Chapter 5: Python More Fun with Functions!

Contents

Time required: 120 minutes

DRY

Don't **R**epeat **Y**ourself (**DRY**) is a principle of software engineering aimed at reducing repetition of software patterns. It you are repeating any code, there is probably a better solution.

Type Hinting

Python is dynamically typed. This means that the type of value that can go into a variable can be changed. We can add typing to Python to be more specific about the types of data that goes into and out of our functions.

Let's take an example of a function which receives two arguments and returns a value indicating their sum:

```
def two_sum(a, b):
    return a + b
```

By looking at this code, one cannot safely and without doubt indicate the type of the arguments for function two_sum. It works both when supplied with int values:

```
print(two_sum(2, 1))  # result: 3
```

and with strings:

```
print(two_sum("a", "b"))  # result: "ab"
```

and with other values, such as lists, tuples et cetera.

Due to this dynamic nature of python types, where many are applicable for a given operation, any type checker would not be able to reasonably assert whether a call for this function should be allowed or not.

To assist our type checker we can now provide type for it in the Function definition indicating the type that we allow.

To indicate that we only want to allow int types we can change our function definition to look like:

```
def two_sum(a: int, b: int):
    return a + b
```

Annotations follow the argument name and are separated by a : character.

Similarly, to indicate only str types are allowed, we'd change our function to specify it:

```
def two_sum(a: str, b: str):
    return a + b
```

Apart from specifying the type of the arguments, one could also indicate the return value of a function call. This is done by adding the -> character followed by the type after the closing parenthesis in the argument list but before the : at the end of the function declaration:

```
def two_sum(a: int, b: int) -> int:
    return a + b
```

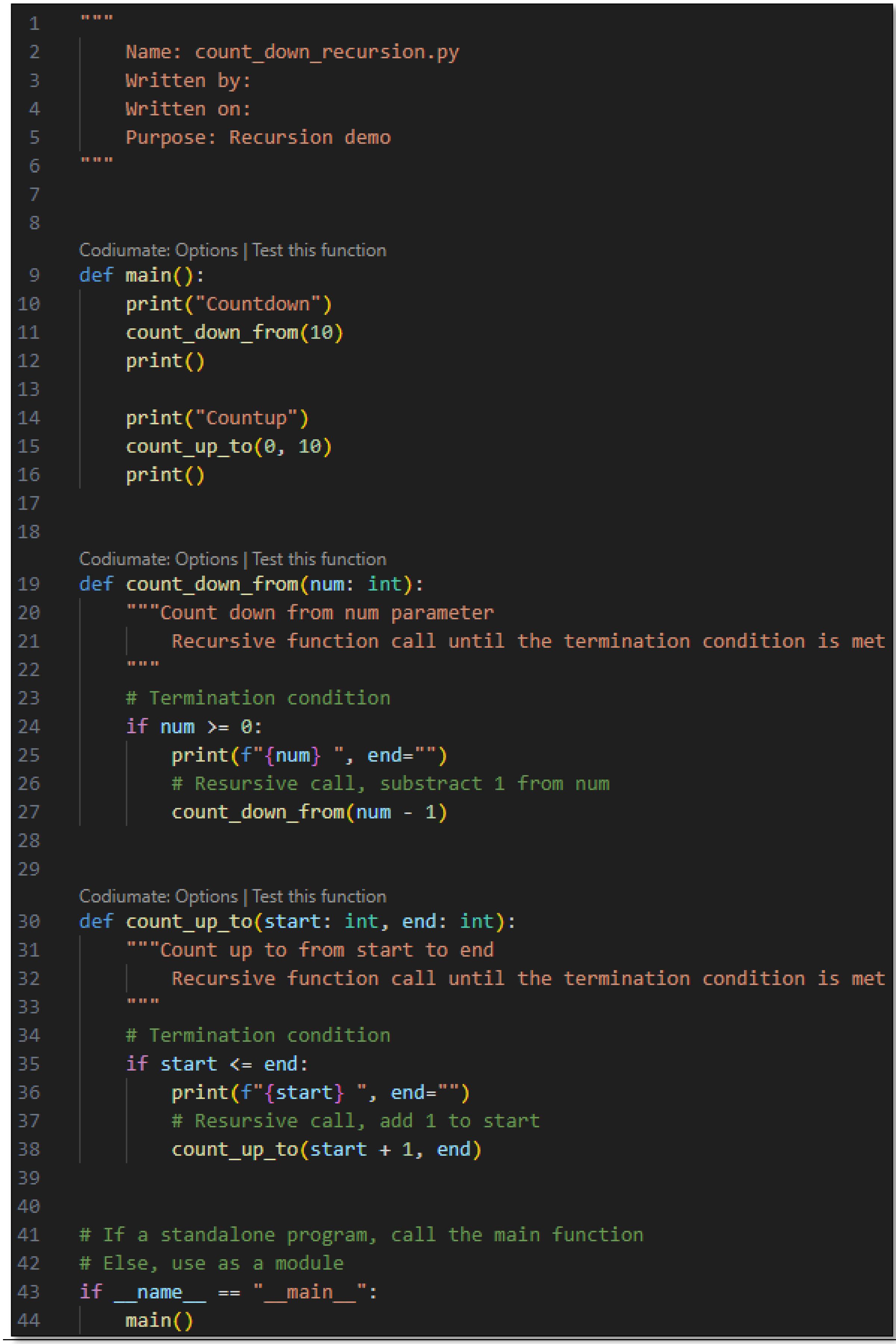
We've indicated that the return value when calling two_sum should be of type int. Similarly, we can define appropriate values for str, float, list, set and others.

Tutorial 5.1: Recursion

A function can be called from another function. Recursion is when a function calls itself to accomplish a task.

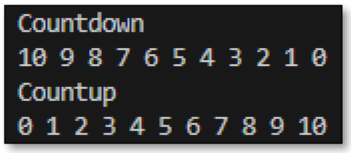
Recursion is a method of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition satisfies the condition of recursion, we call this function a recursive function.

Termination condition: A recursive function has to fulfil an important condition to be used in a program: it has to terminate. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case.



```python
"""
    Name: count_down_recursion.py
    Written by:
    Written on:
    Purpose: Recursion demo
"""


Codiumate: Options | Test this function
def main():
    print("Countdown")
    count_down_from(10)
    print()

    print("Countup")
    count_up_to(0, 10)
    print()


Codiumate: Options | Test this function
def count_down_from(num: int):
    """Count down from num parameter
        Recursive function call until the termination condition is met
    """
    # Termination condition
    if num >= 0:
        print(f"{num} ", end="")
        # Resursive call, substract 1 from num
        count_down_from(num - 1)


Codiumate: Options | Test this function
def count_up_to(start: int, end: int):
    """Count up to from start to end
        Recursive function call until the termination condition is met
    """
    # Termination condition
    if start <= end:
        print(f"{start} ", end="")
        # Resursive call, add 1 to start
        count_up_to(start + 1, end)


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:



```
Countdown
10 9 8 7 6 5 4 3 2 1 0
Countup
0 1 2 3 4 5 6 7 8 9 10
```

Tutorial 5.2: Input Function in utils.py Module

NOTE: This bonus tutorial is inspired by a student submission.

We are going to add to our **utils.py** module that will contain various functions that we will use in future programs.

The following function gets a numeric input. It uses try except to catch any exceptions. If the input is not the correct data type, the function asks the user again. It passes a prompt parameter to identify what the input function is looking for.

This function can be used in any program that needs int or float numeric input and will catch any exceptions. Change the int to a float in the module.

You don’t have to understand how the function works to use it. You can copy and paste it or use it as a module. You just need to know 2 things about how to use the function.

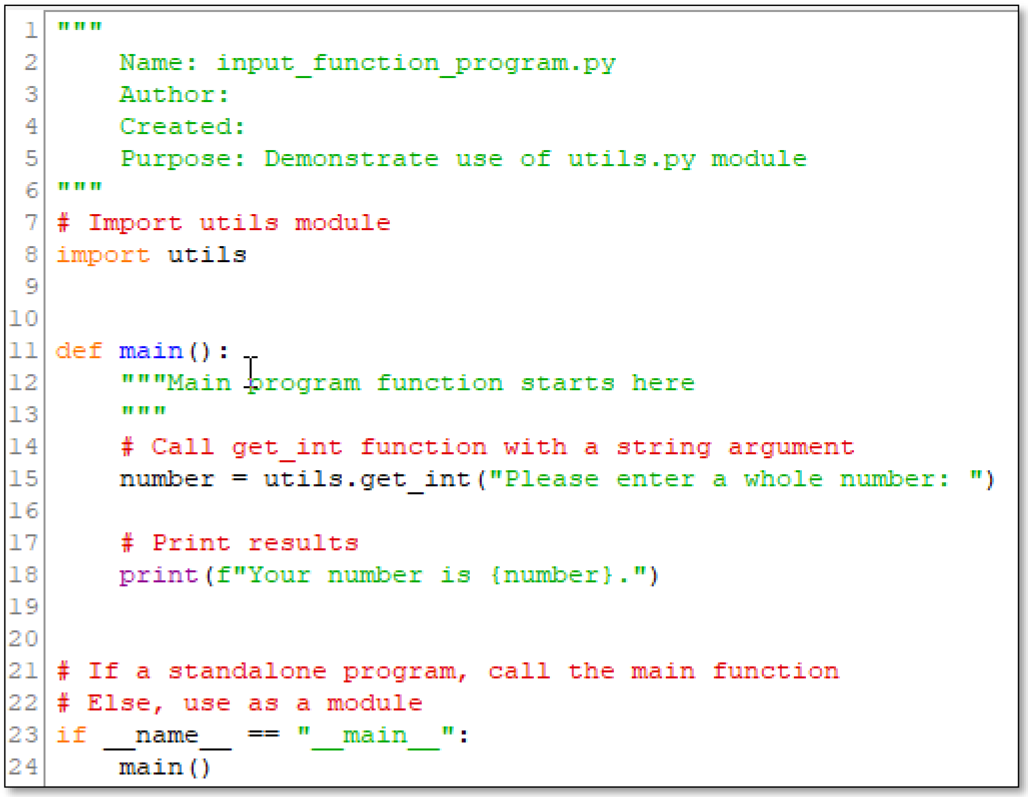
1. The string parameter **what** is the input prompt.
2. The function returns an integer.

Requirements

1. Open **utils.py**
2. Add the following code above the title function.

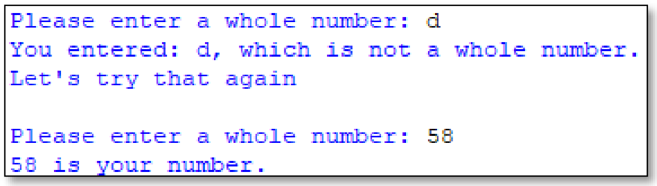
```python
"""
    Name: utils.py
    Author:
    Created:
    Purpose: A utilty module with commonly used functions
"""


def get_int(prompt):
    """Get an integer from the user with try catch
       The prompt string parameter is used to ask the user
       for the type of input needed
    """
    # Declare local variable
    num = 0

    # Ask the user for an input based on the prompt: string parameter
    num = input(prompt)

    # If the input is numeric, convert to int and return value
    try:
        return int(num)

    # If the input is not numeric,
    # Inform the user and ask for input again
    except ValueError:
        print(f"You entered: {num}, which is not a whole number.")
        print(f"Let's try that again.\n")

        # Call function from the beginning
        # This is a recursive function call
        return get_int(prompt)
```

Example program using **utils.py** as a module.



```python
"""
    Name: input_function_program.py
    Author:
    Created:
    Purpose: Demonstrate use of utils.py module
"""
# Import utils module
import utils


def main():
    """Main program function starts here
    """
    # Call get_int function with a string argument
    number = utils.get_int("Please enter a whole number: ")

    # Print results
    print(f"Your number is {number}.")


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:



```
Please enter a whole number: d
You entered: d, which is not a whole number.
Let's try that again

Please enter a whole number: 58
58 is your number.
```

Keep the file, **utils.py**, we will continue to use it throughout the class.

Catching Exceptions using Try and Except

Earlier we saw a code segment where we used the input and int functions to read and parse an integer number entered by the user. We also saw how treacherous doing this could be:

```
>>> prompt = "What is the air velocity of an unladen swallow?\n"
>>> speed = input(prompt)
What is the air velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

When we are executing these statements in the Python interpreter, we get a new prompt from the interpreter, think "oops", and move on to our next statement.

If you place this code in a Python script and this error occurs, your script immediately stops in its tracks with a traceback. It does not execute the following statement.

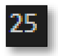
lambda Function

A **lambda** function is an inline, single, or anonymous function which is defined without a name. It is used for simple, short functions, instead of writing out an entire function. Lambda functions are often used in graphical user interface programs.

- A lambda function can take any number of arguments.
- It can only have one expression.
- The result is automatically returned.
- It is defined using the **lambda** keyword.

```
# Filename: lambda_simple.py
# lambda arguments: expression
# argument: a
# expression: a * a
x = lambda a: a * a
# Call and print the lambda function
print(x(5))
```

Example run:

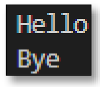


```
25
```

x is passed into the lambda function. "Hello" is return if x > 5, else "bye" is returned.

```
x = lambda x: "hello" if x > 5 else "bye"
print(x(10))
print(x(4))
```

Example run:



```
Hello
Bye
```

Glossary

algorithm A general process for solving a category of problems.

argument A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

body The sequence of statements inside a function definition.

composition Using an expression as part of a larger expression, or a statement as part of a larger statement.

deterministic Pertaining to a program that does the same thing each time it runs, given the same inputs.

dot notation The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

flow of execution The order in which statements are executed during a program run.

fruitful function A function that returns a value.

function A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function call A statement that executes a function. It consists of the function name followed by an argument list.

function definition A statement that creates a new function, specifying its name, parameters, and the statements it executes.

function object A value created by a function definition. The name of the function is a variable that refers to a function object.

header The first line of a function definition.

import statement A statement that reads a module file and creates a module object.

module object A value created by an import statement that provides access to the data and code defined in a module.

parameter A name used inside a function to refer to the value passed as an argument.

pseudorandom Pertaining to a sequence of numbers that appear to be random but are generated by a deterministic program.

return value The result of a function. If a function call is used as an expression, the return value is the value of the expression.

void function A function that does not return a value.

Assignment Submission

1. Attach the program files.
2. Attach screenshots showing the successful operation of the program.
3. Submit in Blackboard.