Session 3: Lists

Lists are an important part of a program. We would like to store a bunch of items in one variable. This lets us

- avoid having to define a new variable for every value we want to store
- process collections that we don't know the size of when writing the code (like a list of email addresses or a list of scraped websites)
- process entire collections simply and efficiently

Basics

Suppose we wanted to create a shopping list app. We would like to be able to store our user's shopping list and retrieve the items from it on demand.

Here is how that list might look:

```
shopping_list = ["bread", "smoked salmon", "cherry tomatoes", "cream cheese"]
```

In Python, we can also `print` the whole list, to view all the items:

```
shopping_list = ["bread", "smoked salmon", "cherry tomatoes", "cream cheese"]
print(shopping_list)
```

Or we can access the items one by one:

```
shopping_list = ["bread", "smoked salmon", "cherry tomatoes", "cream cheese"]
print(shopping_list[0]) # Prints 'bread'
print(shopping_list[1]) # Prints 'smoked salmon'
print(shopping_list[2]) # Prints 'cherry tomatoes'
print(shopping_list[3]) # Prints 'cream cheese'
```

As we can see, a list is an numbered collection of items. Getting an item out of a list by its number is called **indexing**.

In previous sessions, we've seen how `range(x)` starts counting from 0. Similarly, in Python, **list indexing starts at 0**. What that means is that the first item is *at* **index** 0, the second item is at index 1, etc. We access the item at index `i` (where `i` is an integer) by writing `shopping_list[i]` - the name of the list, and then the index in square brackets.

Now what would happen if we try to access the item at index 4 of our shopping list? Let's try:

```
shopping_list = ["bread", "smoked salmon", "cherry tomatoes", "cream
cheese"]
print(shopping_list[4]) # ERROR: list index out of range
```

We get an index error. This is because there is no item at index 4. Make note of how the list has 4 items, but the highest index is 3. In general, a list with `n` items will have a highest index of `n-1`.

We can get the *length* of the list (i.e. the number of items in it), by calling the `len` function:

```
shopping_list = ["bread", "smoked salmon", "cherry tomatoes", "cream
cheese"]
length = len(shopping_list)
print("The length of the list is: " + str(length))
```

Note that the `len` function returns an `int`, so we need to convert that to a `str` before printing.

As a final important note, the empty list (list with 0 items) is `[]`. Thus, if you want to create a list that you fill later:

```
my_list = []
```

The utility of lists

Imagine trying to manage your shopping list without a list. We'd have to use a different variable for each item in the shopping list, like this:

```
shopping_list_item_1 = "bread"
shopping_list_item_2 = "smoked salmon"
shopping_list_item_3 = "cherry tomatoes"
shopping_list_item_4 = "cream cheese"
```

This is certainly annoying, but what's even more so is that there is no effective way to manipulate all items using loops. For example, if we would like to print everything in the shopping list, this is the only thing we can do:

```
print(shopping_list_item_1)
print(shopping_list_item_2)
print(shopping_list_item_3)
print(shopping_list_item_4)
```

However, with lists, we can store all items in one single variable in one single line.

```
shopping_list = ["bread", "smoked salmon", "cherry tomatoes", "cream
cheese"]
```

And we can print the whole list at once using `print(shopping_list)`.

Okay, but that's just for avoiding annoyances. Here's an example of something you definitely can't do without lists. This code allows the user to edit their shopping list:

```
index = int(input("Edit item at which index: "))
value_initial = shopping_list[index]
value_final = input("Replace item "+ value_initial + " with: ")
shopping_list[index] = value_final
```

Since we don't know ahead of time which item the user wants to edit, we can't simply assign `shopping_list_item_x = input(...)`.

More list operations

We can append an element to a list, adding it to the end, or pop an element from the list, taking it off the end:

```
# We forgot to add mayo to the list so we can add it later like so:
shopping_list.append("mayo")
print(shopping_list) # [ "bread", "smoked salmon", "cherry tomatoes",
"cheddar", "mayo" ]

# It turns out we didn't need the mayo after all
shopping_list.pop()
print(shopping_list) # [ "bread", "smoked salmon", "cherry tomatoes",
"cheddar" ]
```

By specifying an index within the list, we can also insert or pop items from anywhere in the list:

```
# We urgently need toilet paper
shopping_list.insert("toilet paper", 0)
print(shopping_list) # [ "toilet paper", "bread", "smoked salmon", "cherry
tomatoes", "cheddar" ]

# It turns out we didn't need the toilet paper after all
shopping_list.pop(0)
print(shopping_list) # [ "bread", "smoked salmon", "cherry tomatoes",
"cheddar" ]
```

Alternatively, you can remove an element by value:

```
shopping_list.remove("mayo")
```

... though this is less common, and also more computationally expensive because the computer needs to go through the entire list to find the mayo, similarly to how you would go down a list from the start to find it.

You can also join two lists together:

```
shopping_page1 = ["bread", "smoked salmon", "cherry tomatoes"]
shopping_page2 = ["cheddar", "mayo"]
shopping_list = shopping_page1 + shopping_page2
# shopping_list is now [ "bread", "smoked salmon", "cherry tomatoes",
"cheddar", "mayo" ]
```

Note: you may have seen what Python calls lists called *arrays* in other programming languages. In most programming languages there is a difference but it is not relevant to Python or this course.

Loops with lists

Recall the `for` loops from last week:

```
n = int(input("Please enter a number: "))
for i in range(1, n): # including 1 excluding n
    print(i)
```

Loops are exceeding useful for *iterating* over lists.

For loops

```
shopping_list = [ "bread", "smoked salmon", "cherry tomatoes", "cream
cheese" ]
n = len(shopping_list)
for i in range(0, n):
  print("I need to buy " + shopping_list[i])

# This will print out:
# I need to buy bread
# I need to buy smoked salmon
# I need to buy cherry tomatoes
# I need to buy cream cheese
```

In fact, we can further simplify that `for` loop, so that we don't need the variables `n` and `i` anymore. In most languages, this is a **for-in** loop:

```
shopping_list = [ "bread", "smoked salmon", "cherry tomatoes", "cream
cheese" ]
for item in shopping_list:
  print("I need to buy " + item)

# This will print out:
# I need to buy bread
# I need to buy smoked salmon
# I need to buy cherry tomatoes
# I need to buy cream cheese
```

Although, if you look back at the original `for` loop with the `range` in it, the word `in` is present there as well.

This gives us an insight as to what `range(a, b)` is doing. It's a function that takes two arguments (a,b) and gives us something like a list `[a, a + 1, a + 2, ..., b - 2, b - 1]` so we were iterating over lists all along.

Slicing

Sometimes you want to operate on, or separate out, a section of elements from a list.

For example, maybe your toddler has put 700 toys in your shopping cart, and you want to buy only the first four.

As another example, *merge sort* is a program that sorts a list by splitting it into two halves, sorting each half, and recombining the two sorted half-lists into one sorted list.

```
shopping_list = [ "bread", "smoked salmon", "cherry tomatoes", "cream
cheese", "stuffed bear", "stuffed bear", "stuffed bear", "stuffed bear",
"stuffed bear" ]
grocery_list = shopping_list[0:4]
print(grocery_list) # [ "bread", "smoked salmon", "cherry tomatoes",
"cheddar" ]
```

List slicing allows you to extract a segment of a list by specifying two indices: left and right.

The left index is the starting index: the item at `your_list[l]` appears in the slice `your_list[l:r]`. But, once again with the pattern of "stopping just short", the right index is the index that you stop just short of: the item at `your_list[r]` does NOT appear in the slice `your_list[l:r]`.

Exercises

Exercise 1. (Free Response)

Create a list of a few countries or cities you want to visit. Print the whole list on one line. Print the second item of the list. Print the length of the list.

Exercise 2. Backwards

(a). Recall Exercise 5 from last week:

```
for i in range(5, 0):
  print("Backwards: " + str(i))
for i in range(5):
  print("Subtraction: " + str(5 - i))
```

Using this, write a program that prints the elements of a list in reverse order.

(b). Read the following code block and predict its result.

```
items = ["foo","bar","foobar"]
backwards = items.reverse()
print("Items: " + str(items))
print("Backwards: " + str(backwards))
```

Now run the code. What happens? Is this what you expected?

Exercise 3. Call In

When a call center gets a phone call, if there is no operator available to take the call, the caller's phone number is added to a queue.

We'll start with the following line of code:

```
queue = [] # the caller queue starts empty
```

(a). Write code that prompts the user to enter a string, then adds that string to the list. (You do not have to check that the string forms a valid phone number.) Repeat this four times, then print the list.

(b). Phone calls should be handled in the order they arrive in. Write code that removes a phone number from the list and prints what remains in the list. Repeat this four times.

Your output should look something like:

```
Incoming call from: 23300000
Incoming call from: 07777111111
Incoming call from: (866) 965-3590
Incoming call from: unknown
The queue is: ['23300000', '0777111111', '(866) 965-3590', 'unknown']
Handled a call. Queue: ['0777111111', '(866) 965-3590', 'unknown']
Handled a call. Queue: ['(866) 965-3590', 'unknown']
Handled a call. Queue: ['unknown']
Handled a call. Queue: []
```

Exercise 4. Shopping List

Let's keep developing the shopping list program.

(a). Begin by prompting a user to enter the number of things they want on their list and then ask them for each separate item. The program should output the items on the list in the same order, each on a separate line, after the user's done with their input.

```
Please enter the size of your shopping list: 3
Please enter item number 0: bread
Please enter item number 1: smoked salmon
Please enter item number 2: cherry tomatoes
Your shopping list is:
bread
smoked salmon
cherry tomatoes
```

(b). Modify your program slightly so that it displays the item number starting from 1 instead of 0.

```
Welcome to the shopping list app!
Please enter the size of your shopping list: 3
Please enter item number 1: bread
Please enter item number 2: smoked salmon
Please enter item number 3: cherry tomatoes
Your shopping list is:
bread
smoked salmon
cherry tomatoes
```

We'll come back to the shopping list program next week, so make sure you save your progress!

Exercise 5. Take

Haskell is a programming language that doesn't provide the builtin functions of `range` and `len`. But it does provide the builtin functions of `take` and `drop`, which Python lacks.

Using list slicing, complete the following code that splits a list in a place where the user specifies:

```
items = [] # fill this list with any items you like
n = int(input("Split the list where: "))

take = # the first n elements of the list
drop = # the rest of the list without the first n elements

print("List has been split: ")
print(take)
print(drop)
```

We'll complete our implementation of `take` and `drop` in week 5.

Exercise 6. Counting Duplicates

Write a program that takes the first element of a list, and checks how many times that element is duplicated in the list.

For instance, the list [1,5,3,2,5,1,6,2,7,3,1,6,1,9,6,3,7] starts with a 1 and contains four 1s, so your program should output:

```
The first element of this list (1) appears 4 times in the list
```

If we remove the first 1 from the list, your program should instead output

```
The first element of this list (5) appears 2 times in the list
```

It might help to maintain a running "tally" variable:

```
tally = 0
for item in items:
  # ... complete the code
  # sometimes, or under some conditions, write tally = tally + 1
```

For full marks, if the first element of the list appears 1 time, your program should print "appears 1 time" instead of "appears 1 times".

Exercise 7. Adding 1

Given a number represented by its digits, write a program that adds 1 to that number.

```
number = [1,5,6,9,9] # representing the number 15699
# add 1 to number here...
print(number) # you should get [1,5,7,0,0]
```

Exercise 8. Look-And-Say (Extremely Hard)

The look-and-say sequence starts with the number 1. Then, you say exactly what you see: "one one", so the next number is 11. The number after that is "two ones", 21, followed by "one two and one one", 1211.

After that is 111221.

The code we're starting off with is:

```
current_sequence = [1]

next_sequence = [] # Store the next sequence
currently_looking_at = current_sequence[0] # Start looking at the first
digit
digit_chain_length = 0 # The first digit starts a chain
for digit in current_sequence:
  # Construct the next sequence here
  pass

# Set the current sequence to the next sequence
current_sequence = next_sequence
```

(a). Complete the code that, when given a number in the look-and-say-sequence (represented by a list), computes the next number in the look-and-say sequence.

Here's some observations that might help:

- Your program should have different behaviours depending on whether the next digit in the list is the same as or different from the digit you're `currently_looking_at`.
- Would it be useful to add 1 to a tally variable every time you see a digit that's the same as the one you're `currently_looking_at`?
- What do you need to add to the `next_sequence` every time you see a digit that's not the same as the one before it? What variables will you have to update / reset?

(b). Wrap your code in a `for` loop to allow this program to calculate the next `n` numbers in the look-and-say sequence, where `n` is a number the user specifies. Print that many numbers in the sequence one by one, starting with 1. (Be careful of off-by-one errors!)

```
Look-and-say depth: 6
1
11
21
1211
111221
312211
```

You will want to use what you learned in session 2, exercise 9.

You can find all the solutions here.