Types of Loops in Python

CMSC 201 Section 60
September 23, 2024

Administrative Notes

Homework 2 is due tonight by midnight

Homework 3 is out

- It's due next Monday, September 30, by midnight

An early note about tests:

- In this class there are two midterm exams and one final
    - According to the syllabus, the first midterm is worth 80 points; the second midterm is worth 130 points and the final is worth 170 points
- The tests are all similar; the only difference is the length

More on tests

Midterm 1 is Wednesday, October 9

- It will cover all material covered in class through Wednesday, October 2

I try to always schedule midterms for Wednesday; and use the previous Monday as review

- All tests are in-class; written using pencil (or pen) and paper
- All tests are open book. You can use your notes; you can use my slides/code samples; you can use PyCharm and run the code to see how it works. But you have to write your code out on paper.
- YOU MAY NOT USE ChatGPT or equivalent. YOU MAY NOT use the assistance of any other person, whether in the room or not. No posting a message somewhere asking for help!!
- The tests consist of three sections: True/False/Multiple Choice, for which there is no partial credit; short answer - one or two sentences or 3-5 lines of code, partial credit; and write code, partial credit.
- I will post a “sample exam” on GitHub the week before the exam that will have the same number and types of questions as the actual exam. That’s intended as study material and test practice for you. It’s entirely optional but many students find it helpful.

More string operations:

This was in last Wednesday's video but I didn't emphasize it

“strip” and “join”

- “strip” removes white space (blank spaces, newlines, tabs) from the beginning and/or end of a string
  - strip() - both beginning and end
  - lstrip() - only beginning
  - rstrip() - only end
- “join” is almost the inverse of split()
  - It takes a list of strings, and returns a single string, connected by whatever you tell Python to use
  - Only works with lists of strings, NOT with lists of other types
  - Syntax:

```
new_str = "*".join(list_var)
```

The name of the list variable goes here

The name of the string variable goes here

Whatever's between the quotes gets inserted into the new string in between each pair of elements in the list

.upper() and .lower()

Converts all letters in strings to UPPERCASE or lowercase

Makes it easier to compare two strings

- You don’t have to worry about the user typing “M” or “m”

Has no effect on string elements that aren’t letters

- Not an error; ‘999’.upper() is just ‘999’
-

New material - Loops & Iterative execution

Iterative Code Execution

So far in Python we’ve discussed:

- ***Sequential*** code execution: execute each line of code, exactly once, in order
    - No skips, no repeating
- ***Conditional*** code execution: execute lines of code if and only if some condition is true
    - Skip code sections if the condition is not true

Today we’ll add a third type:

- ***Iterative*** code execution: execute line(s) of code multiple times
    - A set number of times, or until some condition is true, or maybe forever
    - “Execute this code zero times” is permitted, but be careful

Iterative code: Loops

Three types of loops in Python: called

- “***for*** each” loops
- “***for*** i” loops
- “***while***” loops

“***while***” loops are the most general

- You can solve any problem with a while loop; you never actually have to use a “for each” or “for i” loop
- They’re also the most complex, and the easiest to make an error with
- So sometimes you use a for loop because it’s just easier

Today’s lecture focuses on ***for*** loops; Wednesday we’ll talk about ***while*** loops

“*for*” loops

Allow us to do some things with some or all the elements of a list

Two cases to consider:

- You want to do something with each element in the list, exactly once
    - Called a “for each” loop
- You want to do something with some of the elements in the list, but not necessarily all
    - Called a “for i” loop

“for each” loops - doing the same thing with each element in a list

Start with a list:

grocery_list = [“Milk”, “Eggs”, “Cereal”, “Coffee”, “Apples”, “Strawberries”, “Broccoli”, “Cucumber”, “Tomatoes”, “Green Onions”]

We want to print the list, one item on a line, so that we can send someone else to the store.

Use a “for each” loop.

Remember that reserved word “in”? We’ll use it here.

Example

```
for item in grocery_list:

    print(item)
```

“item in grocery_list” is a boolean conditional. The colon ends the conditional. This continues to execute as long as there are more items.

Indent!!! Just like with if-else. White space is important in Python

What this means:

- “item” is a new variable; not used in the program
- Python automatically creates this to be the same type as the elements of the list
    - What if not all elements are the same type? We’ll get to that in a minute
- “item” is given the value of each element in the list, one at a time, and the code is executed for each value of “item”

Lists with different element types

```
mixed_list = [“eggs”, 12, “milk”, 128.0]
for item in mixed_list:
        print(item)


#here’s a test

for item in mixed_list:
        if item == str(item):
                print(item)
```

- *Python automatically changes the type of item to match the value of each element*

- *This test decides if a value of item is a string, or another type. The == will only be true if “item” is already a string.*

“*for* i” loops

Used when you may not want to iterate over every item in the object

Syntax:

```
for iterator in range(a, b, c): # explained in future slides
    #do something
```

“Iterator” is just a variable that tracks where you are in the list or other object. It is most often an integer, although it doesn’t absolutely have to be. “I” is often used.

The value of “iterator” does NOT have to be pre-set.

“for i” loops

- Called this because “i” is often used as the index variable. But you don’t have to use “i”
- Used to loop through an object - e.g., a list - and optionally skip some elements

range()

range() is a function that takes 3 parameters:

1. A starting integer
2. An ending integer
3. A hop size

Range will give you back all the integers from (1) to (2) hopping by (3) each time. Also, we have to wrap range in list() to see all the numbers at once.

Let’s try it!

Note: doesn’t have to be an integer, but it’s really confusing if it’s not.

Not all the arguments are needed

If you give one argument it’s the end of the range.

- What are the assumed start and hop size?

If you give two it’s the start and end of the range.

- What is the assumed hop size?

for i loops

Here's how a "for i" loop iterates through the items of a list

```
lizards = [“gecko”, “iguana”, “komodo dragon”, “chameleon”]
for i in range(len(lizards)):
    print(lizards[i])
```

Modifying your list

You can modify the contents of a list you’re iterating through, while you’re iterating through it.

An example:

```
grade_list = [98, 92.5, 123, 199.8]
for i in range(len(grade_list)):
	grade_list[i] *= grade_list[i]
print(grade_list)
```

Sample problem

Let's try to take a list of integers and increase each integer by one.

First we'll try it with a "for each" loop. What happens?

Next we'll try it with a "for i" loop. Any better?

Some questions:

1. How can we print the even numbers between two integers x and y with a ***for*** loop?
2. How can we use range to go backwards through a list?

```
range(len(integer_list)-1,0, -1)
```

3. Print all the elements of a list with their index!

“for” loops and strings

A string is zero or more characters treated as a single entity

Sounds kind of like a list, where each element of the list is one character long.

- You can treat it that way

```
word = “supercalifragilisticexpialidocious”:
for i in word:
	print(i)
```

Or:

```
word = “supercalifragilisticexpialidocious”:
for i in range(len(word)):
	print(word[i])
```

Some Examples

The “states” list from last week:

```
states = ["Alabama","Alaska","Arizona","Arkansas","California","Colorado",
"Connecticut","Delaware","Florida","Georgia","Hawaii","Idaho","Illinois",
  "Indiana","Iowa","Kansas","Kentucky","Louisiana","Maine","Maryland",
  "Massachusetts","Michigan","Minnesota","Mississippi","Missouri","Montana",
  "Nebraska","Nevada","New Hampshire","New Jersey","New Mexico","New York",
  "North Carolina","North Dakota","Ohio","Oklahoma","Oregon","Pennsylvania",
  "Rhode Island","South Carolina","South Dakota","Tennessee","Texas","Utah",
  "Vermont","Virginia","Washington","West Virginia","Wisconsin","Wyoming"]
```

- Write a “for” loop that prints out each state that ends in “a”
- Write a “for” loop that prints out each state that ends in a vowel. The hard way; the medium way; and the easy way

Validating input with ***for*** loops:

Input validation - verify that a user has actually entered an integer before trying to convert the input into an int

```
digits = ['0','1','2','3','4','5','6','7','8','9']

score = input("Please enter your test score")
is_digits = True
for j in score:
   if not j in digits:
       is_digits = False

if is_digits == True:
   print ("hooray, you entered a digit")
   score = int(score)
else:
   print ("I'm sorry, that's not a valid test score")
```

- Harder problem: do the same with a float