Set 13: Loops Part 2

Skill 13.1: Control loops with the *break* command

Skill 13.1 Concepts

Loops in Python are very versatile. Python provides a set of control statements that we can use to get even more control out of our loops.

Let's take a look at a common scenario that we may encounter to see a use case for *loop control statements*.

Consider the following list,

```
items_on_sale = ["blue shirt", "striped socks", "knit dress", "red headband",
"dinosaur onesie"]
```

It's often the case that we want to search a list to check if a specific value exists. What does our loop look like if we want to search for the value of "knit dress" and print out "Found it" if it did exist?

It would look something like this,

Code	Output

`for item in items_on_sale:`
`  print(item)`
`  if item == "knit dress":`
`  print("Found it")`
	blue shirt
	striped socks
	knit dress
	Found it
	red headband
	dinosaur onesie

This code goes through each item in *items_on_sale* and checks for a match. This is all fine and dandy but what's the downside?

Once "knit_dress" is found in the list *items_on_sale*, we don't need to go through the rest of the *items_on_sale* list. Unfortunately, our loop will keep running until we reach the end of the list.

Since it's only 5 elements long, iterating through the entire list is not a big deal in this case but what if *items_on_sale* had 1000 items? What if it had 100,000 items? This would be a huge waste of time for our program!

Thankfully you can stop iteration from inside the loop by using *break* loop control statement. When the program hits a break statement it immediately terminates a loop. For example,

Code

```
items_on_sale = ["blue shirt", "striped socks", "knit dress", "red headband",
"dinosaur onesie"]
print("Checking the sale list!")
for item in items_on_sale:
  print(item)
  if item == "knit dress":
    break
print("End of search!")
```

When the break statement is reached, the loop terminates

Output

```
Checking the sale list!
blue shirt
striped socks
knit dress
End of search!
```

Skill 13.1 Exercise

Skill 13.2: Control loops with the continue statement

Skill 13.2 Concepts

While the break control statement will come in handy, there are other situations where we don't want to end the loop entirely. What if we only want to skip the current iteration of the loop?

For example, what if we want to print out all of the numbers in a list, but only if they are positive integers. We can use another common loop control statement called continue.

Code

```
big_number_list = [1, 2, -1, 4, -5, 5, 2, -9]
for i in big_number_list:
  if i <= 0:
    continue
  print(i)
```

When the continue statement is reached, the loop returns to the top

Output

```
1
2
4
5
2
```

Notice a few things,

1. Similar to when we were using the *break* control statement, our *continue* control statement is usually paired with some form of a conditional (*if*/*elif*/*else*).
2. When our loop first encountered an element (-1) that met the conditions of the *if* statement, it checked the code inside the block and saw the *continue*. When the loop then encounters a *continue* statement it immediately skips the current iteration and moves onto the next element in the list (4).
3. The output of the list only printed positive integers in the list because every time our loop entered the *if* statement and saw the *continue* statement it simply moved to the next iteration of the list and thus never reached the print statement.

Skill 13.2 Exercise

Skill 13.3: Apply nested loops to iterate over two-dimensional lists

Skill 13.3 Concepts

Loops can be nested in Python, as they can with other programming languages. We will find certain situations that require nested loops.

Suppose we are in charge of a science class, that is split into three project teams,

```
project_teams = [["Ava", "Samantha", "James"], ["Lucille", "Zed"], ["Edgar", "Gabriel"]]
```

Using a for or while loop can be useful here to get each team,

Code	Output
`for team in project_teams:` `  print(team)`	['Ava', 'Samantha', 'James'] ['Lucille', 'Zed'] ['Edgar', 'Gabriel']

But what if we wanted to print each individual student? In this case, we would actually need to *nest* our loops to be able to loop through each sub-list. Here is what it would look like,

Code	Output

`# Loop through each sublist`
`for team in project_teams:`
`  # Loop elements in each sublist`
`  for student in team:`
`  print(student)`
	Ava
	Samantha
	James
	Lucille
	Zed
	Edgar
	Gabriel

Skill 13.3 Exercise 1

Skill 13.4: Apply list comprehensions

Skill 13.4 Concepts

Python prides itself on allowing programmers to write clean and elegant code. Here, we are going to examine a way we can write elegant loops in our programs using *list comprehensions*.

To start, let's say we had a list of integers and wanted to create a list where each element is doubled. We could accomplish this using a for loop and a new list called doubled,

Code	Output

`numbers = [2, -1, 79, 33, -45]`
`doubled = []`
`for number in numbers:`
`  doubled.append(number * 2)`
`print(doubled)`
	[4, -2, 158, 66, -90]

Let's see how we can use the power of list comprehensions to solve this problem in one line. Here is our same problem but now written as a list comprehension,

Code

```
numbers = [2, -1, 79, 33, -45]
doubled = [num * 2 for num in numbers]
print(doubled)
```

The expression, num * 2 is applied to each num in the list, numbers.

Output

```
[4, -2, 158, 66, -90]
```

The above example can be broken down as follows,

```
new_list = [<expression> for <element> in <collection>]
```

1. Takes an element in the list numbers
2. Assigns that element to a variable called num (our <element>)
3. Applies the <expression> on the element stored in num and adds the result to a new list called doubled
4. Repeats steps 1-3 for every other element in the numbers list (our <collection>)

Skill 13.4 Exercise 1

Skill 13.5: Apply conditionals in list comprehensions

Skill 13.5 Concepts

List Comprehensions are very flexible. We even can expand our examples to incorporate conditional logic. Suppose we wanted to double only our negative numbers from our previous numbers list.

We will start by using a for loop and a list *only_negative_doubled*:

Code

```
numbers = [2, -1, 79, 33, -45]
only_negative_doubled = []


for num in numbers:
  if num < 0:
    only_negative_doubled.append(num * 2)


print(only_negative_doubled)
```

Output

```
[-2, -90]
```

Now, here is what our code would look like using a list comprehension,

Code

```
numbers = [2, -1, 79, 33, -45]
negative_doubled = [num * 2 for num in numbers if num < 0]
print(negative_doubled)
```

Output

```
[-2, -90]
```

In our *negative_doubled* example, our list comprehension:

1. Takes an element in the list *numbers*.
2. Assigns that element to a variable called num.
3. Checks if the condition *num* < 0 is met by the element stored in *num*. If so, it goes to step 4, otherwise it skips it and goes to the next element in the list.
4. Applies the expression *num * 2* on the element stored in *num* and adds the result to a new list called *negative_doubled*
5. Repeats steps 1-3 (and sometimes 4) for each remaining element in the *numbers* list.

We can also use *else-if* conditions directly in our comprehensions. For example, let's say we wanted to double every negative number but triple all positive numbers. Here is what our code might look like,

Code

`numbers = [2, -1, 79, 33, -45]`
`doubled = [num * 2 if num < 0 else num * 3 for num in numbers ]`
`print(doubled)`

Output
[6, -2, 237, 99, -90]

NOTE: This is a bit different than our previous comprehension since the conditional if *num < 0 else num * 3* comes after the expression *num * 2* but before our for keyword. The placement of the conditional expression within the comprehension is dependent on whether or not an *else* clause is used. When an *if* statement is used without *else*, the conditional must go after *for <element> in <collection>.* If the conditional expression includes an *else* clause, the conditional must go before *for.* Attempting to write the expressions in any other order will result in a SyntaxError.

Here are a few list comprehensions in a single block. Take a moment to compare how the syntax must change depending on whether or not an *else* clause is included:

```
numbers = [2, -1, 79, 33, -45]


no_if   = [num * 2 for num in numbers]
if_only = [num * 2 for num in numbers if num < 0]
if_else = [num * 2 if num < 0 else num * 3 for num in numbers]
```

Skill 13.5 Exercise 1