CSI 1401 Week 10 Resources

by Allen Yan

CSI 1401 will wrap up chapter 6 this week and host the second exam. The remainder of chapter 6 goes over some situational tools in defining functions. My resources from previous weeks may provide some help for exam review. You can find them at https://www.baylor.edu/support_programs/index.php?id=967950..

As a reminder, I will be leading a group tutoring session on CSI 1401 every Wednesday from 7:00 pm to 8:00 pm (central time). The session will be conducted online via Microsoft Meetings, where I will be available to provide interactive help to students. If the above time window does not work for you, or if you need additional help, Baylor's Tutoring Center also provides 1-on-1 online tutoring appointments. For more information on signing up for group tutoring or individual tutoring, please visit https://www.baylor.edu/tutoring.

Week 10 Important Topics

Chapter 6 – Functions

- Variable scope
- Namespaces
- Function arguments are references
- Keyword arguments and default parameters

1. Variable scope

Python has a somewhat special variable scoping rule: **all variables declared outside of functions are considered *global variables* and can be accessed by any functions**. But **variables declared inside functions are *local variables* and will be deleted when the function ends**. The first thing to take away from this is to never try to access variables declared inside a function after the function has ended. The second is to note an important caveat about addressing global variables inside a function.

While Python allows functions to access and modify outside variables, functions are intended to show modular behavior, with its variables and contents being "boxed-up" and self-contained. This is part of why parameters exist in function definition despite the fact that functions can directly access outside variables. When a function tries to modify a global variable, a **`global`** statement must first be used. The textbook used the example below to demonstrate the use of global statement, without it inside the function, the variable `employee_name` will not be modified by `get_name()`.

```
employee_name = 'N/A'

def get_name():
```

```
    global employee_name
    name = input('Enter employee name:')
    employee_name = name

get_name()
print('Employee name:', employee_name)
```

Do note, however, that if the global variable being modified is mutable (e.g. a list), then it can be modified without the global statement. But keep in mind that the fact you can do it does not mean you should do it, since it is a fairly questionable programming practice. Nevertheless, check the code segment to see how it is done:

```
employee_name = ['N/A']

def get_name():
    name = input('Enter employee name:')
    employee_name[0] = name

get_name()
print('Employee name:', *employee_name)
```

2. Namespaces

Python uses *namespace* to map variables to objects. Whenever a variable is referred to by the program, the Python interpreter looks in the namespace to find the value of the object referenced by the variable. Namespace is used to make variable scopes work. Each scope in function, such as global and local scope, has its own namespace. Python has three major scopes:

1. Built-in scope: Contains all of the built-in names, such as names of built-in functions.
2. Global scope: Contains all globally defined variables and functions.
3. Local scope: Contains all variables defined in the currently-executing function.

When a variable is referenced in Python, the local scope's namespace is checked, then the global scope, and finally the built-in scope. If none of them contains the name, then a *NameError* is generated. For this reason, local variables that duplicate the names of existing global variables will be prioritized when referenced in functions. This is also why declaring a global variable with the same name as a built-in function is allowed, but will disable the said built-in function for the rest of the program.

3. Function arguments are references

One feature of Python functions is that the arguments are always **passed by reference**. This concept is known as *pass-by-assignment* in Python. This is significant because if the argument passed to a function is *mutable*, such as a list or a dict, then any changes the function makes to the argument will be reflected outside the function. If the argument is immutable, which is the case for most basic variable types like int and strings, then the change will only remain in effect within the function. See the example below.

```
def modify(my_list):
    my_list[1] = 99

my_list = [10, 20, 30]
```

```
modify(my_list)
print(my_list)  # my_list still contains 99!
```

This characteristic is important to keep in mind, especially for the students who came from programming in C++, as Python could potentially mess up variables that are intended to be untouched in the global scope. Unlike C++, Python does not have a `const` keyword for function arguments to prevent modification, either. To avoid undesired changes to the objects used as function arguments, one way is to pass the function a copy of the object if the object is mutable. See example below.

```
def modify(my_list):
    my_list[1] = 99

my_list = [10, 20, 30] # Pass a copy of the list
modify(my_list[:])
print(my_list)  # my_list does not contain 99!
```

4. Keyword arguments and default parameters

When a function has a lot of arguments, or when the arguments have very long names, sometimes your Python code can become unreadable. One way to remedy this is to use Python's *keyword argument* feature. As the name implies, Python allows you to map function arguments to parameters by name, instead of by position in the default way. To illustrate, refer to the example below.

```
def print_book_description(title, author, publisher, year,
version, num_chapters, num_pages):
    # Format and print description of a book...

# This is hard to read
print_book_description('The Lord of the Rings', 'J. R. R.
Tolkien', 'George Allen & Unwin', 1954, 1.0, 22, 456)

# Instead, this is much easier to read
print_book_description(title='The Lord of the Rings',
publisher='George Allen & Unwin', year=1954, author='J. R. R.
Tolkien', version=1.0, num_pages=456, num_chapters=22)
```

Keyword argument also allows you to enter arguments into a function by a different order than the one specified in the function definition, and also allows you to skip optional arguments that are specified in the middle of other required arguments.

Speaking of optional arguments, here is how to use those: Python allows functions to have default parameter values, which can be specified in function definition. If a function argument with a default parameter value is skipped during the function call, the function will still run with the argument mapped to the default parameter value instead of giving an error. See code snippet below to see how it is done.

```
def how_is_today(adj = "good"):
    print("Today is {}.".format(adj))

how_is_today()  # Still works without argument!
```

One thing to be mindful about when using default parameter values is to avoid using mutable objects as the default parameter. Because the default argument object is only created once in the program – when the function is defined – any changes made to the argument in question will carry across subsequent function calls. The textbook uses the following example to demonstrate this:

```
def append_to_list(value, my_list=[]):
    my_list.append(value)
    return my_list

numbers = append_to_list(50)  # default list appended with 50
print(numbers)
numbers = append_to_list(100)  # default list appended with 100
print(numbers)
```

Because the default argument object is an empty list that is only created once at the beginning of the program, any function call that uses the default argument object will simply append to the object that is the originally the empty list. As a result, subsequent function calls that use the default argument object will no longer get the intended the empty list. To avoid this behavior, use the default parameter value of `None`, and create the mutable default object inside the function instead. See example below:

```
def append_to_list(value, my_list=None):  # Use default parameter
value of None
    if my_list == None:  # Create a new list if a list was not
provided
        my_list = []

    my_list.append(value)
    return my_list

numbers = append_to_list(50)  # default list appended with 50
print(numbers)
numbers = append_to_list(100)  # default list appended with 100
print(numbers)
```

Useful Resources

- Basic Python Tutorial on GeeksforGeeks:
  https://www.geeksforgeeks.org/python-programming-language/
    - This page provides links to detailed explanations to many entry-level Python concepts along with examples. I encourage taking a look at it if the examples in your textbook were not clear enough.
- Official Python Documentation:
  https://docs.python.org/3/
    - This may be a bit heavy-handed for a beginner-level programmer since the official Python documentation is very thorough and technical. However, learning how to read official documentations is crucial to becoming a good programmer, because the official documentation contains information on everything you can find about Python elsewhere and more. Therefore, I encourage slowly easing yourself into learning how to read the official documentation.