**Instructor example**

User photo

[Applause Interactive](https://www.udemy.com/user/applause-interactive/)

**Challenge One: Add numbers to your lists.**

**We coded two automatically generated lists in Python: The myMenu list and myShoppingList.**

**We also changed our code to use FOR loops in order to print each item in the list on a new line, without all of the special list related characters like commas and brackets (which appear when simply printing the whole list with a statement like print(myMenu) or similar).**

**So if we're printing each list item on a new line..**

**- Can you think of a way to print numbers before each item - so the numbers appear on the left, at the start of each line, like so:**

**1. Item 1**

**2. Item 2**

**3. Item 3**

**and so forth..**

**Here's a clue from the Python tutorial: https://docs.python.org/3/tutorial/datastructures.html#more-on-lists**

**Consider getting an element's position in the list, during the for loop cycle ;)**

**Have a go, and if you get stuck, check out the solution!**

**NB: Video solutions coming soon so stay tuned for those!**

As is so often the case, there are many ways to solve this. Here's just one way!..

If you look at the Python > Docs > Tutorial (**5.1 Data Structures - More on Lists**) found here: <https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

You'll discover a list function called list.**index(x)**

The docs tell us this function will:

"Return zero-based index in the list of the first item whose value is equal to *x*. Raises a [ValueError](https://docs.python.org/3/library/exceptions.html" \l "ValueError" \o "ValueError" \t "_blank) if there is no such item."

We can use this function in our loop, like so..

for dish in myMenu:

print(myMenu.index(dish) + 1, dish)

What we're doing here, is:

1. Using the forloop to get each element in the myMenu list, and store it temporarily in the dish variable.
2. Call the list.**index()** function on myMenu, passing in the dish variable as the '**x**' argument. This returns that dish's current index (position) in the list, and we know from the docs that the index is "zero based", so it starts at zero.
3. A list starting at zero isn't too user friendly, so we always add 1 with our + 1 code. Now the index number will start at 1, not zero.
4. We print the new index number, followed by a space, and the current dish's string value (name) on the same line.

***Solved!***

Why didn't we use + dish to print the dish after the line number? Well the list.**index()**function returns an **integer**, and we can't join an integer to a **string** using the + operator. That's why we just used a comma and space in the print statement. If we wanted to use +, we'd have to call the str() function to convert the list number integer to a string, and then concatenate. We'd also need to manually join a space, like so...

print(str(myMenu.index(dish) + 1) + " " + dish)

Another method might be to code your own counter variable, and then increment it on each loop through the for loop.

**Challenge Two: Validate the y/n user input.**

**In step 3 of our Munch app in Python, we asked the user: "Would you like a shopping list for this menu?"**

**Let's add some validation, to make sure the user enters a valid input before proceeding.**

**- Can you think of a way to make it really clear to the user that we want them to enter a yes or no answer?**

**- Can you think of a way to validate the answer, so that if the user enters an input other than a lowercase 'y', or 'n', the app asks them to try again?**

**- Can you think of a way to allow them to proceed by also accepting an uppercase 'Y' or 'N', for example, if they have their caps lock on?**

**This is a tricky one! The clue is to use a loop, but exiting a loop hasn't been covered in detail yet, so don't worry if you need the solution. This is a good one to think about on your own before seeing the solution, so you really understand the problem, before you see how many coders solve it :)**

**NB: Video solutions coming soon so stay tuned for those!**

So as mentioned, this is a tricky one! But let's *break* it down ;)

First of all, let's try to make it really clear to the user what we are looking for in terms of a **valid input**. Good user experience design should always make the desired input simple and obvious..

How about we change our input string to this:

answer = input("Would you like a shopping list for this menu? Enter 'y' or 'n'... ")

That's clearer.

Next up, we want to check for either a "y" or "n" string input (stored in our answer variable). Those two string inputs are what we have decided to 'allow' as a valid response. If the response is valid, we can let it through into our existing if/else logic, and if it's not valid, we want to ask the user for their input again. In fact, if the response is not valid, we want to ask again and again, **repeating until we get a valid response** - only then should the program be allowed to continue. Another way to think about it is, only then, should we allow the response to continue into the rest of our logic/control flow.

So how can we do this?

Well how about a while loop? The while loop is perfect for performing this kind of validation, and as is so often the case, there are a few ways to achieve what we're looking for.

Here's one way, using pseudo code..

ask for input

while input not correct:

ask for input

But this method, although simple, isn't the best method. Why? Because there's a common coding principle known as DRY, or **Don't Repeat Yourself!**

Using the above method, we're repeating the same line of code twice (the lines where we ask the user for an input). This breaks the DRY rule. If we get into the habit of doing this, we could end up with lots of duplicate code - increasing the likelihood of introducing bugs later on.

So how about another way, still using the while loop? Well, here's a popular method (again, in pseudo code)...

while True:

ask for input

if input correct:

break

else:

notify the user

We use a little trick here. We code a while True loop. Basically, this is an infinite loop, as True is by it's very nature **boolean True**, so the loop will always repeat. Coding while True is like saying, while 1 == 1. This would evaluate True. Well it may seem a bit odd at first, but using boolean logic, True always evaluates True - it is True!

So while True gets the loop running. Then, we ask the user for their input within the loop. If we detect a correct input, we use a special break statement to **break out of the loop**. As soon as we tell Python to break out of the loop, it can continue running the code below, leaving the loop behind. This is like saying *"continue with the rest of the program now.."*.

If, while inside the loop still, we don't detect the correct input, we can use the else condition to notify the user they have made a mistake (invalid input), and allow the loop to run again. Remember, the only way we can escape this loop is to give a correct input and break out!

This is a 'better' method, in the sense that it doesn't violate the DRY principle.

So let's use it! But before we do, let's consider checking for a "y" or "n" input. How about we also check for a capital "Y" or "N" as well, just in case the user has made a 'valid' input, but left their caps lock on by mistake. That's 4 valid input strings to look for.

But rather than code...

if input1 correct or if input2 correct or if input3 correct or if input4 correct

...Python gives us a neat little shortcut here. We can use in () syntax to check for an element **in**a **list/sequence**.

if answer in ("input1", "input2", "input3", "input4")

As mentioned, this is much neater, and a nice way to 'list' **valid** inputs, whether they are strings, integers, floats, etc.

So let's put it all together.. if we use this second method for our validation, along with the in ()syntax, we end up with...

# 3. Build shopping list?

while True:

answer = input("Would you like a shopping list for this menu? Enter 'y' or 'n'... ")

if answer in ("y", "Y", "n", "N"):

break

print("Hmm. Sorry, I didn't catch that. Please try again...")

if answer in ("y", "Y"):

buildShoppingList()

else:

print()

print("You got it! Bye for now :)")

Now, by the time we 'go below' the while loop in our program, the answer variable can only contain a valid yes or no response. Our final conditional if/else code will separate yes from no, and of course, we need to check for both valid 'yes' strings now: "y" and "Y". Nice!

***Validation solved!***

Now you've seen this method, with the while True loop and break statement, keep it up your sleeve ;)

**Challenge 3: Make your code a little more modular.**

**In the Munch app MVP, we signed off using an else statement, which we wanted to run if the user chooses not to ask Munch to build a shopping list. We used the code:**

**else:**

**print()**

**print("You got it! Bye for now :)")**

**How about we make our end program code a little more modular, so that if we ever choose to give the user a different ending, we can work on that code separately?**

**- Can you move the two print statements into a new function?**

**- Can you call the new function if the else condition evaluates true?**

**You can name your third function anything that makes sense to you!**

**Good luck! :)**

**NB: Video solutions coming soon so stay tuned for those!**

There are three steps to solving this challenge: Creating a new function, moving the existing code into the new function, and calling the function at the correct point. Let's tackle them in order...

**Step One - Code a new function**

def endProgram():

As you can see, we called our's *endProgram*, but feel free to use your own name.

**Step 2 - Move the existing code into the function body**

Now that we've created the function, let's move the existing print statements into the body..

def endProgram():

    print()

    print("You got it! Bye for now :)")

**Step 3 - Call the new function**

if answer in ("y", "Y"): #<--- Note that we changed this code in challenge 2.

buildShoppingList()

else:

endProgram() # <--- Call your new function here

As you can see, we're calling our new *endProgram* function using the else condition in our conditional code. This code will now call the function if the user enters any input, other than 'y', and the closing print() dialogues will run.

Of course, now that our code is in the function, it's more modular. If we ever want to change the way the program behaves if the user chooses not to ask for a shopping list, we can easily locate the correct code to edit, and work on that specific module. Nice :)

***Solved!***