Chapter 4

Functions

Learning Objectives

- Functions
- Scope
- For vs. While

4.1 Functions

"Fruitful"

The book makes the distinction between functions that return something (*fruitful*) and those that do not (*non-fruitful*). The distinction and terminology makes sense, but the *non-fruitful* functions should still *do something*, right? When the effect isn't shared back, we call it a *side-effect*. Examples of *side-effects* are: `print()`ing something to the screen, having a turtle draw something and manipulating a file.

Python typically assumes "fruitfull" operation while running a program. Thus, whenever executing something it stays quiet unless you explicit tell it to inform you (`print()`). This in contrast to the Python console which will proudly tell you whenever it has done something.

(extra) Recursion

Within a function you can call another function. The function could also call itself. This is potentially very powerful, but not without risk. It is easy to run out of *system resources*, like memory, when using recursion.

Unless you really know what you are doing, you should use `while` (chapter 8). When you are in doubt, ask yourself: "which one is more readable?" (it's usually the while which is why you should prefer it).

4.2 Scope

Scope is the term used to described where something (like a variable or function) is available. When you declare a variable, i.e. use it for the first time, a bit of memory is reserved for it. This memory will remain reserved until you *un-indent*. Make sure to "create" the variable in such a way that it is effective.

See the code in Listing 4 for some examples. The variable `total` on lines 2, 13 and 25 are totally different. Take the one on line 2, it stops existing after line 9. We need to initialize it on line 2 (and not for example on line 7) so we have access to it on line 9. We could have named the variable on line 25 anything. It just happens that "total" seems like an appropriate name for both locations.

```
def get_five_numbers():
    total = 0
    for i in range(5):
        number = input("Give a number ")
        number = int(number)

        total = total + number

    return total

def get_user_input():
    # intial values to get us started
    total = 0
    print("0 to stop")
    number = int(input("Give a number "))
    while number != 0:
        # process the number
        total =  total + number

        # get the next number
        number = int(input("Give a number "))

    return total

total = get_five_numbers()
print(total)

total = get_user_input()
print(total)
```

Listing 4: `for` vs `while`

4.3 for vs. while

How to choose between the three ways to loop over a piece of code. As mentioned before: *you should only use recursion if you really understand it!*. In general, think of recursion as a *divide and conquer* technique more

than a way to "iterate".

This leaves `for` and `while`. Use `for` when you know the (upper bound of the) number of iterations. Using `for` allows python to optimally use the computer to execute your program.

This means `while` should be used when you don't know the number of iterations needed. For example: keep accepting user input until they give a special value. There is no way to know how long this will take, so use a `while`.

When using `while` you risk an infinite-loop. We'll not go to deep into "proving" your code works. Think about it, try it, and if you ever get stuck: **CTRL-C** (it doesn't mean copy on the command-line).

4.4 Homework

Reading

- Chapter 4: Functions (read up-to an exercise section, see below, and make the exercises first)
- *after you have finished chapter 4,* and if you are intrigued by the concept of recursion, consider looking at Appendix C: Recursion; C1.

Assignments

1. 4.9
2. 4.17
3. rewrite you Mandala program from last week using functions, can you make your code more readable?