Introducing Lists

University of Wyoming COSC 1010

Adapted from: *Python Crash Course 3rd Ed* By Eric Matthes

Working with Lists

Working with Lists

- Last time we saw how to create a simple list
- we also learned to work with individual elements in a list
- This time we will learn how to loop through an entire list
- This can be down with just a few lines of code
- *Looping* allows you to take the same action, or set of actions, with every item in a list

Looping through an Entire List

- Often you will want to run through a list
- Performing the same task on each element
- Perhaps you want to sum all the numbers in a list manually
- Or display all titles from a list of headlines

Looping Through an Entire List

- When you want to do the same action for every item, you can use a `for` loop
- Suppose we had our list of cities from last time
- And this time you want to go through and print each city individually
- Sure the list is small, we could do it manually by every index
- But that is repetitive and tedious

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities[0])
print(cities[1])
print(cities[2])
print(cities[3])
```

```
Laramie
Casper
Jackson Hole
Cheyenne
```

Looping Through an Entire List

- There is a better way to do this
- Using a `for` loop avoids both the tedium and repetition
- It also works for long lists where doing it manually isn't an option
- Let's take a look at using a `for` loop

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]

for city in cities:
    print(city)
```

```
Laramie
Casper
Jackson Hole
Cheyenne
```

Looping Through an Entire List

- That code section starts by declaring a list as we saw last time
- Then it defines a `for` loop
- The for loop is of the form **`for`** ***noun*** **`in`** ***list name*** **`:`**
- Recall lists are pluralized. the noun used is typically the singular version of the list name

Looping Through an Entire List

- Python repeats the code inside the loop as many times aas needed
- Note the *indentation* in the code
- Code indented under the `for city in cities:` line is the code executed during the loop

A Closer Look at Looping

- Looping is one of the most common wats a computer automates repetitive tasks
- In the loop above we used it to print the city names
- The line `for city in cities:` tells python to take the first value in the list and then associate it with the variable city
- Each iteration through it re-associates city with the next value stored
- Until it reaches the end of the List

A Closer Look at Looping

- When you're using loops for the first time remember the set of steps is repeated for each item
- This means if you have millions of items in the list python will run through the loop millions of times
- Also remember when writing loops you can choose any variable name you want
- It is helpful to select something meaningful for the individual item
- Single adn plur names help you identify where you are

Doing More Work Within an for Loop

- You can do just about anything for each item in a loop
- We can build on the previous example by printing a message for each city
- Let's say that each is a city in Wyoming

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]

for city in cities:
    print(f"{city} is a city in Wyoming!")
```

```
Laramie is a city in Wyoming!
Casper is a city in Wyoming!
Jackson Hole is a city in Wyoming!
Cheyenne is a city in Wyoming!
```

Doing More Work Within an for Loop

- The difference from the previous loop is out inclusion of the *f-string*
- You can write as many lines as you want in a loop
- Every indented line following the line `for city in cities` is part of the loop
- Each indented line is executed once for each item in the list

Doing More Work Within an for Loop

- You can do as much or as little work on an item as you want
- In practice it is often helpful to do a lot of work on items
- So don't be afraid to do as much work as you need/want

Doing Something *After* a for Loop

- What happens once a loop has finished executing?
- Usually you want to summarize a block of output or more on to more work
- Any lines of code after the `for` loop that aren't indented will be executed
- For example after out cities print we can do a final message

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]

for city in cities:
    print(f"{city} is a city in Wyoming!")
print("Which one is your favorite?")
```

```
Laramie is a city in Wyoming!
Casper is a city in Wyoming!
Jackson Hole is a city in Wyoming!
Cheyenne is a city in Wyoming!
Which one is your favorite?
```

Doing Something *After* a for Loop

- When processing data using a `for` loop , adding a print after is a good way to summarize
- It also helps to signify that all the data has been gone through
- For example, if you're setting up a game ou can see that all the characters have been initialized
- You might then write some additional code after the loop to display a `play now` button

Avoiding Indentation Errors

- Python uses indentation to determine how a line, or a group of lines, is related to the rest of the program
- The earlier print line was part of the loop because it was *indented*
- Python's use of indentation makes code easy to read

- It uses whitespace to force you to write neatly formatted code
- This code has a clear visual structure

Avoiding Indentation Errors

- In longer programs there may several layers of indentation
- These indentation levels help gain a sense of program organization
- As you begin to write code that relies on proper indentation there are some errors that may arise
- Maybe over indenting, maybe forgetting to indent

Forgetting to Indent

- When writing a for loop you **always** need to indent the next line
- Failing to do so will result in an error

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]

for city in cities:
print(f"{city} is a city in Wyoming!")
```

```
 File "/tmp/ipykernel_1264/3736988856.py", line 4
   print(f"{city} is a city in Wyoming!")
   ^
IndentationError: expected an indented block after 'for' statement on line 3
```

Forgetting to Indent

- In that code snippet we *forgot* to indent the print statement
- As a result python gives us an *indentation error*
- This errors are typically easily resolved, by going back and indenting what is needed

Forgetting to Indent Additional Lines

- Sometimes your loop will run without errors
  - But still won't produce the desired result
- This can happen when you want to have several tasks happen within the loop
  - But fail to indent all but the first line

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
```

```
for city in cities:
    print(f"{city} is a city in Wyoming!")
print(f"{city} is neat!")
```

```
Laramie is a city in Wyoming!
Casper is a city in Wyoming!
Jackson Hole is a city in Wyoming!
Cheyenne is a city in Wyoming!
Cheyenne is neat!
```

Forgetting to Indent Additional Lines

- That second print call is supposed to be indented and executed for every city
- However, as it isn't indented it only prints for the final city in the loop, after the loop finishes
- This is a logical error, the syntax is valid but the code doesn't produce the desired result
- If you expect something to be executed for every item, but is only executed for one it may be an indentation error

Indenting Unnecessarily

- If you accidentally indent a line that doesn't need to be indented Python also gives an error

In [ ]:
```
message = "Hello Class!"
    print(message)
```

```
  File "/tmp/ipykernel_1264/730423226.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

Indenting Unnecessarily

- In the previous code the `print()` isn't part of a sub-block of code
- As a result it shouldn't (and can't) be indented
- You can avoid unexpected indentation errors by indenting only when you have specific reason to
- In the programs you write at this stage, the only lines that should be indented are the ones for loops

Indenting Unnecessarily After a Loop

- Much like you can fail to indent code meant to be in a loop
- You can also accidentally indent code that should occur after a loop
- This is another instance of a logical error
- Often the code will run, but not with the anticipated output

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]

for city in cities:
    print(f"{city} is a city in Wyoming!")

    print(f"What is your favorite Wyoming city?")
```

```
Laramie is a city in Wyoming!
What is your favorite Wyoming city?
Casper is a city in Wyoming!
What is your favorite Wyoming city?
Jackson Hole is a city in Wyoming!
What is your favorite Wyoming city?
Cheyenne is a city in Wyoming!
What is your favorite Wyoming city?
```

Forgetting the Colon

- The colon at the end of a `for` loop line is important
- It tells python to start interpreting the next line(s) as part of the loop
- If you forget the colon you will get an error

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]

for city in cities
    print(f"{city} is a city in Wyoming!")
```

```
  File "/tmp/ipykernel_1264/1029366817.py", line 3
    for city in cities
                      ^
SyntaxError: expected ':'
```

Forgetting the Colon

- Without the colon Python isn't sure what you're trying to do
- It may think you were writing more code for a more complex loop
- If the interpreter can find a possible fix it will suggest one
- Some errors are easy and obvious, others aren't

Making Numerical Lists

- Many reasons exist to store a set of numbers
- Fore example, keeping track of images in a grid
- In data visualization you'll almost always work with sets of numbers
  - Temperatures
  - Distances
  - Population sizes
  - Latitude and Longitude

Making Numerical Lists

- Lists are ideal for storing sets of numbers
- Python provides a variety of tools to help you work efficiently with lists and numbers
- Once you understand these tools your code will work well even with lists containing millions of items

Using the `range()` Function

- Python's `range()` function makes it easy to generate a series of numbers
- `range()` can be used to print a series of numbers
- Among other uses

In [ ]:
```
for value in range(1,5):
    print(value)
```

```
1
2
3
4
```

Using the `range()` Function

- Note the code in the previous snippet looks like it should print `1, 2, 3, 4, 5`
- However, it only prints `1-4`
- This is a result of the *off-by-one* nature of a lot of programming
- The `range()` function starts at the lower bound given and stops when it reaches the upper
- As it reaches the upper bound before the inner code runs, it is not included I If your output is different that expected when using `range()` try checking your bounds

Using `range()` to Make a List of Numbers

- If you want to make a list of numbers you can convert the results of `range()`
- They can be directly converted into a list with the `list()` function
- When you wrap `list()` around a call to the `range()` function it will output a list of numbers

In [ ]:
```
numbers = list(range(1,6))
print(numbers)
```

```
[1, 2, 3, 4, 5]
```

Using `range()` to Make a List of Numbers

- We can also tell `range()` to skip numbers
- If you give a third argument to `range()` Python uses that value as a step size
- You can use this to do things like generating lists of even (or odd) numbers

In [ ]:
```
even_numbers = list(range(2, 11, 2))

print(even_numbers)
```

```
[2, 4, 6, 8, 10]
```

Using `range()` to Make a List of Numbers

- In that code snippet the function starts with 2
- It then adds two at every increment
- It does so until it reaches the end of the range

Using `range()` to Make a List of Numbers

- You can create almost any set of numbers using `range()`
- Suppose you want to create a list of the first 10 square numbers
- We know how to make a list
- We know how to square numbers
- We know how to set a range

In [ ]:
```
squares = []
for value in range(1,11):
```

```
    square = value ** 2
    squares.append(square)
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Using `range()` to Make a List of Numbers

- We started with an empty list of called `squares`
- Then we tell Python to loop through the first 10 numbers
  - The first 10 squares logically must be part of the squares of 1-10
- Inside the loop we range our current `value` to the power of two
- We add our squared number to our `squares` list

Simple Statistics with a List of Numbers

- A few python functions are helpful when working with a list of numbers
- For example, it is easy to find the:
  - Minimum `min()`
  - Maximum `max()`
  - Sum `sum()`

In [ ]:
```
digits = list(range(1,10))
print(min(digits))
print(max(digits))
print(sum(digits))
```

```
1
9
45
```

List Comprehensions

- The approach described early for generating squares required three or four lines of code
- A *list comprehension* allows you to generate the same list in only a line of code
- A *list comprehension* combines the `for` loop and the creation of elements
- These are a more *advanced* level of python

In [ ]:
```
squares = [value**2 for value in range(1, 11)]
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

List Comprehensions

- To use this syntax you must first begin with a descriptive name for a list
- Next, you open a set of square brackets
- within those brackets you define the expression for the values you want to store
- Here the expression is `value**2`
- Then the loop is written, still within brackets

List Comprehensions

- The for loop in this example is `for value in range(1,11)`
- this feeds the values 1 through 10 into the expression `value**2`
- Notice, there was no colon after this `for` loop
- the result was the same as having the assignment within the loop codeblock

Working with Part of a list

- In our initial list lecture we learned how to access an individual item in a list
- Now we've been working our way through all elements of a list
- In python you can also work with a specific group of items in a list called a *slice*

Slicing a List

- To make a slice, you specify the index of the first and last elements you want to work with
- As with the `range()` function, python stops one item before the upper bound
- So, to get the first three items in a list you would request indices 0 through 3
- this would return elements 0,1,2

In [ ]:
```
states = [
    "Alabama", "Alaska", "Arizona", "Arkansas", "California",
    "Colorado", "Connecticut", "Delaware", "Florida", "Georgia",
    "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa",
    "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland",
    "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri",
    "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey",
    "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio",
    "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island", "South Carolina",
    "South Dakota", "Tennessee", "Texas", "Utah", "Vermont",
    "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"
```

```
]

print(states[0:3])
```

```
['Alabama', 'Alaska', 'Arizona']
```

Slicing a List

- The previous code prints the first three US states
- The output retains the structure of a list
- Any subset can be generated, so you can take elements from the center as well as the ends

In [ ]:
```
states = [
    "Alabama", "Alaska", "Arizona", "Arkansas", "California",
    "Colorado", "Connecticut", "Delaware", "Florida", "Georgia",
    "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa",
    "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland",
    "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri",
    "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey",
    "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio",
    "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island", "South Carolina",
    "South Dakota", "Tennessee", "Texas", "Utah", "Vermont",
    "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"
]

print(states[27:31])
```

```
['Nevada', 'New Hampshire', 'New Jersey', 'New Mexico']
```

Slicing a List

- If you omit the first bound Python assumes you want to start at the front
  - `states[:3]` is equivalent to `states[0:3]`
- A similar syntax works with the end of the list
- If you want to include all elements starting part way through to the end you can omit the second bound
  - `states [47:]`

In [ ]:
```
states = [
    "Alabama", "Alaska", "Arizona", "Arkansas", "California",
    "Colorado", "Connecticut", "Delaware", "Florida", "Georgia",
    "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa",
    "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland",
    "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri",
    "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey",
    "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio",
    "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island", "South Carolina",
    "South Dakota", "Tennessee", "Texas", "Utah", "Vermont",
```

```
    "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"
]

print(states[:3])
print(states[47:])
```

```
['Alabama', 'Alaska', 'Arizona']
['West Virginia', 'Wisconsin', 'Wyoming']
```

Slicing a List

- This syntax allows you to output all elements from any point in the list
- This applies regardless of list size
- Recall that a negative number allows us to access a certain distance from the end of the list
- You can use this same idea to access the final `x` amount of elements in a list
- So if you want the final three states: `states[-3:]`

In [ ]:
```
states = [
    "Alabama", "Alaska", "Arizona", "Arkansas", "California",
    "Colorado", "Connecticut", "Delaware", "Florida", "Georgia",
    "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa",
    "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland",
    "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri",
    "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey",
    "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio",
    "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island", "South Carolina",
    "South Dakota", "Tennessee", "Texas", "Utah", "Vermont",
    "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"
]

print(states[-3:])
```

```
['West Virginia', 'Wisconsin', 'Wyoming']
```

Looping Through a Slice

- You can use a slice in a `for` loop
- This lets you loop through only a subset of elements
- This works in the same way as a normal loop would
- The difference is that where we would normally only have the list name, we include the slice bounds

In [ ]:
```
states = [
    "Alabama", "Alaska", "Arizona", "Arkansas", "California",
    "Colorado", "Connecticut", "Delaware", "Florida", "Georgia",
    "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa",
    "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland",
```

```
    "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri",
    "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey",
    "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio",
    "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island", "South Carolina",
    "South Dakota", "Tennessee", "Texas", "Utah", "Vermont",
    "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"
]

for state in states[-4:]:
    print(state)
```

```
Washington
West Virginia
Wisconsin
Wyoming
```

Looping Through a Slice

- Slices are useful in a number of situations
- When you're working with data you can use slices to process the data in chunks
- Or if you're building web applications you can use slices to display information

Copying a List

- Often you may start with an existing list and may make a new list based on the first
- But how does copying a list in Python work?
- To copy a list you can make a slice that includes teh entire original list
- This is done by omitting the bounds, only including the colon
  - `new_states = states[:]`
- But why not `new_states = states[]` ?

Copying a List

- `new_states = states[]` does not make a true copy of the list
- It instead makes a second name for the same list
- Meaning any changes we made to `states` would be reflected in `new_states` and vice versa
- we are just associating a new variable with the same value

In [ ]:
```
states = [
    "Alabama", "Alaska", "Arizona", "Arkansas", "California",
    "Colorado", "Connecticut", "Delaware", "Florida", "Georgia",
    "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa",
    "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland",
    "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri",
```

```
    "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey",
    "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio",
    "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island", "South Carolina",
    "South Dakota", "Tennessee", "Texas", "Utah", "Vermont",
    "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"
]

new_states = states

del states[0]
print(new_states[0])
```

```
Alaska
```

Copying a List

- So to make a new list based of the original we want to use the slice method
- First we make our new variable `new_states`
- Then we assign the whole slice of `states` to it
- `new_states = states[:]`

In [ ]:
```
states = [
    "Alabama", "Alaska", "Arizona", "Arkansas", "California",
    "Colorado", "Connecticut", "Delaware", "Florida", "Georgia",
    "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa",
    "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland",
    "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri",
    "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey",
    "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio",
    "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island", "South Carolina",
    "South Dakota", "Tennessee", "Texas", "Utah", "Vermont",
    "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"
]

new_states = states[:]

del states[0]
print(states[0])
print(new_states[0])
```

```
Alaska
Alabama
```

Tuples

- Lists work well for storing collections of items
- These collections may change throughout the run of a program
- The ability to modify lists is a particularly important often
- Sometimes you may want to create a list of items that doesn't change

- *Tuples* allow you to do just that
- Python refers to items that cannot change as *immutable*, and *immutable* list is a *tuple*

Defining a Tuple

- A tuple looks just like a list
- The difference is that it uses parenthesis rather than square brackets
- Once a tuple is defined individual items can be accessed using their index
- That is the same as you would for a list
- Suppose you want to set dimensions for a rectangle that never change

In [ ]:
```
dimensions = (200,50)
print(dimensions[0])
print(dimensions[1])
print(dimensions)
```

```
200
50
(200, 50)
```

Defining a Tuple

- We defined a tuple dimensions
- it uses parenthesis rather than a square bracket
- Each item can be printed separately or together from the tuple
- But, tuples are supposed to be *immutable* what happens if we try to change one?

In [ ]:
```
dimensions = (200,50)
dimensions[1] = 200
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_1264/1314032110.py in <module>
      1 dimensions = (200,50)
----> 2 dimensions[1] = 200

TypeError: 'tuple' object does not support item assignment
```

Defining a Tuple

- We can try to assign a new value to the second element
- However, Python will kick back an error when a line tries to change it
- This is exactly what we want

Defining a Tuple - Notes

- Tuples are really defined by the commas
- the parenthesis are unneeded, but better for clarity
- You can define a Tuple with a single element by having a trailing comma
- `my_tuple = 1886,`

Looping Through all Values in a Tuple

- Looping through a tuple is easy
- It is done exactly in the same fashion as lists!

```
for dimension in dimensions:
    print(dimension)
```

Writing Over a Tuple

- Tuples can't be modified, as we saw
- They can however have a new value assigned to the variable holding them
- For example to change our `dimensions` tuple we can redefine a new tuple

In [ ]:
```
dimensions = (200,50)
print(dimensions)

dimensions = (50,50)
print(dimensions)
```

```
(200, 50)
(50, 50)
```

Writing Over a Tuple

- We have our initial tuple and can print it for viewing
- We then associate a new tuple with the variable `dimensions`
- Python has no qualms as reassignments are fine
- When you compare with lists, tuples are simple data structures
- Use them if you want to store a set of values that should not be changed