, `24`) are passed in as ***arguments*** to the `add()` function during invocation.
2. Since `user_input_one` is passed first, the value stored inside it is then assigned to the first ***parameter*** in the `add()` function definition. In this case, the first parameter is `number_one`. Thus, `number_one` also has the value of `42`.
3. Since `user_input_two` is passed second, the value stored inside it is then assigned to the second ***parameter*** in the `add()` function definition. In this case, the second parameter is `number_two`. Thus, `number_two` now also has the value of `24`.
4. `add()` has thus "received" values from outside its own definition, and can use them in its operations.

What makes this more confusing is that you *can* name your arguments and parameters the same:

```
# Function definition
def add(number_one, number_two):
    addition = number_one + number_two
    print(addition)


# Function invocation/call
number_one = float(input("Enter a number to add: "))
number_two = float(input("Enter another number to add: "))

add(number_one, number_two)
```

This is possible and works totally fine, so why not do this? Imagine you work for **NASA** as an astrophysicist, and you need a formula to calculate the volume of a planet. You might write a function like this:

```
import math

SPHERE_VOL_CONSTANT = math.pi * 4 / 3


def print_volume_of_planet(planet_radius):
    planet_volume = SPHERE_VOL_CONSTANT * planet_radius ** 3
    print("The volume of this planet is {}".format(planet_volume))


planet_radius = 6378.0  # Earth; in km; approx.
print_volume_of_planet(planet_radius)
```

Output:

```
The volume of this planet is 1086781292542.8892
```

Cool, but in space, there are all sorts of object that are spherical. Let's say we wanted a function that will calculate the volume of a star. You might do this:

```
import math

SPHERE_VOL_CONSTANT = math.pi * 4 / 3


def print_volume_of_star(star_radius):
    star_volume = SPHERE_VOL_CONSTANT * star_radius ** 3
    print("The volume of this star is {}".format(star_volume))
```

```
star_radius = 37.5  # Polaris; in in Solar radii; approx.
print_volume_of_star(star_radius)
```

The inefficiency here is probably pretty clear: both functions do *exactly the same thing, in the exact same way*. The question here is: why have two functions that do the same thing for specific situations, when we can have one single function that will work for both?

The **idea way** to write a function like this would be:

```
import math

SPHERE_VOL_CONSTANT = math.pi * 4 / 3


def print_volume_of_sphere(radius):
    volume = SPHERE_VOL_CONSTANT * radius ** 3
    print("The volume of this spherical object is {}.".format(volume))


earth_radius = 6378  # in km; approx.
print_volume_of_sphere(earth_radius)

polaris_radius = 37.5  # in in Solar radii; approx.
print_volume_of_sphere(polaris_radius)
```

This makes `print_volume_of_sphere()` an extremely reusable function that can be used with literally any sphere, astral body or not.