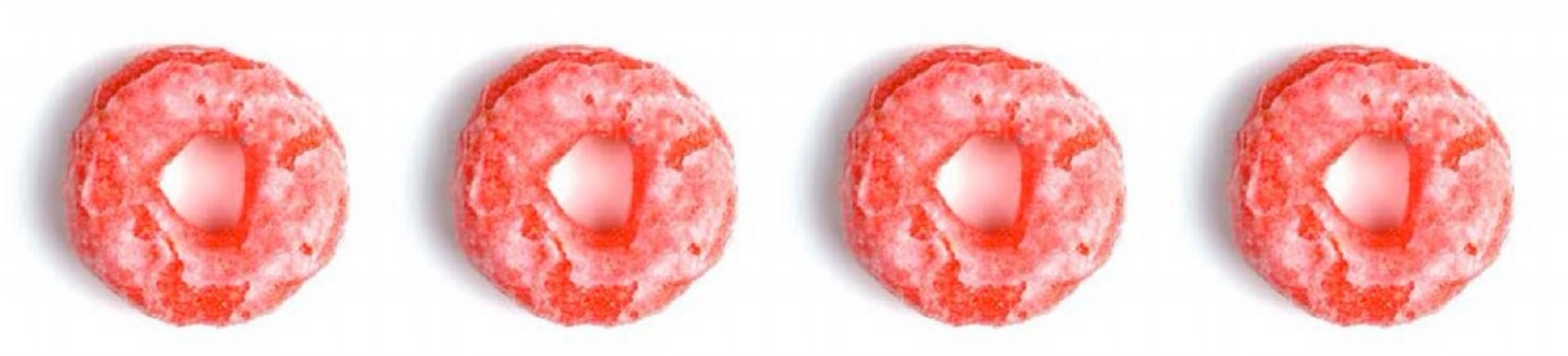

CSCI 141

Scott Wehrwein

`for` loops using the `range` function

Goals

- Know the behavior of the `range` function with 1, 2, and 3 arguments.
- Know how to use the `range` function in the header of a `for` loop.

Sequences in Python: Ranges

Lists are great if you have a list of things, but what about:

"Do `some_thing()` 10 times"? ugh.

```
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    some_thing()
```

New function to the rescue: `range`
makes it easy to generate lists like this.

Sequences in Python: Ranges

```
for i in range(5):
    print(i)
```

This code prints:

```
0
1
2
3
4
```

The `range` function returns a sequence of integers.

Not technically a list, but acts like one: more on this later

Sequences in Python: the `range` function

`range(a)`: from **0** *up to* but *not including* **a**

```
for i in range(5):
    print(i, end=" ")
```

prints: `0 1 2 3 4`

`range(a, b)`: from **a** *up to* but *not including* **b**

```
for i in range(2, 5):
    print(i, end=" ")
```

prints: `2 3 4`

`range(a, b, c)`: sequence from **a** *up to* but *not including* **b**
counting in *increments* of **c**

```
for i in range(1, 8, 3):
    print(i, end=" ")
```

prints: `1, 4, 7`

Converting ranges to lists

The `range` function returns a **sequence** of integers.

It's not technically a **list**: print(range(4)) does not print `[1, 2, 3]`

To turn the range into a list (e.g., to print it), we can use the list function:

```
list(range(2, 5)) => [2, 3, 4]
```

Range function: Demo

- demo in shell
  - one, two, and three argument versions

Size of a `range`

```
for i in range(5):
    print(i, end=" ")
```

prints: 0 1 2 3 4

```
for i in range(2, 5):
    print(i, end=" ")
```

prints: 2 3 4

```
for i in range(1, 8, 3):
    print(i, end=" ")
```

prints: 1 4 7

How many elements are in `range(n)`?

Size of a `range`

```
for i in range(5):
    print(i, end=" ")
```

prints: 0 1 2 3 4

```
for i in range(2, 5):
    print(i, end=" ")
```

prints: 2 3 4

```
for i in range(1, 8, 3):
    print(i, end=" ")
```

prints: 1, 4, 7

How many elements are in `range(a, b)`?

Back to `for` loops...

a variable name

`for` keyword

`in` keyword

```
for var_name in sequence:
    codeblock
```

????

a **sequence**: either a `list` or a call to `range`

an indented code block: one or more statements to be executed **for each** iteration of the loop

`while` loops are annoying.

- Often, you want: “Do `some_thing()` 10 times”
- With a while loop you need to:

```
i = 0
while i < 10:
    some_thing()
    i += 1
```

I don’t even care about `i`, it’s just bookkeeping!

- Wouldn’t it be great if we could:

```
for i in range(10):
    some_thing()
```

We can!

Revisiting Repetition

```
for var_name in sequence:
    codeblock
```

- balance3.py - rewrite yearly bank account balance with a for loop

- dice_possibilities2.py All possible outcomes of two 6-sided dice

- sum100.py Sum the first 100 integers

- sum_positive2.py Sum the positive numbers entered by a user until they enter a negative.

`while` **vs** `for`

Are for loops **always** better?

`while` **vs** `for`

Task: Repeatedly prompt the user for a secret password until they type it correctly.

Would you use a `for` loop or a `while` loop?

`while` **vs** `for`

Task: Ask the user for two numbers (**a** and **b**), then print the sum of the numbers between **a** and **b** (including **b** itself).

Would you use a `for` loop or a `while` loop?

`while` **vs** `for`

Task: Sum the numbers from 1 to 340, leaving out those divisible by 5.

Would you use a `for` loop or a `while` loop?

`while` vs `for`: Upshot

- If your program knows how long the loop needs to go, for loops are often cleaner.
  - Example: Do something K times.
  - Example: Do something to a list of N things
- If you don't know how many times the loop will go before it starts, while loops usually make more sense.
  - Example: loop until a number reaches zero
  - Example: prompt for a password until the user gets it right