Introduction to Programming W8

Functions I.

2022/05/27

1 Function definition

A **function** is a block of code of related statements for a specific task. The basic syntax of a Python function is the following:

```
def function_name(arguments):
        """docstring"""
        statement(s)
```

A function is executed only if it is *called*. Because of this, functions make code reusable and programs more organized. Consider the following code:

```
def rectangle_area(width, height): #function definition
        """Calculates the area of a rectangle,
        given its width and height as arguments"""
        area = width * height
        return area

my_area = rectangle_area(8,5) #function call
print(my_area)
```

The function header starts with the `def` keyword, marking the *function definition*. After that there is the *function name* (in the above instance, `rectangle_area`), to identify the function. Immediately after the function name, there are parentheses that can include some *arguments* or sometimes called *parameters*, separated by commas (here, `width` and `height`). These arguments are passed as inputs and used in the function body when the function is called. It is important that the arguments are ordered, so one has to pay attention when calling a function with arguments. If the function does not take any arguments, the parentheses are empty. The colon indicates the end of the function header. Below the header, there is an optional documentation string or *docstring* to briefly describe the function. You have already used docstrings in your exercise program headers.
One or more indented statements make up the body of the function. The statements in the above example are the one that defines `area` and the `return` statement that returns the value of `area`. The argument variables can be used inside the body, and new variables can be defined (here, `area`). It is important that the variables created here can be only used in the function body! The `return` statement returns the *output value* of the function. It consists of the keyword `return` and an expression. Above, the variable `area` is returned, but one can return `width * height` straight away without assigning it to a new variable. If there is no return statement of a function, it returns `None`. Such functions are sometimes called procedures: the function body is still executed, but the function exists only to do a specific action. For example, if you append an object to a list defined outside of the function but return nothing, the action is still performed. The example below is a function without a return statement:

```
def greeting(name): #function definition
        """Says hi to the person passed in as an argument (must be a string)"""
        print("Hi " + name + "!")

greeting("Bob") #function call
```

Here, although nothing is returned, the `print` statement is executed when the function is called with the argument `"Bob"`. On the other hand, if one would try to use the first example in the following way, there would be no output:

```
def rectangle_area(width, height):
        """Calculates the area of a rectangle,
        given its width and height as arguments"""
        area = width * height
        return area

#next is function call, but we didn't print the output or store it in a variable, used it in
↪  another object, etc.
rectangle_area(8,5)
```

If there is a return statement, usually we want to use the function output somehow. For instance, in the original example, we store the output of the function in a new variable, then print it. Calling the function within the print function would be fine as well: `print(rectangle_area(8,5))`.

In the example below, two functions are defined: `rectangle_area` and `assinging_func`. The `assinging_func` function takes three arguments (two numbers and a dictionary). In the function body, a new element is added to the dictionary from the arguments. The key will be a tuple (remember these are defined within round brackets) from the arguments, and assigning the value involves a function call to `rectangle_area` that takes inputs from the function arguments.

```
def rectangle_area(width, height):
        """Calculates the area of a rectangle,
        given its width and height as arguments"""
        area = width * height
        return area

def assinging_func(rect_width, rect_height, app_dict):
    """Assigns a new key-value pair to a dictionary,
    where the key is a tuple of the rectangle's width
    and height, and the value is the area of the rectangle."""
    app_dict[(rect_width, rect_height)] = rectangle_area(rect_width, rect_height)

rect_areas = {} #still an empty dictionary

#there was no return statement in this function, so we can assume the function does its
↪  intended job without assigning it to a new variable
assinging_func(6,8, rect_areas)
print("Rectangle area dictionary", rect_areas)
assinging_func(12,2, rect_areas)
print("Rectangle area dictionary", rect_areas)
```

Notice that the argument names in the function definition (and how these are used in the function body) are not necessarily the names of the variables you pass into it when calling it! Also, function arguments can be different types of objects, variables, and even functions. Consider the example below:

```
names_scores = {"Bob":40, "Monica":80, "Paul":91}
passed_list = []
failed_list = []


def passornot(score_dict, passed, failed):
    """Function that evaluates if a student is passed or failed,
    and stores the name of the students accordingly in separate
    lists. As an argument, it takes a dictionary where the keys
    are the student names and the values are the scores. The two other
    arguments are two lists to add the names to."""
    for name, score in names_scores.items():
        if score < 60:
            failed.append(name)
        else:
            passed.append(name)
```

```python
passornot(names_scores, passed_list, failed_list)
print("Students passed:", passed_list)
print("Students failed:", failed_list)
```

As mentioned before, the argument names in the function definition (and how these are used in the function body) are not necessarily the names of the variables you pass into it when calling it. However, the code below would be correct as well, and there is no problem with it:

```python
names_scores = {"Bob":40, "Monica":80, "Paul":91}
passed_list = []
failed_list = []

#notice that we use the same names for the lists in the function definition
def passornot(score_dict, passed_list, failed_list):
    for name, score in names_scores.items():
        if score < 60:
            failed_list.append(name)
        else:
            passed_list.append(name)

passornot(names_scores, passed_list, failed_list)
```

However, if you wish to modify the same objects with every function call (in the above example the two lists), the best is not to make them as arguments. For example, if you know that in every function call you want to modify `passed_list` and `failed_list`, the function definition header `def passornot(score_dict):` would make more sense, because the name of the two lists you append to will never change.

Generally, a function call must involve the correct number of arguments. The second call of the function `circle_area` below would result in an error "circle_area() takes 1 positional argument but 2 were given".

```python
def circle_area(r):
        pi = 3.14
        return pi*(r*r)

print(circle_area(3)) #correct number of arguments
print(circle_area(3,4)) #incorrect number of arguments, will result in an error
```

On the other hand, if a function involves *default arguments*, these are not needed to be specified in a function call. Default values for arguments are provided by using the usual assignment operator =, for example:

```python
def greeting(name, message="Hello, ", times=3):
        print((message + name + "!\n")*times)

print("Only providing the name argument, using the default for the others:")
greeting("Bob") #1 positional argument
print("Overwriting the default value for message (order is important):")
greeting("Bob", "Hi, ") #2 positional arguments
print("Overwriting the default value for message and times (order is important):")
greeting("Bob", "Hi, ", 1) #3 positional arguments
print("Overwriting times, but because of the order, the name needs to be specified in the call:")
greeting("Bob", times=5) #1 positional and 1 keyword argument
print("Here, everything is specified by name/keyword, so the order is unambiguous:")
greeting(times=2, message="Hi, ", name="Bob") #3 keyword arguments
```

The function above has three arguments, but providing `message` and `times` in a function call is optional. If the argument variable names are not specified in the function call, the arguments must be passed in order. If specifying the name, the position of that argument is irrelevant. Arguments without default value must be passed in every function call. One important thing to note is that non-default arguments cannot follow a default argument in the function definition. For example, the function definition below is incorrect and would result in an error:

```python
#SyntaxError: non-default argument follows default argument:
def greeting(message="Hello, ", name, times=3):
        print((message + name + "!\n")*times)
```

2 Other function rules, conventions

1. In Python, it does not matter if a function has a return statement or not, the function definition must always be present in your code before calling the function.

```python
my_area = rectangle_area(8,5)

def rectangle_area(width, height):
        area = width * height
        return area
```

With the code above, you will encounter an error that the function you are trying to call is not defined.
2. Although optional, it is always a good idea to document functions with a docstring (especially long, complex ones), because it helps you understand your code weeks, months, years after you wrote it. Maybe even more importantly, it helps others to make sense of your code. Bonus: you can access the docstring of a function with the `.__doc__` attribute (note the double underscores).

```python
def greeting(name):
        """Says hi to the person passed in as an argument (must be a string)"""
        print("Hi " + name + "!")

print(greeting.__doc__)
```

3. Function names have the same naming conventions as variables. You can use letters, numbers and underscores, but cannot use spaces. Also, the name of variables and function names must start with a letter or underscore (you cannot start it with a number). You cannot use the reserved keywords (recheck W2 material if unsure). It is always advisable to use function names that make sense and help the reader understand the role of the function in your code.