CSI 1401 Week 12 Resources

by Allen Yan

Similar to last week's in-depth focus on strings, CSI 1401 will introduce some advanced programming techniques involving lists and dictionaries in Python over the next two weeks. Compared to the concept-heavy contents of the earlier weeks, the topics for these two weeks are more about memorizing syntax and should be less stressful. You will also find plenty of material that have already been introduced in earlier weeks.

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.

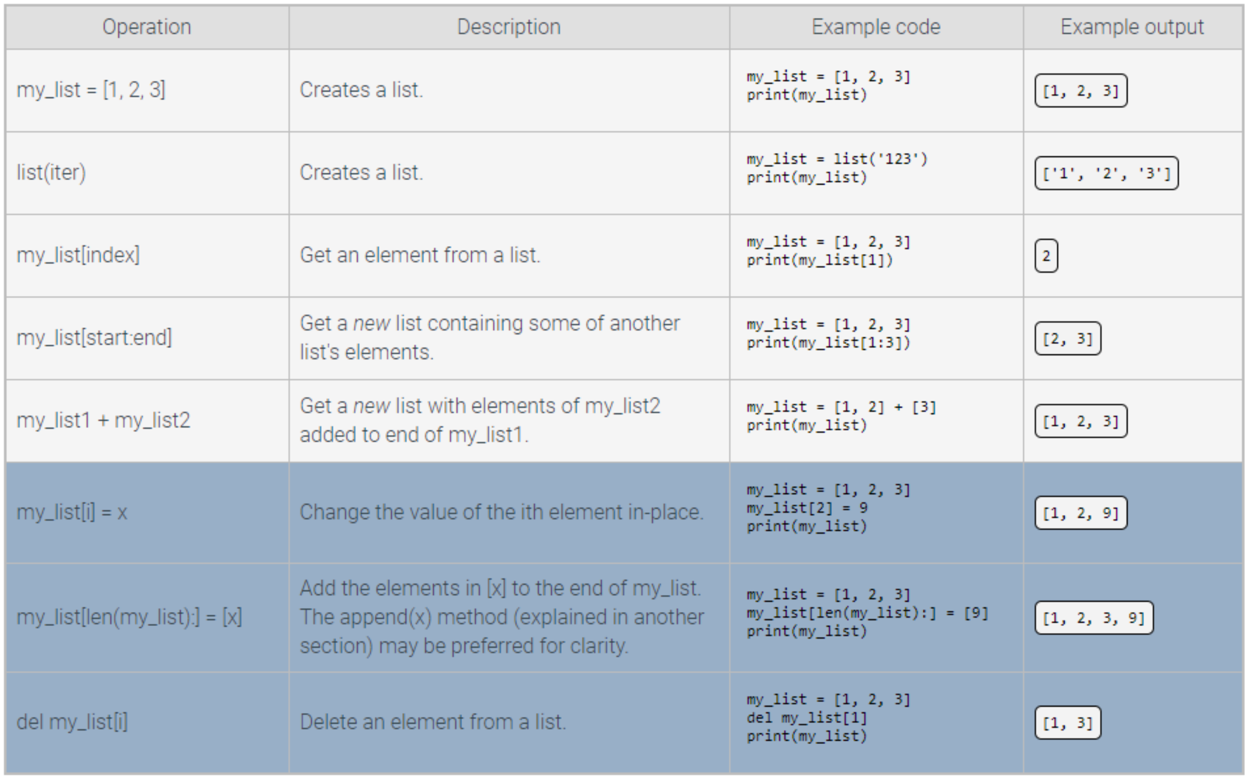
Week 12 Important Topics

Chapter 8 – Lists and Dictionaries

- Common list operations
- List methods
- List slicing

1. Common list operations

A list is a **container object** in Python, where related objects are stored sequentially and labeled by indices. Lists are the most commonly-used container types because they are straightforward. Below is a table of the most common list operations, where the highlighted rows indicate operations that modify a list in-place. Most of the listed operations should be review at this point.



Operation	Description	Example code	Example output

my_list = [1, 2, 3]	Creates a list.	`my_list = [1, 2, 3]`
		`print(my_list)`
			`[1, 2, 3]`

list(iter)	Creates a list.	`my_list = list('123')`
		`print(my_list)`
			`['1', '2', '3']`

my_list[index]	Get an element from a list.	`my_list = [1, 2, 3]`
		`print(my_list[1])`
			`2`

my_list[start:end]	Get a *new* list containing some of another list's elements.	`my_list = [1, 2, 3]`
		`print(my_list[1:3])`
			`[2, 3]`

my_list1 + my_list2	Get a *new* list with elements of my_list2 added to end of my_list1.	`my_list = [1, 2] + [3]`
		`print(my_list)`
			`[1, 2, 3]`

my_list[i] = x	Change the value of the ith element in-place.	`my_list = [1, 2, 3]`
		`my_list[2] = 9`
		`print(my_list)`
			`[1, 2, 9]`

my_list[len(my_list):] = [x]	Add the elements in [x] to the end of my_list. The append(x) method (explained in another section) may be preferred for clarity.	`my_list = [1, 2, 3]`
		`my_list[len(my_list):] = [9]`
		`print(my_list)`
			`[1, 2, 3, 9]`

del my_list[i]	Delete an element from a list.	`my_list = [1, 2, 3]`
		`del my_list[1]`
		`print(my_list)`
			`[1, 3]`

Image snipped from course zyBooks

2. List methods

Similar to strings from last week, Python lists come with a slew of library functions that can make your life easier in occasions where they are needed. Once again, it is not as importance to memorize the syntax of each of these methods as it is to know that they exist – you can always look up their exact syntax as long as you know such a function exists, and most of the time looking up documentations is not considered cheating. A brief description of the list methods covered in this chapter is listed below.

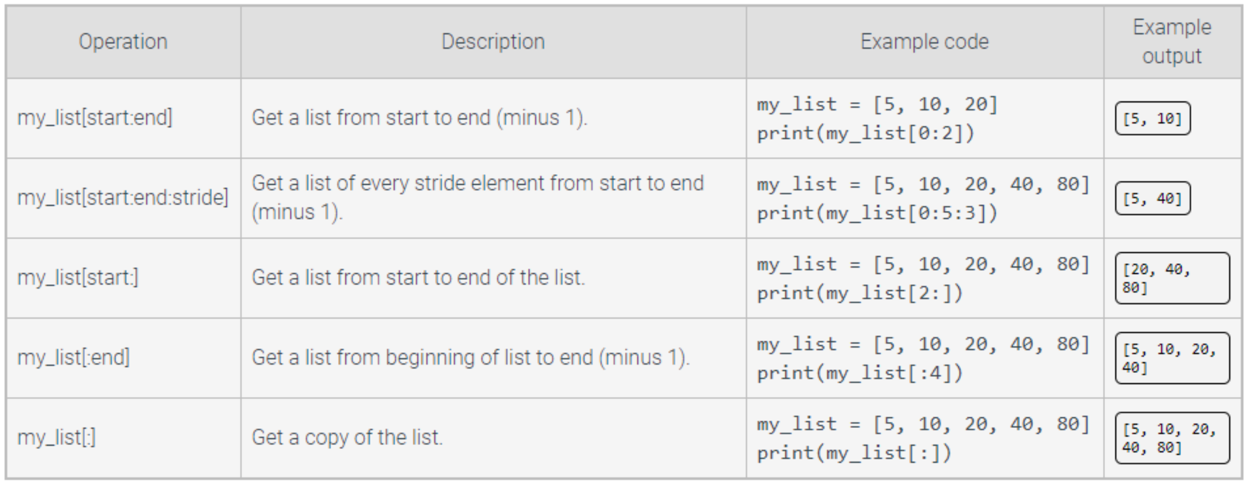
- **`list.append(x)`** – Add x to the end of the list. Use `list.extend()` if multiple elements are appended.
- **`list.extend([x])`** – Similar to `list.append()`, but adds all elements in [x] to list, you can pass in a list literal or a list variable to the function.
- **`list.insert(i,x)`** – Inserts x into list before position i. If inserting multiple elements is intended, use slice assignment instead.
- **`list.remove(x)`** – Removes first item in list with value x.
- **`list.pop()`** – Removes and returns last item in list.
- **`list.pop(i)`** – Removes and returns i-th item in list.
- **`list.sort()`** – Sorts the items of list in-place. If returning a sorted copy of the list is desired, use `sorted(list)` instead.
- **`list.reverse()`** – Reverses the elements of a list in-place. If returning a reversed copy of the list is desired, use `list[::-1]` slice notation instead.
- **`list.index(x)`** – Returns index of first element in list with value x.
- **`list.count(x)`** – Returns number times value x is in list.

- **`all(list)`** – Returns True if every element in list is True(!= 0), or if list is empty.
- **`any(list)`** – Returns True if any element in list is True.
- **`max(list)`** – Returns the maximum element in list.
- **`min(list)`** – Returns the minimum element in list.
- **`sum(list)`** – Returns the sum of all elements in list.

The above list of methods are by no means exhaustive. There are tons of other niche list methods in Python. If you ever run across an occasion where you want to perform an operation on lists but feel like it can be generalized pretty easily, you can try to search up if there is already an existing function for it in Python. Chances are, there probably is. This is one of Python's biggest perks over other programming languages – the amount things that is already done for you.

3. List slicing

Recall the string slicing notation from last week. We briefly mentioned that the same slicing notation also works the same way on Python lists, and here they are. Below is a table of the most-commonly used list slicing operations in Python.



Operation	Description	Example code	Example output
my_list[start:end]	Get a list from start to end (minus 1).	`my_list = [5, 10, 20]` `print(my_list[0:2])`	`[5, 10]`
my_list[start:end:stride]	Get a list of every stride element from start to end (minus 1).	`my_list = [5, 10, 20, 40, 80]` `print(my_list[0:5:3])`	`[5, 40]`
my_list[start:]	Get a list from start to end of the list.	`my_list = [5, 10, 20, 40, 80]` `print(my_list[2:])`	`[20, 40, 80]`
my_list[:end]	Get a list from beginning of list to end (minus 1).	`my_list = [5, 10, 20, 40, 80]` `print(my_list[:4])`	`[5, 10, 20, 40]`
my_list[:]	Get a copy of the list.	`my_list = [5, 10, 20, 40, 80]` `print(my_list[:])`	`[5, 10, 20, 40, 80]`

Image snipped from course zyBooks

One thing to keep in mind with list slicing is that all slicing operations return a copy of the sliced portion of the list. This is less importance for string slicing because strings are immutable objects in Python. But Python lists are mutable, and there are list slicing operations that are made to perform in-place modification of lists, those are called **slice assignments**. Regular slicing and slice assignment share almost the exact same syntax, so it is very easy to confuse the two, but it is important to be able to distinguish the two, because they are very different operations. See code example below.

```
a = [1, 2, 3, 4, 5]
b = a[0:3]  # normal slicing, returns copy
print(*a)  # [1, 2, 3, 4, 5]
print(*b)  # [1, 2, 3]

a[0:3] = [100, 101, 102]  # slice assignment, modifies in-
place
print(*a)  # [100, 101, 102, 4, 5]
```

As you can see, the two slicing operations in the code above had identical syntax, but behaved very differently. The first slice returned a copy of the sliced list and assigned it to b, and the second slice made available the portion of the original list a for modification. A good rule of thumb to tell regular slicing apart from slice assignment is that **if slicing happens on the left side of an assignment operator, it's probably slice assignment; if slicing happens on the right side of an assignment operator, it's likely to be regular slicing**.

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.