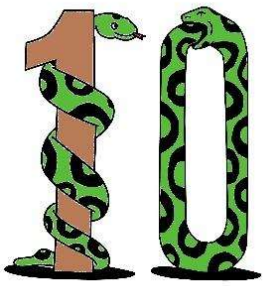

for Loop and List Processing

A *Pythonic* Way of Looping Through a List

To access all the items in a list, a Python programmer need not index the first item of the list and repeatedly increment the index to the end of the list as was coded in the `while` loop example in Chapter 9. That way of iterating through a list is a common method used with other programming languages. But, because a Python list itself is *iterable*, the items of a list can be accessed sequentially in a `for` loop without the use of list indexes. This *Pythonic* style applies to other iterable Python data types.

> ***The concept of "iterable" is important.***
>
> The `for` loop can step through an iterable data type's sequence of elements.
>
> ```
> for variable_name in iterable:
>     #block of code to execute
> ```

for Loop

The `for` loop allows a block of code to repeat a designated number of times. The number of

times is determined by stepping through an *iterable* data type.

The example below steps through all the items of a list. With each iteration of the `for` loop, `x` sequentially represents the current list item and gets printed.

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

for x in ex_list:
    print(x)
```

FIFO List Processing

FIFO, or *First In First Out*, is like a line of people where each person peacefully waits for his or her turn for a drink at the water fountain. This type of line is called a *queue*. The first person in line gets to take a drink first, the second in line goes next, and so on.

Department of Motor Vehicles FIFO Queue

Patrons waiting to see a clerk at the Department of Motor Vehicles are languishing in a queue waiting for ticket numbers 80 through 85 to be called. Thankfully, the Python interpreter will process this queue faster than the DMV. (Be ready to revisit this program soon.)

DMV_queue.py

```
ticket_numbers = [80,81,82,83,84,85]
for x in ticket_numbers:
    print("Ticket number " + str(x) + " has been called.")
```

LIFO List Processing

LIFO, or *Last In First Out*, processes the last item to get in line first, then the second to last, and so on. This kind of seemingly unfair line is called a *stack*. A common analogy for LIFO processing is a stack of pancakes. The last pancake cooked is placed on the top of the pancake stack and gets eaten first. (FIFO pertains to queues and LIFO pertains to stacks.)

The Laundry Stack

The person who puts his laundry on the pile last gets his laundry back first. The person who puts his laundry in the laundry pile first must wait the longest to get his laundry done because his laundry is at the bottom of the pile.

laundry_stack.py

```
import time

laundry_stack = []

for x in range(4):
    name = input("Enter a name: ")
    laundry_stack.append(name)
    print(name + " has laundry added to the stack.\n")

time.sleep(1)
print(laundry_stack[0] + " has run out of clean underwear!\n")

for x in range(4):
    time.sleep(1)
    print(laundry_stack.pop() + "'s laundry is done.")
```

The `time` module is imported. An empty list is assigned to `laundry_stack`. The first `for` loop iterates four times. Each time the loop iterates it receives a person's name via the `input` function, appends the name to the list, and prints that the person's laundry has been added to the stack. The `time` module's `sleep` method allows a programmer to delay execution of the program. When making the `time.sleep` call, the number of seconds to delay is included as an argument. When the delay finishes, the execution of the program resumes, and a message about the first person's dire laundry situation gets printed.

The second `for` loop begins. A delay is added to give the effect of people waiting for laundry to be finished. The last person in the list gets his laundry first. The `pop` method removes his name, and a message prints indicating that his laundry is done. The loop proceeds in LIFO order. After the `for` loop finishes processing every item in the `laundry_stack` list, the list is empty.

`laundry_stack`

Looping Through a String

Strings are also *iterable*.

```
word = "PYTHON"

for x in word:
    print(x)
```

With each iteration of the `for` loop, the value of `x` takes on the corresponding character in the string. Each character represented by `word` will print successively on a different line.

range Function

The `range` function returns a special, *iterable* data type called range that represents a sequence of integers.

```
range(10)
type(range(10))
```

Using the range Function with for

The `for` loop iterates through a `range` function's returned value. The number of times the `for` loop will iterate is determined by the argument(s) sent to the `range` function. When only one argument is sent to the `range` function, the loop iterates the number of times indicated by the argument.

In the next example, `x` is created in the `for` statement and followed by the keyword `in`. The `range` function call's argument is `10`, which returns a range data type containing the iterable sequence of integers `0` to `9`. At the end of the first line is a colon to denote the repeated block of code. The `for` loop will iterate and run its block of code ten times. The indented code repeats as long as `x` stays within the acceptable range.

Notice, the `print` statement executes ten times, but by default the variable `x` starts its count at `0` and ends at `9`. Displaying `x` on the last line reveals that `x` stayed at `9` and did not increment to `10` before exiting the `for` loop.

```
for x in range(10):
    print(x)

x
```

In this next example, the `range` function is sent two arguments. The first is the starting number for the count. The second is the ending number. The `range` function stops its count right before the second argument. Shifting the `for` loop's starting and stopping values over by one results in the integers `1` to `10` being printed.

```
for x in range(1,11):
    print(x)
```

Revisiting the DMV_queue.py program earlier in the chapter, now modify the code to reduce the amount of typing necessary by using the `range` function. Had the earlier program included a really long list of numbers, there would have been a lot of typing!

DMV_queue.py

```
ticket_numbers = range(80,86)
for x in ticket_numbers:
    print("Ticket number " + str(x) + " has been called.")
```

In the previous example, by default, the `for` loop incremented `x` by one with each pass of the loop. The below code includes a third argument to specify how much to increment `x` with each pass of the `for` loop.

```
for x in range(2,11,2):
    print(str(x) + " is even.")

for x in range(1,11,2):
    print(str(x) + " is odd.")
```

Decrementing with range Function and for Loop

When counting backwards *or decrementing*, the first argument to the `range` function is the high number, the second argument is the low number to stop right before, and the third argument is negative. Here, the range starts its count at 10 and stops just before zero.

```
for x in range(10,0,-1):
    print(x)
```

list Function

The `list` function couples nicely with the `range` function. When the `range` function call is sent as an argument to the `list` function, the `list` function returns a list of each value generated by the `range` call. The values are assigned to the list in order.

```
ex_list = list(range(1,101))
ex_list
ex_list[0]
ex_list[-1]
```

Error to Look Out For

Be careful if the looped code changes the size of a list. In the following code, the `for` loop is attempting to print and pop off all the items in `ex_list`. Notice what goes wrong.

```
ex_list = list(range(0,10))

for x in ex_list:
    print(ex_list.pop())

ex_list
```

While the `for` loop works its way from the beginning of `ex_list`, items pop off the end of `ex_list`. When `5` gets printed, the `for` loop stops because there are no more items left on the list because of the pops.

reversed Function

One possible solution to the error above is to use the `reversed` function. A `for` loop normally

iterates from the beginning of a list to the end of the list. The `reversed` function allows the iteration to begin at the last item of the list and work backwards. The `reversed` function does not mutate `ex_list`.

```
ex_list = list(range(0,10))

for x in reversed(ex_list):
    print(ex_list.pop())

ex_list
```

Compute

In the laundry_stack.py program earlier in the chapter, the `laundry_stack.pop()` call was executed in the block of the second `for` loop. Why did the `for` loop avoid the error of terminating too early?

---

break out of a Loop

Just like the `while` loop, a programmer can break out of a `for` loop early.

Quit as Soon as a Good Idea is Found

good_idea.py

```
ideas = ["bad","bad","bad","good","bad"]

for x in ideas:
    print(x)
    if x == "good":
        break
```

The last item on the above list does not get printed.

continue

The keyword `continue` enables the program to immediately continue to the next iteration of the `for` loop without executing the rest of that iteration's block of code.

Skip the Peanuts

Buy everything on the list but peanuts. There is a food allergy to avoid.

no_peanuts.py

```
groceries = ["milk", "eggs", "peanuts", "coffee"]

for x in groceries:
    if x == "peanuts":
        continue
    print("Please buy " + x + ".")
```

Compute Solution

The `for` loop avoided the error of terminating too early, because the number of iterations was not linked to the length of the `laundry_stack` list. Instead, the number of iterations was established as `for x in range(4):` which causes the `for` loop to execute four times regardless of the length of the `laundry_stack` list.

Chapter 10 Practice

Random Addition Problems Generator

Create a program called addition_drill.py that generates 10 random addition facts. The addends should be random integers in the range of 1 through 10. Have the user enter the sum for each problem, but don't generate the next problem until the user has answered the current problem correctly. The program must include a `for` loop.

Notice that missed problems are repeated.

Possible input and output:

```
5 + 7 = 11
5 + 7 = 12
6 + 2 = 8
3 + 4 = 7
8 + 6 = 13
8 + 6 = 14
4 + 8 = 12
5 + 3 = 8
7 + 10 = 17
2 + 5 = 7
9 + 8 = 17
1 + 1 = 2
```