CSI 1401 Week 14 Resources

by Allen Yan

We did it, everyone. The end of semester is near. This week will be the last week of class. Given that this is not a full week, and that no topic is listed for this week on the syllabus, it is likely that the lectures this week will simply be reviews for the final exam. Therefore, I will also focus on reviewing important material for the final exam in this resource.

As we approach the end of the semester, Baylor's Tutoring Center provides 1-on-1 online tutoring appointments for those that need help with studying. For more information on signing up for group tutoring or individual tutoring, please visit https://www.baylor.edu/tutoring.

Final Exam Important Topics

- Chapter 3: Types (Week 3)
- Chapter 4: Branching (Week 4-5)
- Chapter 5: Loops (Week 6-7)
- Chapter 6: Functions (Week 8-10)
- Reading Files

The topics listed above are ones central to all Python programming, and therefore are pretty much guaranteed to appear in the final exam and final practicum. That is not to say you should only study those and ignore everything else taught for the rest of the semester – you should still study everything. Many Python programming topics, with the exception of a few, is more about knowing what things exist and knowing where and how to look, rather than memorizing everything. The topics I listed above are the ones you should prioritize memorizing, since most of them are concept-heavy. Given the limitation in length of a single resource, I will briefly go over each of the topics, please refer to resources from the previous weeks and the textbook for additional details.

1. Chapter 3: Types (Week 3)

Chapter 3 covers the basics of strings, lists, sets, and dictionaries in Python. Due to length restriction I cannot go over all of them. Since most students struggle with dictionaries the most, I will focus on dictionaries.

Dictionaries allow users to store data using associative relationships: insertions in dictionaries must be done using key-value pairs, the key is a term that describes it's associated value and will be used to access the stored value in the dictionary later, and the value will be the actual data stored. Dictionaries declaration has a somewhat complicated syntax, refer to the example below:

```
presidents = {
    'George W. Bush': 2001,
    'Barack Obama': 2009,
```

```
    'Donald Trump': 2017
}
```

The above example creates a dictionary named presidents with names of U.S. presidents as the key and their year of inauguration as the value.

A table of common dictionary operations is listed below.

Operation	Description
`dict[k] = val`	Inserts value val into the dictionary with key k, if k already exists in the dictionary, then the associated value is updated to val
`del dict[k]`	Deletes the key k and its associated value from the dictionary

2. Chapter 4: Branching (Week 4-5)

From chapter 4, if-else branching is by far the most important topic to take away since it is used in essentially every program after. Like the name implies, if-else branches allow the program to *branch out* do one thing *if* a certain condition is met, but branch to do another thing otherwise. For example, consider the example program below which tells whether a number is even or odd:

```
my_num = int(input('Enter a number: '))
if my_num % 2 == 0:
    print("Number is even.")
else:
    print("Number is odd.")
```

If your program requires a more sophisticated level of branching than the simple one-or-the-other logic used in the example program above, you may elect to use else-if statements between your initial if and else statements. Else-if statements allow the program to evaluate additional conditions you set before defaulting to the else statement. You may add as many else-if statements as you need, and Python will never complain about it (although adding too many of them will make the code difficult to read). For example, the “check fruit” program below demonstrates else-if statements, note that else-if statements are denoted using the keyword `elif` in Python:

```
my_fruit = input('Enter a fruit: ')
if my_fruit == "Apple":
    print("Apples are delicious!")
elif my_fruit == "Banana":
    print("Bananas are great!")
elif my_fruit == "Strawberry":
    print("Strawberries are yummy!")
elif my_fruit == "Grape":
    print("Grapes are nutritious!")
else:
    print("Sorry, I don't know what fruit that is.")
```

Keep in mind that the else statement at the very end is always guaranteed to execute if none of the above if and elif statements are executed. Sometimes you may not want broad behaviors like that in your program. In which case, you can leave out the entire else statement and have your code consist of only if and/or elif statements, it is syntactically okay to do that.

You can also put if statements inside other if or elif statements to create *nested if-else statements*, where the inner if statements are only evaluated if the condition for the outer if statement is met. See the following example code:

```
my_fruit = input('Enter a fruit: ')
if my_fruit == "Apple":
    print("Apples are delicious!")
elif my_fruit == "Banana":
    num = int(input('How many? '))
    if num == 1:
        print("1 banana is not enough!")
    elif num == 2:
        print("2 banana is just right!")
    elif num == 3:
        print("3 banana is too much!")
```

3. Chapter 5: Loops (Week 6-7)

Chapter 5 covers looping in depth, with for loops posing a slightly higher importance. For loop is a very powerful loop structure in Python. It is very convenient for looping over a container, or iterating through a range of numbers. The syntax for the for loop works as follows:

```
for loop_variable in container_or_range:
    # loop body here
```

For loops could be used in two ways depending on the situation. One of them is simply looping through a range of numbers. This is usually the more verbose way, especially when the for loop is used to iterate through a container. See example below:

```
my_list = ['a', 'b', 'c']

for i in range(len(my_list)):
    print(my_list[i])
```

The `range()` function shown above returns a series of integers ranging from 0 to 1 less than the length of the list. Sometimes looping by a range of numbers is the only option available, and in those situations, it is perfectly fine to use that approach. But for looping through containers, there is usually a more convenient short-hand that does the same thing:

```
my_list = ['a', 'b', 'c']

for element in my_list:
    print(element)
```

The `element` variable in the above code is a temporary variable managed by the for loop, and it automatically updates to the value of the next element inside the container for each iteration of the

loop. This method could save you a bit of work on long programs and make your code more readable. However, be aware that this method of iterating through a container does not allow you to change the contents of the original container. For example, if you execute the code below, you will notice the contents of `my_list` did not change:

```
my_list = ['a', 'b', 'c']

for element in my_list:
    element = 'd'

for element in my_list:
    print(element)
```

This is due to the fact that when you structure a for loop to update the loop variable with the contents of a container, Python binds the value of the items in the list to the loop variable name, instead of the actual items objects. If you need to modify the contents of a container, then you have to use the number-based looping approach, or use the `enumerate()` function, which will return both the index and the element of the container. I have included both approach in the example below:

```
my_list = ['a', 'b', 'c']

# number-based
for i in range(len(my_list)):
    my_list[i] = 'd'

for element in my_list:
    print(element)

# enumerate approach
for index, element in enumerate(my_list):
    my_list[index] = 'e'
    # element does not update right away
    print(element)

# now the updates will show
for element in my_list:
    print(element)
```

The above example also shows that the `element` variable is only updated once at the beginning of each iteration of the loop, so if the content of the list is changed inside an iteration, the `element` variable will not reflect that change within the same iteration.

For loops are very convenient compared to while loops, but does not completely replace the latter. It is important to be able to make the judgement call of which loop to use for different programs. Personally, my rule of thumb is that if the loop is iterating through a container, or if the number of looping iterations is known, then use for loop, otherwise use while loop. The course's textbook offers the following advices:

1. *Use a for loop* when the number of iterations is computable before entering the loop, as when counting down from X to 0, printing a string N times, etc.
2. *Use a for loop* when accessing the elements of a container, as when adding 1 to every element in a list, or printing the key of every entry in a dict, etc.
3. *Use a while loop* when the number of iterations is not computable before entering the loop, as when iterating until a user enters a particular character.

4. Chapter 6: Functions (Week 8-10)

Chapter 6 covers user-defined functions in painstaking details, and I highly recommend reviewing this chapter thoroughly as many of its topics are fairly complex but crucial. For this review, I will cover the basics of function definition.

A function is a named block of code. In Python, a function definition consists of a *header* and a *body*. The header of the function includes the name of the function and its parameters, and the body of the function contains all of the code that would be run when the function is called. See below for an example of a function definition:

```
def function_name(parameter_1,parameter_2):  # function header
    return parameter_1+parameter_2  # function body
```

Function *parameters* are the variables defined in a function header that will come from outside of the function when the function is called. When a function is called, the user will provide the function parameters with values called *arguments* to work with. All defined parameters must be provided with an argument during function call in order for the function to execute properly. A function can have one or multiple parameters.

Functions have the option to return some value at the end of its execution. The returned value can then be used for other computations outside the function. For example, Python's standard `len()` function returns the length of the argument passed to the function. To return a value in user-defined function, simply use the `return` keyword followed by the desired variable to return. A function can have any number of return statements, which allows for the option to return different things depending on how the function runs. But keep in mind that during any particular function call, only one return statement will be executed. Because **the function will end as soon as a return statement is reached**. For this reason, always make sure when defining a function that all of the work in the function is finished before calling the return statement. See the example below for a demonstration:

```
def my_func(my_list):  # bad function
    return sum(my_list)  # returns here
    for elem in my_list:  # this printing loop will never be
reached
        print(elem)


def my_func2(my_list):  # good function
    for elem in my_list:  # printing before returning fixes the
problem
```

```
        print(elem)
    return sum(my_list)  # returns here


nums = [1, 2, 3, 4, 5]
print(my_func(nums))  # prints only the sum of nums
print(my_func2(nums))  # prints the contents of nums, followed by
the sum of nums
```

Lastly, keep in mind that each function call can only return one variable. If you need your function to return multiple variables, you can package them into a container, such as a list or a dictionary, then return the container as a single variable.

5. Reading files

This is the only listed topic that was not covered in the assigned portion of the textbook, but rather during lecture only. There is a chapter in the textbook dedicated to file operations, which is chapter 13. You may read that chapter if you wish, but it likely contains way more information than you need. For the final exam, you can probably get away with only knowing how to read from a file. Reading files in Python can be summarized into the following steps:

1. Open file object
2. Read the file using either `read()` or `readlines()`
3. Close file object

Whether you choose to use `read()` or `readlines()` is entirely a matter of preference and need. `read()` reads the entire file into a single string, which you then will probably need to process on your own, most likely using the `split()` function. `readlines()` reads the file into a list of strings, with each line stored in a separate element. You can also use `readline()` if you want to read a single line at a time instead of the entire file at once. See code example below.

```
f = open('myfile.txt')  # step 1: create file object

contents = f.read()  # step 2: read file text into a string
contentslist = f.readlines()  # step 2 alternative : read file
text into a list of strings separated by lines
line = f.readline()  # step 2 alternative: read a single line

f.close()  # step 3: close the file
```

Useful Resources

I hope that the CSI 1401 resources I shared with you this semester were helpful and efficient in addressing your academic needs. As we approach the end of the fall semester, I would like to let you know that this is the last resource for this semester. All previous resources for this semester can be found here: https://www.baylor.edu/support_programs/index.php?id=967950.

- 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.