ITEC 1150 Week 7
Chapter 4
LISTS & MORE

Problem Areas From Last Week

- The try/except was something that many people were confused by, so let's look some code walk through it in Python Tutor.
- Another thing that caused problems: Remember that if a function returns something, you have to assign that to a variable.
- Finally, remember the general requirements in the lab slides—missing those was one of the biggest issues last week!

```
1.  def processing(number):
2.      total = 0
3.      for item in range(number):
4.          total += item
5.      return total
6.
7.  def main():
8.      while True:
9.          try:
10.             number = int(input("Enter a number: "))
11.             total = processing(number)
12.             print(f'The total of all numbers between 0 and
    {number} is {total}')
13.             break
14.         except:
15.             print("Only the digits 0-9 can be entered!")
16.     print("Thanks for using the program!")
17.
18. main()
```

Lists

Our first data structure!

- **Data Structures** are ways to organize multiple pieces of data.
- So far, we've only put single pieces of data into variables.
- **Lists** are one of the most common data structures and are present in nearly every programming language.

Lists Are Everywhere

- Think about real life and how many lists you have:
  - TODO lists
  - Class lists
  - Shopping lists
  - Contact lists
  - and so forth…
- Programs have uses for all the same sorts of things—you probably have an app for at least some of the things on the left, and they're stored as lists in those apps.

Coffee Sales Reimagined

- In the `coffee_sales` program from week 3 (chapter 1 part 1), we needed separate variables for the cost and cups sold of each drink.
- Suppose we had to add two new drink types (espresso and chai) to the program. That would be at least four more variables, four more input() statements, four more lines to add to the sales table—now think of what our program would look like if we had as many different drinks as the typical coffee shop!
- Now imagine we want to let the user add their own drink types in that program—we can't do that right now, because we have to make variables for each drink type before the program even runs!

Lists Can Solve Those Problems

```
1.  drink_types = []
2.  drink_prices = []
3.  drinks_sold = []
4.
5.  while True:
6.      drink_types.append(input('What is your drink type? '))
7.      drink_prices.append(float(input('What is the price? ')))
8.      drinks_sold.append(int(input('How many did you sell? ')))
9.      add_another = input("Do you want to add another drink? (y/n) ")
10.     if add_another != 'y':
11.         break
```

- The variables on the first three lines are being declared as empty lists.
- Each time through the loop, we add another drink type, price, and cups sold to the respective list.
- This is far fewer variables than our original `coffee_sales`, but can hold dozens (in actuality, hundreds of millions) of different drinks!

A List Is a Data Structure

- A list isn't a simple piece of data, rather it is a group of items with certain properties and methods.
- Because of this, a list is a **very** useful way to organize data in your program
- One of the most important benefits of a list: it gives you a way to save & return a series of data points that are created in a loop.
- Other structures: tuples (part of ch 4) and dictionaries (ch 5)
  - Though tuples are not the main focus of the class, we will try to recognize them when we see them. Think about a function definition or call – where are the parameters/ arguments kept?
- Also - arrays, hash tables, trees, sets, heaps, and many other structures exist to manage data (for future programming classes)

Syntax, Structure, & Usage

DETAILS ABOUT LISTS IN PYTHON

Lists – What They Look Like

- A **list** is declared with square brackets, and items in the list are separated with commas:

```
[] # Empty list
['Lake', 'Hennepin', 'Lyndale'] # A list declared with three strings
```

- A list can be put into a variable, just like a single piece of data can be put into a variable.

```
streets = ['Lake', 'Hennepin', 'Lyndale']
```

- You can refer to items within the list by their **index** (position number, which starts at 0).

```
print(streets[1]) #Prints 'Hennepin'
```

Lists Can Contain Any Data Type

Strings, Integers, Floats, and even other Lists (or any other data structure) can be put into lists:

```
1. villains = ['Voldemort', 'Darth Vader', 'Sauron']   # string data
2. quiz_scores = [10, 9, 10, 8]                        # int data
3. rainfall_amts = [0.8, 1.3, 1.1, 2.1, 2.6, 0.2, 0.6] # float data
4. mixed_items = ['Hello', 100, 5.55]                  # varying data types
5. list_of_lists = [[1,2,3], [4,5,6]]                  # a list of lists!
```

Checking If Something Is In A List

- The `in` keyword is very useful to see if an element is in a list:

```
1. villains = ['Catwoman', 'Darth Vader', 'Sauron', 'Lex Luthor', 'Magneto']
2. if 'Darth Vader' in villains:
3.     print('Darth Vader is in the list')
4. else:
5.     print('Darth Vader is NOT in the list')
6. if 'Voldemort' in villains:
7.     print('Voldemort is in the list')
8. else:
9.     print('Voldemort is NOT in the list')
```

Creating Lists, Adding, and Changing Data

- There are a variety of ways of creating lists and adding data. Some of these include:
  - We can create a list and add data in one step
  - We can create a list and add data whenever our program needs to
  - We can turn data into lists with functions
- Of course, we can also change the data in our list.

Initializing a List With Data

- As we've already seen, you can create a list and put data into it in one step:

```
1. streets = ['Lake', 'Hennepin', 'Lyndale']
```

- This is mostly useful if your program has **constant** data (something that is part of the program and should not change):

```
1. ALLOWABLE_GRADES = ['A', 'B', 'C', 'D', 'F', 'W', 'NC']
2.
3. grade = input("Enter the student's grade: ")
4. if not grade in ALLOWABLE_GRADES:
5.     print(f"{grade} is not a recognized grade. Please try again.")
```

By convention, variables that contain constant data are named with ALL_CAPS as a reminder that they should not be changed.

Creating And Adding To A List

- Much more often, we will start with an empty list and add data later.
- The `.append()` method will add an element to the end of the list:

```
1. villains = [] # initialize empty list
2. villains.append('Catwoman')
3. villains.append('Darth Vader')
4. villains.append('Sauron')
5. villains.append('Lex Luthor')
6. villains.append('Magneto')
7. print(villains)
```

- Output:

```
['Catwoman', 'Darth Vader', 'Sauron', 'Lex Luthor', 'Magneto']
```

- Notice that when we pass a list to the `print()` function, the output looks just like we'd type if we were creating a list with data already in it.

The `list()` Function

- The `list()` function tries to turn an argument into a list.
- The argument that is passed to `list()` must be **iterable**, meaning it has to be a sequence of something.
- `range()` returns a sequence of numbers, so we can turn it into a list.
- A `string` is a sequence of characters, so we can turn it into a list.
- A single number is not a sequence and will cause an error if you pass one to `list()`.

```
1.  numbers = list(range(1,10))
2.  print(numbers)
3.
4.  letters = list('Hennepin')
5.  print(letters)
6.
7.  empty_list = list()
8.  print(empty_list)
9.
10. numbers2 = list(7) # Error!
11. print(numbers2)
```

Example: Adding User Input To A List

```
1. print('Welcome to the Guest List Builder')
2. guests = []
3. while True:
4.     new_guest = input('Type name & enter (hit enter twice to quit): ')
5.     if new_guest == '':
6.         break
7.     else:
8.         guests.append(new_guest)
```

Changing (Reassigning) List Items

```
1. villains = ['Voldemort', 'Darth Vader', 'Sauron']
2. quiz_scores = [10, 9, 10, 8]
3. rainfall_amts = [0.8, 1.3, 1.1, 2.1, 2.6, 0.2, 0.6]
4.
5. villains[0] = 'Catwoman'
6. print(villains)
7. quiz_scores[1] = 10
8. print(quiz_scores)
9. rainfall_amts[4] = 1.2
10. print(rainfall_amts)
```

- Output:

```
['Catwoman', 'Darth Vader', 'Sauron']
[10, 10, 10, 8]
[0.8, 1.3, 1.1, 2.1, 1.2, 0.2, 0.6]
```

Accessing List Data

- We can access individual elements in a list, but more importantly, we can **iterate** over a list and work with each element in turn.

Processing Every Element In A List

- "Iteration" means to repeat a process. Lists and Ranges are 'iterable', meaning they're designed to allow us to repeat a process on each element.
- The ability to put many pieces of data into a list and then do something with each one is one of the key reasons we use lists.

```
1. print("Here are all the villains in our list:")
2. for villain in villains:
3.     print(villain)
```

- Output:

```
Here are all the villains
in our list:
Catwoman
Voldemort
Darth Vader
Sauron
Lex Luthor
Magneto
```

Alternate Method Using Index

- The `len()` function gets the length of the list which we can turn into a range.
- With a little formatting, we can add numbers to our list:

```
1. print("Here are all the villains in our list:")
2. for idx in range(len(villains)):
3.     number = f"{idx + 1}."
4.     print(f"{number:<3}{villains[idx]}")
```

- Output:

```
Here are all the villains
in our list:
1. Catwoman
2. Voldemort
3. Darth Vader
4. Sauron
5. Lex Luthor
6. Magneto
```

Accessing a Single List Item By Index

```
1. villains = ['Voldemort', 'Darth Vader', 'Sauron']
2. quiz_scores = [10, 9, 10, 8]
3. rainfall_amts = [0.8, 1.3, 1.1, 2.1, 2.6, 0.2, 0.6]
4.
5. print(villains[0])
6. print(quiz_scores[1])
7. print(rainfall_amts[4])
```

- It's important to know that if you use an index that doesn't exist, you will get an error!

- Output:

```
Voldemort
9
2.6
```

Example: Index Method With Parallel Lists

- If we have multiple lists of the same length, using the index method helps us access the same element of each list:

```
1. drink_types = ['Coffee', 'Tea', 'Cappuccino']
2. drink_prices = [4.99, 2.99, 6.49]
3. drinks_sold = [187, 42, 73]
4.
5. for idx in range(len(drink_types)):
6.     print(f"{drink_types[idx]:<15}${drink_prices[idx]:>6.2f}{drinks_sold[idx]:>6}")
```

- It's important that the lists are all the same length when you do this!

Removing Data From A List

- Obviously, there are times we'll want to remove things from our lists…

Pop – Removes From The List And Returns It

- `pop()` removes and returns the last item in the list if you don't provide an argument:

```
1. guests = ['Abdi', 'Becky', 'Cho']
2. print(guests)
3. uninvited = guests.pop()
4. print('Name removed: ', uninvited)
5. print('Guest list: ', guests)
```

- Passing an argument to `pop()` allows you to specify the index of the element:

```
1. print(guests)
2. nixed_second_guest = guests.pop(1)
3. print('Name removed: ', nixed_second_guest )
4. print('Guest list: ', guests)
```

Remove – Useful Method If You Know The Exact Data To Remove

- Perhaps we don't want Sauron to come to the party.
- The remove method will uninvite him, if he's there.
- To be safe, put all code like this in a conditional block:

```
1. if 'Sauron' in guests:
2.     guests.remove('Sauron')
3. else:
4.     print('No such guest.')
5. print('Guest list: ', guests)
```

Other Things We Can Do With Lists

- A list is more than just a bucket to put lots of data into.
- We can also do things like sort the list, add up all the numbers in a list, and more.

Sorting

- We can sort lists:

```
1. streets = ['11th','Lake','Grand']
2. streets.sort()
3. print(streets)
4.
5. grades = [98, 87, 91]
6. grades.sort()
7. print(grades)
```

- Strings sort using alphabetical rules, so `'199'` will come before `'20'`— something to be aware of.
- Numbers sort smallest to largest.
- Calling `sort()` on a list with mixed types (for example, strings and ints) will cause an error.
- We can reverse-sort a list like this:

```
1. grades.sort(reverse=True)
```

(We'll talk about the keyword in the arguments in coming weeks.)

Analyzing

```
1. rainfall_amts = [0.8, 1.3, 1.1, 2.1, 2.6, 0.2, 0.6]
2. rainfall_amts.sort()
3. print(rainfall_amts)
4. total_rain = sum(rainfall_amts) # Adds all the numbers in a list
5. avg_rain = round(sum(rainfall_amts) / len(rainfall_amts), 2)
6. min_rain = min(rainfall_amts) # Finds the minimum value
7. max_rain = max(rainfall_amts) # Finds the maximum value
8. print(f'Weekly rain facts: \nMin {min_rain}\tMax {max_rain}\tAvg {avg_rain}\tTotal {total_rain:.2f}')
```

```
[0.2, 0.6, 0.8, 1.1, 1.3, 2.1, 2.6]
Weekly rain facts:
Min 0.2 Max 2.6 Avg 1.24    Total 8.70
```

Use A Loop For Max Control Of Printing

```
1. # Use this code with the guest list from above.
2. # prints list items with other info & treats last list item differently – see
   below…
3. for index in range(len(guests)):
4.     if index < len(guests) -1:
5.         print(f'Guest #{index + 1} is {guests[index]}, ', end = "")
6.     else:
7.         print(f'and Guest #{index + 1} is {guests[index]}.')
```

The last item of any list is at the position `len(listname) - 1`.

`end = ""` prevents a `\n` from printing after each item.
We use the else for the last item to get different content.

Another Print Loop, Different Look...

```
1. # possibilities are endless
2. print('{:<30}{:>7}'.format('Guest Name','Ticket'))
3. for index in range(len(guests)):
4.     print('{:<30}{:>2}{:>5}'.format(guests[index],'#',index + 1001))
5. print('{:.<30}{:.>7}'.format('Total',len(guests)))
```

```
Ahmed, Bob, Carol, Dinh, Ella, Fahid
Guest Name                    Ticket
Ahmed                         # 1001
Bob                           # 1002
Carol                         # 1003
Dinh                          # 1004
Ella                          # 1005
Fahid                         # 1006
Total.............................6
```

Here the right-hand column is a calculation, but what if it was being printed from a second parallel list?
Hint: This will come up in lab

Using * And + With Lists

- Want a list of 100 zeros?

```
1. zeros = [0] * 100
2. print(zeros)
```

- Want to add all the items in one list to all the items in another list?

```
1. my_bands = ['Prince', 'Daft Punk', 'Led Zeppelin']
2. friends_bands = ['Lady Gaga', 'Rhianna', 'Beyonce']
3. roadtrip_playlist = my_bands + friends_bands
4. print(roadtrip_playlist)
```

Add Items w/ a Loop & Printing Shortcut

```
1. print('Welcome to the Guest List Builder')
2. guests = []
3. while True:
4.     new_guest = input('Type name & enter (hit enter twice to quit): ')
5.     if new_guest == '':
6.         break
7.     else:
8.         guests.append(new_guest)
9. print(guests) # compare results
10. print(*guests, sep = ", ")
```

This code is used to get data for the code on the following slides.

The notation:
`print(*listname, sep = ", ")`
is a shortcut to print a list without brackets & quotation marks, with a separator.

Copying A List

MORE TO THINK ABOUT THAN YOU MIGHT EXPECT

Copying A List's Data (Or Just A Slice)

- You can make a copy of part or all of a list by using the syntax below:

  `<list_name>[start_index : end_index]` ← `end_index` is *exclusive*, just like with a range.

- If you leave the start or end index out, Python will default to the beginning or ending index.

```
1. grades = [98, 87, 91, 100, 77]
2. grades_copy = grades[:]
3. print(grades_copy)
4. grades_copy = grades[1:3]
5. print(grades_copy)
6. grades_copy = grades[0:4]
7. print(grades_copy)
```

- Output:

```
[98, 87, 91, 100, 77]
[87, 91]
[98, 87, 91, 100]
```

Aliasing (Copy-By-Reference)

- Using the `[:]` syntax copies the data. However, when we pass a list to a function or assign the value of a variable containing a list to another variable, something different happens.
- When we pass `grades` to the `example_function` on line 5, it may not look like we're changing `grades`—it may look like the function is changing its own copy. However, when we print `grades`, you can see it changed.
- The same is true when we assign the value of `grades` to a new variable named `grades_copy`. When we change `grades_copy`, printing both shows us that the list changed in both places.
- If you want to make sure you don't accidentally change a list, make a copy of the *data* as shown on the previous slide.

```
1.  def example_function(list):
2.      list.pop()
3.
4.  grades = [98, 87, 91, 100, 77]
5.  example_function(grades)
6.  print(grades)
7.
8.  grades_copy = grades
9.  grades_copy.pop()
10.
11. print(grades)
12. print(grades_copy)
```

Real-World Example

- Suppose Amelia shares an online document (Word, Google Doc, whatever) with her classmate Bashiir.
- Bashiir now has a *reference* to the original document, and any changes Amelia or Bashiir make to the document will be seen by both people.
- If that's not what they want, Bashiir needs to save the data as a new document.

Aliasing vs Copying

- *Generally*, data structures can be aliased and simpler values (like strings, ints, etc.) cannot.
- For this class, you don't need to worry about it too much other than to understand what we've talked about tonight so you can avoid problems.
- If you're studying computer science, this will come up again in the future and you'll get a much deeper explanation.

Lists - summary

Lists are really useful

They are one of the most common data structures in Python (or most programming languages, for that matter)

Learn how to create them, copy them, add data, remove data, use them in a loop for calculations and printing

Programming majors need to understand copying vs. aliasing

NOTE: download, run and try to explain the file demo4_quiz_avg_list.py file – to be used in class discussion.