Module 6: Functions and Modules

February 26, 2021

Last time we talked about the implementation of repetitive behavior (loops). The lecture covered (pre-test) loops with the `while` and `for` statements, and the special behavior of the `break` and `continue` statements. We furthermore discussed the issue of post-test loops in Python.

Today we will discuss functions, modules and packages, have a quick look at the Python's standard library and the Python Package Index, and introduce the principle of recursion and recursive function definitions. When Python programs become complex, it is a good idea to structure the code using functions, modules, and packages. Proper modularization will keep the code base easier to understand and maintain, and is in fact one of the principles of good software development. Another important principle is the reuse of already existing functionality (rather than implementing it again), so familiarity with the standard libraries is important.

Next time we will cover special data structures in Python, which will allow us to work with more powerful data items than just the individual numbers, strings and Booleans that we have used so far.

Functions

We have seen and used functions before (`print` , `input` , and `bin` , for example), now it is time to have a more systematic look at them and to define functions ourselves. Functions are useful whenever a piece of code forms a self-contained component and is (potentially) used repeatedly in the program. It simplifies coding, testing, debugging and maintenance if this code is only defined once (as a function) and just called when it is needed.

Here is an example of code with unnecessary repetitions:

In [1]:
```
player1 = input("Please enter name of player: ")
print(f"Hello {player1}, welcome to the game!")
player2 = input("Please enter name of player: ")
print(f"Hello {player2}, welcome to the game!")
player3 = input("Please enter name of player: ")
print(f"Hello {player3}, welcome to the game!")
```

```
Please enter name of player:  a
Hello a, welcome to the game!
Please enter name of player:  b
Hello b, welcome to the game!
Please enter name of player:  c
Hello c, welcome to the game!
```

Essentially, this code does the same thing three times. This is impractical, for example, if we want to add more players (we have to duplicate a lot of code), or if we want to change the text of the outputs to the players, for instance translate it to Dutch (we would have to do that at all occurrences of the text). So, it might be useful to define functions that ask for a player's name and for displaying a personalized welcome message.

Function definitions in Python have the following basic form:

```
def <function name>(<function parameter(s)>):
    <do something>
    return <value(s)>
```

That is, they define the name of the function, which parameters they take as input arguments at runtime (can be none), what they actually do and the value(s) that they return to the calling function (can be none). Functions have to be defined before they can be used. For example, we can define a function that takes a name as a parameter and welcomes a player accordingly:

In [ ]:
```
def greet_player(name):
    print(f"Hello {name}, welcome to the game!")

player1 = input("Please enter name of player: ")
greet_player(player1)
player2 = input("Please enter name of player: ")
greet_player(player2)
player3 = input("Please enter name of player: ")
greet_player(player3)
```

This already looks better, but we still have redundant code for asking the players to enter their names. We can define another function for that, that has no input parameters, but returns a value to the caller:

In [2]:
```
def ask_name():
    name = input("Please enter name of player: ")
    return name

def greet_player(name):
    print(f"Hello {name}, welcome to the game!")

player1 = ask_name()
greet_player(player1)
player2 = ask_name()
greet_player(player2)
player3 = ask_name()
greet_player(player3)
```

```
Please enter name of player:  a

Hello a, welcome to the game!

Please enter name of player:  b

Hello b, welcome to the game!

Please enter name of player:  c

Hello c, welcome to the game!
```

Note that one of our functions contains a return statement (`return name`), and the other does not. When a function does not explicitly define a return value, the `None` value (representing “nothingness”) is returned by default. Thus, the return statement can be omitted, as implicitly an (empty) return to the caller will still happen at the end of the function. A return can however also be used to return to the caller at any another point of the function if desired.

Alternatively, if we think that we will never need the actions separately, we could simply define one function that carries out the input and the printout:

In [ ]:
```
def ask_name_and_greet_player():
    name = input("Please enter name of player: ")
    print(f"Hello {name}, welcome to the game!")
    return name

player1 = ask_name_and_greet_player()
player2 = ask_name_and_greet_player()
player3 = ask_name_and_greet_player()
```

The functions in the example above are quite short, only one or two lines, and you might wonder if that is actually worth the effort. Well, they were only simple examples to illustrate how functions work. You will soon see in the homework exercises and also in your projects that functions are usually longer and more complex. Below is another example (still small), where the function comprises five lines of code and even a loop:

In [3]:
```
# function for computing the factorial of a number n
def fac(n):
    fac = 1
    while n > 0:
        fac = fac * n
        n = n - 1
    return fac

# reading a number from input
n = int(input("Please enter an integer number: "))

# loop computing the factorials for all number from n to 1
while n > 0:
    print(f"{n}! is {fac(n)}")
    n = n - 1
```

```
Please enter an integer number:  12
```

```
12! is 479001600
11! is 39916800
10! is 3628800
9! is 362880
8! is 40320
7! is 5040
6! is 720
5! is 120
4! is 24
3! is 6
2! is 2
1! is 1
```

This example illustrates another feature of functions: variable names are local to the function. That is, argument names and variables declared inside a function are not visible outside of the function. The area in a program where variables are visible is also called their scope. As a general rule, a variable's scope is the block that it has been declared in, starting from the point where its name was defined. In the example above, this means that the `n` in the `fac` function is another `n` than the one in the program below the function. The function does not change the value of the n in the calling (part of the) program. This behavior makes a lot of sense sense: If you use a function from a library, where you cannot or do not want to look into its implementation, it is good to know that it will not interfere with your own variables. Generally, data should only be passed to and retrieved from functions through their parameters and return values, respectively, as this is the safest way to avoid undesired side effects from e.g. variables that overwrite each other unintentionally.

Note here that the `global` statement can be used to make variables known across scopes. Functions would have access to its value also without it being passed as a parameter, and they could write results to it without returning it explicitly. The use of the `global` statement is however strongly discouraged for the reasons given

above, so we will not discuss it further.

The functions in the examples above have no or only one parameter, but if several inputs are needed, they can simply be defined as comma-separated lists of parameters. For example, a function `add(a,b)` for adding to numbers. When calling the function, the arguments have to passed in the order of the parameters.

Furthermore, functions can have named parameters. They are identified by their name, so their order does not matter when calling the function. They are also given a default value when they are declared, so that when the calling function does not specify them, there is a standard value to work with. Here is an example of a function with one “normal” and two named parameters with default values:

In [ ]:
```
def rectangle_print(letter, columns=3, rows=2):
    for i in range(1,rows+1):
        print(letter*columns)

rectangle_print("a")
rectangle_print("b", 5, 2)
rectangle_print("c", rows=3)
```

The above example illustrates nicely how named parameters can or can not be defined by the caller, depending on what is needed. Note that also the `print` function, which we have used a lot, has named parameters, for example to specify the separator between the string arguments that are passed to print:

In [4]:
```
print("a", "b", "c", sep="\n")
```

```
a
b
c
```

In fact, the arbitrary number of strings that can be passed to the print function to be printed to the screen is an example of yet another kind of parameters, the VarArgs (for variable number of arguments) parameters. Parameters become VarArgs by defining them with a single (*) or double (**) star in front of them. For example, we can use a starred parameter to define a sum function that adds all of its arguments:

In [5]:
```
def sum_up(*numbers):
    s = 0
    for n in numbers:
        s += n
    return s

print(sum_up(1,2,3,4,5))
print(sum_up(1,2,3,4,5,6,7))
```

```
15
28
```

Technically, the arguments for a single-starred parameter form a tuple, and the arguments of a double-starred parameter form a dictionary, hence the latter have to be defined as key-value pairs. What this means will become clear in the next lecture, where we will talk about those data structures in detail.

Modules

Functions make pieces of code in our program file reusable. Modules make it possible to reuse code across different program files. To use the content of another module (file) in a Python program, it has to be imported. We have already seen a module import in the number-guessing game example in the lecture on loops:

```
import random
number = random.randint(1,10)
[...]
```

The `import` statement in the first line imports the `random` module from Python's standard library. Afterwards we can use the functions that it provides, for example the `random()` function that generates a random number within a specified range.

Note that import statements can in principle be placed at any point in the program, as long as they happen before using their functions. This is not considered good coding style, however, as imports scattered all over the program make it difficult to see what has already been imported, and might even lead to redundant and conflicting imports. To avoid such problems, all imports that are used in a .py file should be placed at its beginning.

We can also create modules ourselves. There are different forms in which modules can exist. The simplest is a .py file that contains different functions. (In fact, any .py file we create is a module that can potentially be used by other programs.) Generally, it makes sense to create modules for sets of functions that somehow belong together and where it would make sense to distribute them (to other programmers) as a unit.

For example, we might provide the `ask_name()` and `greet_player()` functions from above in a `player_management.py` module, along with other related functions such as, e.g., `show_score()` or `show_game_over()`:

```
def ask_name():
    name = input("Please enter name of player: ")
    return name

def greet_player(name):
    print(f"Hello {name}, welcome to the game!")

def show_score(name, score):
    print(f"Hello {name}, your current score is {score}.")

def show_game_over(name):
    print(f"GAME OVER! Sorry, {name}...")
```

The module and its functions might then be used by a game as follows:

In [14]:
```
import lib.player_management
import random

player = player_management.ask_name()
player_management.greet_player(player)

score = random.randint(0,10) # (replace by actual code for playing game)

if score > 0:
    player_management.show_score(player,score)
else:
    player_management.show_game_over(player)
```

```
Please enter name of player:  aerew

Hello aerew, welcome to the game!
Hello aerew, your current score is 9.
```

Also when loading modules the Python interpreter acts as usual and will step through the module when importing it. The function definitions in the module are read, and thus the functions are subsequently available

for use. If the module contains other code, outside of function definitions, it will be executed, too.

Note that when creating own modules, it is advisable to not use names that are already taken by modules from the standard library or other popular collections. Name clashes can lead to errors and unpredictable problem behaviour. So, if you observe weird behavior in relation to your modules and imports, check if maybe there is a name clash. There are ways to influence Python's importing mechanism in more detail to circumvent problems, but that gets quite technical and furthermore tends to be unstable, so we will not discuss it in this course.

There are two variations of the `import` statement that you will often see in Python code that is around: The `from … import …` statement can be used to import individual functions from a module, for example:

In [16]:
```
from random import randint
print(randint(1,10))
```

```
2
```

This way, the function is available without using the module name as a prefix. It is not recommended to use this kind of import, however, as it comes with a risk of name clashes and unreliable behavior. The other variation is the `import … as …` statement, which defines an alias for the module name. For example:

In [ ]:
```
import random as rd
print(rd.randint(1,10))
```

This is handy in particular for defining shorter aliases for long module names, but it should be used with care, too, to avoid name clashes.

It is important to realize that all Python .py files are modules. However, not all of them are (just) made for being imported into other modules. Some also have code that should be executed when the module is executed as a script or with `python -m` . Interestingly, a module can discover if it is being executed directly (in Python speech: running in the main scope) or as an import. A pattern that is often used in Python modules is to include such a check, and call a dedicated main function if the method is indeed running in the main scope:

```
if __name__ == "__main__":

  # execute only if running in main scope
  main()
```

This allows to have a module providing functions that are useful to import by other modules, and at the same time, for example, offering a command line interface that is useful when starting the module as a script, but would only be a nuisance would it be run during a standard import.

Packages

Packages are used to organize sets of modules hierarchically. Technically, they are folders that contains modules and a special `__init__.py` file (to indicate that the modules in this folder form a package). We don't go into further detail here, but note that when using modules from collections of functionality such as the Python Standard Library, they are distributed as packages.

The Standard Library and other Sources of Functionality

The Python Standard Library, which is distributed with the Python installation, contains `random` and a large number of other useful packages, such as the `calendar` , `statistics` or `io` libraries. We will see and use some of them in the following lectures. The website https://docs.python.org/3/library/index.html

(https://docs.python.org/3/library/index.html) lists what is contained in the standard library of Python and is a good reference during development.

In addition to the packages in the standard library, there are also many packages available from other sources. The Python Package index at https://pypi.python.org/pypi (https://pypi.python.org/pypi) is the central repository for Python packages, currently ca. 290,000 of them. With the Anaconda platform a large number of popular packages has already been installed on your system, but sometimes packages are not yet installed on your machine, so you have to do that yourself before you can use them. How to do this within Anaconda is described at https://docs.anaconda.com/anaconda/navigator/tutorials/manage-packages/#installing-a-package (https://docs.anaconda.com/anaconda/navigator/tutorials/manage-packages/#installing-a-package). Alternatively, the basic way to do install new packages is via the command line, using the `pip` tool that comes with the Python installation: `pip install <module>`

Recursion

See Recursion.

Just a joke, but it points to the basic idea of recursion: self-reference. You may recall recursive function definitions from mathematics, for instance:

x^n = x * x^{n-1} if n > 0, and 1 if n = 0

That is, the problem is solved by referring to a smaller instance of the same problem, until it is so small that it is trivial. For example, with this definition 35 is calculated as follows:

3^5 = 3 * 3^4 = 3 * 3 * 3^3 = 3 * 3 * 3 * 3^2 = 3 * 3 * 3 * 3 * 3^1 = 3 * 3 * 3 * 3 * 3 * 3^0 = 3 * 3 * 3 * 3 * 3 * 1 = 3 * 3 * 3 * 3 * 3 = 3 * 3 * 3 * 9 = 3 * 3 * 27 = 3 * 81 = 243

Recursions in Python (and other programming languages as well) follow the same principle, but the definition in code looks a bit different:

In [17]:
```
# recursive function to compute x**n
def pow(x,n):
    if n > 0:
        return x * pow(x,n-1)
    else:
        return 1

# main program calling the function
x = int(input("Please enter x: "))
n = int(input("Please enter n: "))
x_to_the_power_of_n = pow(x,n)
print(f"{x} ** {n} = {x_to_the_power_of_n}")
```

```
Please enter x:  3
Please enter n:  6

3 ** 6 = 729
```

Internally, the following calls and returns of pow(x,n) happen at runtime:

```
pow(3,5)
    pow(3,4)
        pow(3,3)
            pow(3,2)
                pow(3,1)
                    pow(3,0)
                    return 1
                return 3
            return 9
        return 27
    return 81
return 243
```

Now let us look at an example for which there is not already an operator in Python. For example, the Fibonacci number for an integer n is defined as:

fib(n) = fib(n-1) + fib(n-2) if n > 1, 1 if n = 1 and 0 if n = 0

In Python:

In [18]:
```
def fib(n):
    if n > 1:
        return fib(n-1) + fib(n-2)
    elif n == 1:
        return 1
    else:
        return 0

n = int(input("Please enter an integer number: "))
print(f"fib({n}) = {fib(n)}")
```

```
Please enter an integer number:  34
```

```
fib(34) = 5702887
```

The call stack here is a bit more complex than above (only shown for fib(4) to keep it short):

```
fib(4)
    fib(3)
        fib(2)
            fib(1)
            return 1
            fib(0)
            return 0
        return 1
        fib(1)
        return 1
    return 2
    fib(2)
        fib(1)
        return 1
        fib(0)
        return 0
    return 1
return 3
```

Here is another, non-mathematical example:

In [19]:
```
def walk_up_and_down(floors):
    if floors > 0:
        print("Walk one floor up.")
        walk_up_and_down(floors-1)
        print("Walk one floor down.")
    else:
        print("Reached the top floor!")

walk_up_and_down(3)
```

```
Walk one floor up.
Walk one floor up.
Walk one floor up.
Reached the top floor!
Walk one floor down.
Walk one floor down.
Walk one floor down.
```

The call stack here is straightforward again. Note that also without an explicit "return value" statement, the function returns control to its caller when it is finished, but then without returning a value:

```
walk_up_and_down(3)
    walk_up_and_down(2)
        walk_up_and_down(1)
            walk_up_and_down(0)
            (return)
        (return)
    (return)
(return)
```

Each recursion can be implemented by an iterative loop, and vice versa. For the walking-up-and-down example, a corresponding iterative version is:

In [20]:
```
def walk_up_and_down_iteratively(floors):
    i = 0
    while i < floors:
        print("Walk one floor up.")
        i = i + 1
    print("Reached the top floor!")
    while i > 0:
        print("Walk one floor down.")
        i = i - 1

walk_up_and_down_iteratively(3)
```

```
Walk one floor up.
Walk one floor up.
Walk one floor up.
Reached the top floor!
Walk one floor down.
Walk one floor down.
Walk one floor down.
```

Generally, iterative implementations tend to be faster (because they do not have to create several instances of the same method), but sometimes a recursive solution is more elegant (though maybe more difficult to understand).

Exercises

Please use Quarterfall to submit and check your answers.

1. Leap Years

In our lifetimes (unless we happen to get veeery old) a leap year occurs every four years. But actually, the rule is a bit more involved: A year is a leap year if it is a multiple of 4, but not a multiple of 100, unless it is also a multiple of 400. For example, 1984 and 2000 were leap years, but 1900 and 1985 were not. Write a function is_leap_year(year) that tests if the year is a leap year. If so, the function should return True, and False otherwise. Implement the function using only one Boolean expression. You can use the code below to test your function (the first line defines a list of years to iterate over in the loop – lists will be explained in detail in the next lecture):

```
tests = [1900, 1984, 1985, 2000, 2018]

for test in tests:
    if is_leap_year(test):
        print(f”{test} is a leap year")
    else:
        print(f”{test} is not a leap year")
```

The output should be:

```
1900 is not a leap year
1984 is a leap year
1985 is not a leap year
2000 is a leap year
2018 is not a leap year
```

2. Calculator

Write a program that acts a simple calculator, asking the user if they want to add, subtract, multiply or divide two arbitrary numbers. Define functions add(x,y), subtract(x,y), multiply(x,y) and divide(x,y) for this. (Normally one would not define functions for these basic operators, but this is just an exercise...) After the user has selected an operation, they are asked to enter the numbers x and y. The program calculates and prints the result. The output should be something like:

```
You have four options:
1 - Add
2 – Subtract
3 - Multiply
4 - Divide
Enter choice (1/2/3/4): 2
Enter first number: 34
Enter second number: 53
34 - 53 = -19
```

3. Password Generator

People often use passwords that are too short or too simple and can easily be guessed. (“123456”, “Password” and “12345678” were the most frequently used passwords in 2017!) Moreover, people tend to use the same password for different services, which makes it easy for criminals to take over other accounts once they have

obtained one of the passwords. Thus, it is wise to use passwords that are reasonably long (8 characters minimum), consist of seemingly random sequences of letters (use of special characters is by the way not so important), and have a separate password for each account.

Write a program that helps you to create reasonably good passwords. Therefore define and implement a function create_password(length) that takes the desired length of the password as parameter. If a password shorter than 8 characters is requested, the function should refuse to create it (as it would not be secure). If the requested length is longer, then the function should fill the password with random letters (upper and lower case) and numbers. You can use the following code to test your function:

```
print(create_password(4))
print(create_password(8))
print(create_password(12))
print(create_password(16))
```

The output should be something like:

```
Too short, please create longer password.
None
pk4lU4Cr
UPFzFg6Pn14r
ALdVi3yT0khuxzTr
```

Hint: Consider using the `random.choice()` function from the `random` library, which can be used to select a random character from a predefined string.

4. Basic Statistics

Use the statistics package from the Python standard library to define and implement a function `print_basic_statistics()` with the following characteristics:

- the function takes arbitrarily many numbers as input
- the default case is that the function prints the arithmetic mean, median, standard variation and variance of the input data to the screen
- via a named parameter the calling code should also have the option to select only one of the four to be printed

You can use the following code to test your function:

```
print_basic_statistics(91,82,19,13,44,)
print_basic_statistics(91,82,19,13,44,73,18,95,17,65,output="median")
```

The output should be something like:

```
The mean of (91, 82, 19, 13, 44) is 49.8.
The median of (91, 82, 19, 13, 44) is 44.0.
The standard deviation of (91, 82, 19, 13, 44) is 35.6.
The variance of (91, 82, 19, 13, 44) is 1267.7.

The median of (91, 82, 19, 13, 44, 73, 18, 95, 17, 65) is 54.5.
```

5. Ackermann Function

The Ackermann function (named after the German mathematician Wilhelm Friedrich Ackermann) grows rapidly already for small inputs. It exists in different variants, one of the common definitions is the following (for two nonnegative integers m and n):

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Define and implement a (recursive) function ackermann(m,n) that computes the Ackermann function value for two nonnegative integers m and n. Then, write a test program that computes the results for calling the function with growing m and n of the same value, starting with m = n = 0 and incrementing by 1 in each iteration. The output should be something like:

```
ackermann(0,0) = 1
ackermann(1,1) = 3
ackermann(2,2) = 7
[...]
```

What is the last value that your program computes before you get a `RecursionError`? (Hint: It might be that the outputs in the IPython console in Spyder are too verbose to see anything. You can alternatively run your program from the command line to see more.) What does this error mean?

Extras for the Weekend

CheckiO (https://checkio.org/ (https://checkio.org/)) is a game where you need to code in Python (or JavaScript) to get further. By now you should know enough Python to try it out and solve the challenges there.

If you want to get your brain twisted with something really geeky, have a look at quines. Quines are programs that print themselves (i.e., their own code) during execution, but it is not allowed that they read in their code from the .py file. Here is an example of a quine in Python:

```
q="\nprint('q='+repr(q)+q)"
print('q='+repr(q)+q)
```

Can you write another one?

In [ ]: