CptS 111, Spring 2023
Lect. #14, Mar. 6, 2023
Class Notes

Today's agenda:

1. The idea of repeating commands
2. `for` -loops
3. The `range()` function
4. Counting `for` -loops
5. Counting `for` -loop: special case

Ch. 6

Loops

1. The Idea of Repeating Commands
Suppose we want to create a list of 5 integers that are entered by the user. We could use the following code:

```
integers = []
int1 = int(input('Enter integer #1: '))
int2 = int(input('Enter integer #2: '))
int3 = int(input('Enter integer #3: '))
int4 = int(input('Enter integer #4: '))
int5 = int(input('Enter integer #5: '))
integers.append(int1)
integers.append(int2)
integers.append(int3)
integers.append(int4)
integers.append(int5)
print(integers)
```

Notice that there's a lot of repetition in this code. It seems as though there should be a more efficient way of writing this code. In fact, often when we're programming, we want to repeat a series of commands, and we do this using loops. We'll study two different kinds of loops, `for` -loops and `while` -loops, and when we should use each kind.

2. `for` -loops

We'll consider two variations of `for` -loops, what I term counting `for` -loops and iterating `for` -loops. They both actually iterate, but it's helpful to think of two different constructs.

In this lecture, we'll cover counting `for` -loops, and on Wednesday we'll cover iterating `for` -loops, but before we begin our discussion of counting `for` -loops, we need to cover the `range()` function.

3. The `range()` function

The `range()` function is a special type of function which can be used to create a sequence of integers, e.g., for a list or tuple, but it's **commonly used in a header of a counting `for` -loop to create a sequence of integers that can be used for counting.**

The `range()` function can have 1, 2, or 3 arguments. If it has only one argument, it's the stop value. If it has two arguments, they're the start and stop values. If it has three arguments, they're the start, stop, and increment values.

```
range(stop):              Starts at 0, ends 1 before stop
range(start, stop):       Starts at start, ends 1 before stop
range(start, stop, inc):  Starts at start, increments by inc, and
                          ends 1 before stop
```

You can't use the range function alone because it won't return anything.

In [1]:
```
# When used alone, range() doesn't show us the sequence of integers it
# produces; it shows the arguments

range(7)
```

Out[1]:
```
range(0, 7)
```

Instead, to see the sequence of integers the `range()` function creates we can nest the `list()` or `tuple()` functions with it. The easiest way to understand the `range()` function is probably to see a bunch of examples.

In [2]:
```
# Use list() function to see sequence of integers produced by range()

list(range(7)) # range() produces a sequence; list() shows them as a list
```

Out[2]:
```
[0, 1, 2, 3, 4, 5, 6]
```

Notice that while the list of integers ends with `6` , there are seven integers. Thus, `range(7)` creates 7 integers. In fact, sometimes it's convenient to generate a list of integers using `range()` .

In [3]:
```
# Use start and stop values to create a list of integers

integers = list(range(1, 11))
print(integers)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

This is much faster than typing: `integers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` . Python aims to please! But let's continue with the examples.

In [4]:
```
# Use of start, stop, and increment values
# Sequence of integers starts at 1, increments by 3, and ends before 7

list(range(1, 7, 3)) # start+inc, start+inc+inc, start+inc+inc+inc, ...
```

Out[4]:
```
[1, 4]
```

Notice in this example that the first integer is 1, the second integer is 1 + 3 = 4, but because 1 + 3 + 3 = 7 and the stop value of 7 can't be one of the integers in the sequence, the sequence stops at `4`. If we want to include `7`, we have to use a stop value of `8`.

In [5]:
```
# Sequence of integers starts at 1, increments by 3, and ends with 7
# We must use 8 as our stop value for 7 to be in the list of integers

list(range(1, 8, 3))
```

Out[5]:
```
[1, 4, 7]
```

We can also use negative values for start, stop, and inc. The same rules apply.

In [6]:
```
# Using negative values for start, stop, and increment
# Sequence of integers starts at -1, decrements by -1, and ends before -11

list(range(-1, -11, -1))
```

Out[6]:
```
[-1, -2, -3, -4, -5, -6, -7, -8, -9, -10]
```

We can combine positive and negative values.

In [7]:
```
# Combining positive and negative arguments
# Sequence starts at 5, decrements by -1, and ends before -6

list(range(5, -6, -1))
```

Out[7]:
```
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5]
```

The default increment value is always 1.

In [8]:
```
# Sequence starts at -5, increments by 1, and stops before 6

list(range(-5, 6))
```

Out[8]:
```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
```

In addition, because the `range()` function only requires integer values, anything evaluating to an integer works as an argument.

In [9]:
```
# We can use math operations as long as they result in integers
# Sequence starts with 1+1=2, increments by 3-2=1, and stops before 2+2=4

list(range(1+1, 2+2, 3-2))  # == list(range(2, 4, 1))
```

Out[9]:
```
[2, 3]
```

We won't use such complex arguments for our counting `for` -loops. We'll usually be limited to simple arguments such as `range(5)` . However, we *will* use the following type of argument often as well.

In [12]:
```
# This is very useful and important
# Use of list length as an argument for the range() function

nums = [11, 17, 5, 23, 13, 7, 19]
print(len(nums))
list(range(len(nums))) # == list(range(7))
```

```
7
```

Out[12]:
```
[0, 1, 2, 3, 4, 5, 6]
```

In the example above, `len(nums)=7` . Thus, Python uses 7 as the argument of the `range()` function. Notice that the actual elements of `nums` is totally unrelated to the `range()` function. We only care about the length of `nums` .

Now we're ready to learn about counting `for` -loops!

4. Counting `for` -loops

Template for a counting `for` -loop:

```
for <counter> in range(<integer>):  # for-loop header
    <loop_body>
```

In the `for` -loop header, `<counter>` is the **loop variable**. It takes on the values of the sequence of integers created by the `range()` function. The arguments of the `range()` function can be like the ones shown in the examples above.

Importantly, counting `for` -loops ALWAYS use the `range()` function in their headers! Let's first consider a very simple example of a counting `for` -loop.

In [10]:
```
# Simple for-loop example that prints the integers 1 through 10

for i in range(10):
    print(i+1, end=' ')
```

```
1 2 3 4 5 6 7 8 9 10
```

Why did we use `i+1` in the `print()` argument? Why did we use `end=' '` ?

Next, let's look at an example using a list.

In [11]:
```
# Use of list length (len(goodies)) to generate sequence of integers

goodies = ['cookies', 'croissants', 'coffee ice cream', 'scones', 'brownies
for i in range(len(goodies)):
    print(f'I love {goodies[i]}!')
```

```
I love cookies!
I love croissants!
I love coffee ice cream!
I love scones!
I love brownies!
```

Next, look at the example we considered at the beginning of this lecture. We prompted a user for an integer 5 times and appended each entry to a list.

In [13]:
```
# Much more efficient way of prompting for integers!

integers = []
number_ints = int(input('Enter the number of integers you want to create: '
for i in range(number_ints):
    integer = int(input(f'Enter integer #{i+1}: '))
    integers.append(integer)
print(integers)
```

```
Enter the number of integers you want to create: 5
Enter integer #1: 2
Enter integer #2: 3
Enter integer #3: 5
Enter integer #4: 7
Enter integer #5: 11
[2, 3, 5, 7, 11]
```

Not only is this more efficient and easier to read than if we were to write five different integer prompts and five different append statements, but we're also less likely to make a typing mistake because we reduced our code from 12 lines to 5 lines!

Next, consider the following example.

In [14]:
```
# Code to calculate total and average costs of textbooks this semester
# Prompt user for cost of books

book1 = float(input('Enter cost of first book: '))
book2 = float(input('Enter cost of second book: '))
book3 = float(input('Enter cost of third book: '))
book4 = float(input('Enter cost of fourth book: '))

# Calculate total and average costs
total_cost = book1 + book2 + book3 + book4
avg_cost = total_cost / 4

# Print costs
print()
print(f'The total cost for 4 books is ${total_cost:.2f}.')
print(f'The average textbook cost is ${avg_cost:.2f}.')
```

```
Enter cost of first book: 73
Enter cost of second book: 195
Enter cost of third book: 7
Enter cost of fourth book: 101

The total cost for 4 books is $376.00.
The average textbook cost is $94.00.
```

Let's use our knowledge about lists and loops now to write code that's more general and also more efficient.

In [15]:
```
# Code to calculate total and average costs of textbooks this semester
# Prompt user for number of books purchased

num = int(input('Enter the number of textbooks you bought: '))
books = []
for i in range(num):
    book = float(input(f'Enter the cost of book #{i+1}: '))
    books.append(book)

# Calculate and print costs
print()
total = sum(books)
print(f'The total cost for {num} books is ${total:.2f}.')
print(f'The average textbook cost is ${total/num:.2f}.')
```

```
Enter the number of textbooks you bought: 4
Enter the cost of book #1: 73
Enter the cost of book #2: 195
Enter the cost of book #3: 7
Enter the cost of book #4: 101

The total cost for 4 books is $376.00.
The average textbook cost is $94.00.
```

Note that we could have used `sum(books)` in our print statement, but it's a little more efficient to calculate it separately because `total` is used twice in our print statements.

Don't panic if you find counting `for` -loops to be confusing! Just look at the template and study the examples above.

5. Special Case of a Counting `for` -loop

Suppose we want to print items from a list and include a marker between the items, but we don't want the marker after the final item, e.g., we want our output to look like the following:

```
cookies | croissants | coffee ice cream | scones | brownies
```

Let's first try this using the entire list.

In [16]:
```
for i in range(len(goodies)):
    print(goodies[i], '|', end=' ')
```

```
cookies | croissants | coffee ice cream | scones | brownies |
```

This is almost what we wanted, but we don't want the last `|` . Here's what we need to do.

In [17]:
```
# Use len(goodies) - 1 and then print the last item outside the loop

for i in range(len(goodies) - 1):  # subtract 1 from len(goodies)
    print(goodies[i], '|', end=' ')
print(goodies[-1])   # This is an example of why negative indexing is usefu
```

```
cookies | croissants | coffee ice cream | scones | brownies
```

Remember this because it will often be useful!