Working with Lists

BY DR. SABEEHUDDIN HASAN

Looping through Lists

- *Looping* allows you to take the same action, or set of actions, with every item in a list.
- You are able to work efficiently with lists of any length, including those with thousands or even millions of items
- **Looping Through an Entire List**
  - In a list of numbers, you might want to perform the same statistical operation on every element
  - You can use Python's `for` loop for doing the same action with every item in a list

Looping through Lists

- Say we have a list of names, and we want to print out each name in the list
- We could do this by retrieving each name from the list individually
- This can be cumbersome to do this with a long list of names
- We will have to change our code each time the length of list is changed
- Let's use a for loop to print out each name in a list of people

```
people = ['alice', 'david', 'carolina']
for person in people:
        print(person)
```

- The output is shown below
  - alice
  - david
  - carolina

Looping through Lists

- ***A Closer Look at Looping***
  - The set of steps is repeated once for each item in the list, no matter how many items are in the list
  - You can choose any name you want for the temporary variable that will be associated with each value in the list
  - Helpful to choose a meaningful name for every item
    - **for cat in cats:**
    - **for dog in dogs:**
    - **for item in list_of_items:**
  - Using singular and plural names can help differentiate between a single item and a list

Looping through Lists

- ***Doing More Work Within a for Loop***
  - See the following example
    - **people = ['alice', 'david', 'carolina']**
    - **for person in people:**
      - **print(f"{person.title()}, that was a great job!")**
  - The output is as follows
    - Alice, that was a great job!
    - David, that was a great job!
    - Carolina, that was a great job!
  - One can write as many lines of code as one likes in the for loop
  - Every indented line following the line **for person in people:** is considered *inside the loop*
  - Each indented line is executed once for each value in the list

Avoiding Indentation Errors

- Python uses indentation to determine how a line, or group of lines, is related to the rest of the program
- Python's use of indentation makes code very easy to read
- It uses whitespace to force you to write neatly formatted code with a clear visual structure
- It is necessary to watch for indentation errors
- People sometimes indent lines of code that don't need to be indented or forget to indent lines that need to be indented

Avoiding Indentation Errors

- ***Forgetting to Indent***
- Always indent the line after the for statement in a loop. If you forget, Python will remind you:

```
people = ['alice', 'david', 'carolina']
for person in people:
print(person)
```

- The call to print() should be indented, but it's not

```
print(person)
^
IndentationError: expected an indented block after 'for' statement on line 2
```

Avoiding Indentation Errors

- ***Forgetting to Indent Additional Lines***
  - Sometimes your loop will run without any errors but won't produce the expected result
  - This can happen when you're trying to do several tasks in a loop and forget to indent all of them

```
people = ['alice', 'david', 'carolina']
for person in people:
        print(f"{person.title()}, that was a great job!")
print(f“Have a good day, {person.title()}.\n")
```

  - The last line gets printed only once. This is a logical error
  - The last line should have been indented too under the for loop

Avoiding Indentation Errors

- ***Indenting Unnecessarily After the Loop***
  - If you accidentally indent code that should run after a loop has finished, that code will be repeated once for each item in the list
  - This is a logical error

```
people = ['alice', 'david', 'carolina']

for person in people:

        print(f"{person.title()}, that was a great job!")


        print("This is a great job!")
```

  - Since the last line is indented, so it will be printed for every person, whereas it should have been printed only once at the end

Avoiding Indentation Errors

- ***Forgetting the Colon***
  - The colon at the end of a for statement tells Python to interpret the next line as the start of a loop

```
people = ['alice', 'david', 'carolina']
for person in people
        print(person)
```

  - If you accidentally forget the colon 1, you'll get a syntax error because Python doesn't know what you are doing

```
for people in persons
                    ^
SyntaxError: expected ':'
```

  - Python doesn't know if you forgot the colon or you had to write a more complex loop

Making Numerical Lists

- Lists are ideal for storing sets of numbers, and Python provides a variety of tools to help you work efficiently with lists of numbers
- In data visualizations, one works with sets of numbers, such as temperatures, distances, population sizes, or latitude and longitude values etc.
- ***Using the range() Function***
  - Python's range() function makes it easy to generate a series of numbers

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

  - Although this code looks like it should print the numbers from 1 to 5, it doesn't print the number 5:

```
1
2
3
4
```

Making Numerical Lists

- In this example, range() prints only the numbers 1 through 4
- This is another result of the off-by-one behavior
- The range() function causes Python to start counting at the first value you give it, and it stops when it reaches the second value you provide and never prints 5
- To print the numbers from 1 to 5, you would use range(1, 6):

```
for value in range(1, 6):
    print(value)
```

- You can also pass range() only one argument, and it will start the sequence of numbers at 0
- For example, range(6) would return the numbers from 0 through 5

Making Numerical Lists

- ***Using range() to Make a List of Numbers***
  - If you want to make a list of numbers, you can convert the results of range() directly into a list using the list() function

    numbers = list(range(1, 6))

    print(numbers)

    [1, 2, 3, 4, 5]
  - We can also use the range() function to tell Python to skip numbers in a given range
  - Python uses the third argument to range(), as a step size when generating numbers

    even_numbers = list(range(2, 11, 2))

    print(even_numbers)

    [2, 4, 6, 8, 10]

Making Numerical Lists

- You can make a list of the first 10 square numbers

```
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]

Making Numerical Lists

- ***Simple Statistics with a List of Numbers***
  - A few Python functions are helpful when working with lists of numbers

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

Making Numerical Lists

- ***List Comprehensions***
  - A *list comprehension* allows you to generate this same list in just one line of code
  - A list comprehension combines the for loop and the creation of new elements into one line, and automatically appends each new element

    squares = [value2 for value in range(1, 11)]**

    print(squares)

    [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
  - In this example the expression is value**2, which raises the value to the second power
  - The for loop generates the numbers you want to feed into the expression

Working with Part of a List

- ***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 second index you specify

    players = ['charles', 'martina', 'michael', 'florence', 'eli']

    print(players[0:3])
  - This code prints a slice of the list and includes first three players in the list

    ['charles', 'martina', 'michael']

Working with Part of a List

- You can generate any subset of a list

  players = ['charles', 'martina', 'michael', 'florence', 'eli']

  print(players[1:4])

- This time the slice starts with 'martina' and ends with 'florence':

  ['martina', 'michael', 'florence']

- If you omit the first index in a slice, Python automatically starts your slice at the beginning of the list:

  players = ['charles', 'martina', 'michael', 'florence', 'eli']

  print(players[:4])

- Without a starting index, Python starts at the beginning of the list:

  ['charles', 'martina', 'michael', 'florence']

Working with Part of a List

- If you omit the second index in a slice, Python automatically starts your slice at the end of the list:

  players = ['charles', 'martina', 'michael', 'florence', 'eli']

  print(players[2:])

- Python returns all items from the third item through the end of the list:

  ['michael', 'florence', 'eli']

- This syntax allows you to output all of the elements from any point in your list to the end, regardless of the length of the list
- If you want to output the last three players on the roster, we can use the slice players[-3:]

  players = ['charles', 'martina', 'michael', 'florence', 'eli']

  print(players[-3:])

Working with Part of a List

- ***Looping Through a Slice***
  - You can use a slice in a for loop if you want to loop through a subset of the elements in a list

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print("Here are the first three players on my team:")
for player in players[:3]:
        print(player.title())
```

  - Python loops through only the first three names

```
Here are the first three players on my team:
Charles
Martina
Michael
```

Working with Part of a List

- ***Copying a List***
  - To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index ([:])

```
my_foods = ['pizza', 'falafel', 'carrot cake']
friend_foods = my_foods[:]
print("My favorite foods are:")
print(my_foods)
print("My friend's favorite foods are:")
print(friend_foods)
```

  - We make a copy of my_foods and assign it to friend_foods

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake']
My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']
```

Working with Part of a List

- To prove we actually created two lists, look at the following code

```
my_foods = ['pizza', 'falafel', 'carrot cake']
friend_foods = my_foods[:]
my_foods.append('cannoli')
friend_foods.append('ice cream')
print("My favorite foods are:")
print(my_foods)
print("My friend's favorite foods are:")
print(friend_foods)
```

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli']
My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'ice cream']
```

Working with Part of a List

- If we had simply set friend_foods equal to my_foods, we would not produce two separate lists

```
my_foods = ['pizza', 'falafel', 'carrot cake']


# This doesn't work:
friend_foods = my_foods


my_foods.append('cannoli')
friend_foods.append('ice cream')
print("My favorite foods are:")
print(my_foods)
print("\nMy friend's favorite foods are:")
print(friend_foods)
```

Working with Part of a List

- Instead of assigning a copy of my_foods to friend_foods, we set friend_foods equal to my_foods

  My favorite foods are:

  ['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

  My friend's favorite foods are:

  ['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

- The output shows that both lists are the same now, which is not what we wanted

Tuples

- Lists work well for storing collections of items that can change throughout the life of a program
- However, sometimes you'll want to create a list of items that cannot change and tuples allow you to do just that
- Python refers to values that cannot change as *immutable*, and an immutable list is called a *tuple*

Tuples

- ***Defining a Tuple***
  - A tuple looks just like a list, except you use parentheses instead of square brackets
    - **dimensions = (200, 50)**
    - **print(dimensions[0])**
    - **print(dimensions[1])**
    - 200
    - 50

Tuples

- If we try to change one of the items in the tuple dimensions as given below:

  dimensions = (200, 50)

  dimensions[0] = 250

- This will generate the following error

  TypeError: 'tuple' object does not support item assignment

- *Tuples are technically defined by the presence of a comma; the parentheses make them*
- *look neater and more readable.*
- *If you want to define a tuple with one element, you need to include a trailing comma:*

  my_tuple = (3,)

Tuples

- ***Looping Through All Values in a Tuple***
  - You can loop over all the values in a tuple using a for loop, just as with a list:

```
dimensions = (200, 50)
for dimension in dimensions:
        print(dimension)
```

  - Python returns all the elements in the tuple, just as it would for a list:

```
200
50
```

Tuples

- ***Writing Over a Tuple***
  - Although you can't modify a tuple, you can assign a new value to a variable that represents a tuple

```
dimensions = (200, 50)
print("Original dimensions:")
for dimension in dimensions:
        print(dimension)
dimensions = (400, 100)
print("\nModified dimensions:")
for dimension in dimensions:
        print(dimension)
```

Tuples

- Python doesn't raise any errors this time, because reassigning a variable is valid

  Original dimensions:

  200

  50

  Modified dimensions:

  400

  100

- When compared with lists, tuples are simple data structures.
- Use them when you want to store a set of values that should not be changed throughout the life of a program.

Tuples

- Converting a list to a tuple and vice versa

```
number_list = [1,2,3,4,5]
tup = tuple(number_list)
print (tup)
lst = list(tup)
print (lst)
```