LECTURE 7

FOR AND WHILE LOOPS

MCS 260 Fall 2020
Emily Dumas

REMINDERS

- Quiz 3 available
- Projects 0 and 1 are out
- Project 1 autograder to be opened soon

COMPARING SEQUENCES

We talked about comparison operators >, >=, <, <=.

In addition to comparing numbers, Python allows comparison of sequences, e.g.

```
[1,2,3] > [1,1,8]
[9,8,7] < [9,8,7,6]
"McIntosh" > "Honeycrisp"
(4,) >= ()
```

The two sequences must be of the same type.

Python uses **lexicographical order** for sequences, also known as **dictionary order**.

To evaluate `L < M`, line up corresponding elements:

`L[0]`	`L[1]`	`L[2]`	...
`M[0]`	`M[1]`	`M[2]`	...

Locate the first unequal pair, and compare using <.

If we run out of elements of one sequence, consider the shorter sequence to be less.

Code points compare according to number, which means

```
"A" < "Z" < "a" < "z"
```

Therefore:

```
[1,2,3] > [1,1,8]                # True
[9,8,7] < [9,8,7,6]              # True
"McIntosh" > "Honeycrisp"        # True
(4,) >= ()                       # True
```

WHILE LOOPS

The syntax

```
while condition:
    statement
    statement
```

will repeatedly do the following:

1. Evaluate condition; if False, skip the rest of this list and move on. Otherwise,
2. Execute the statements in the block.
3. Return to the first step.

Called a **loop** because it returns to a previous line.

The code block following a while is called the **body** of the loop.

Most while loops will change a variable in the body, affecting the condition.

```
n = 1
while n <= 10:
    print(n)
    n = n + 1
```

This prints the numbers from 1 to 10.

FOR LOOPS

The syntax

```
for name in container:
    statement
    statement
```

can be used with any sequence as the container.

It will assign the name to one of the elements of container and run the loop body, repeating until each element of container has been used exactly once.

Example:

```python
for c in "MCS 260":
    if c == " ":
        print("space")
    elif c in "0123456789":
        print("digit")
    else:
        print("letter? (non-digit non-space)")
```

Output:

```
letter? (non-digit non-space)
letter? (non-digit non-space)
letter? (non-digit non-space)
space
digit
digit
digit
```

EXITING A LOOP

Both types of loops (for, while) have a way of ending "normally".

Sometimes it is helpful to exit the loop early, or from the middle of the body.

The **break** keyword does this. It applies to the innermost loop containing it.

```
n=1
while True:
    n = n + 1
    if n > 9:
        break
print(n)
```

RANGE

Other containers are allowed in for loops.

There are some that generate the items one by one, rather than computing everything in advance, e.g. `range(N)` generates the integers from 0 to $N-1$.

```python
for n in range(10):
    print(n+1)
```

We will talk more about generators in the future. For now, why use them?

The following is slow, as it creates a list of 50 million items:

```python
L = list(range(50_000_000))
for x in L:
    # do stuff with x
    # possibly exit the loop early
```

Better way:

```python
for x in range(50_000_000):
    print(x)
    if x > 100:
        break
```

This is very fast (only 102 items generated).

ENUMERATED ITERATION

What if you need the index during iteration?

This method works, but is not recommended:

```python
L = [9,8,2,4,1,1,5]
for i in range(len(L)):
    print("At index",i,"we have item",L[i])
```

Another way:

Use an extra index variable, increment it manually.

```python
L = [9,8,2,4,1,1,5]
i = 0
for x in L:
    print("At index",i,"we have item",x)
    i = i + 1
```

Best way:

Use the `enumerate()` function. It turns a sequence like `[7,6,5]` into an enumerated sequence `[ (0,7), (1,6), (2,5) ]`.

```python
L = [9,8,2,4,1,1,5]
for i,x in enumerate(L):
    print("At index",i,"we have item",x)
```

AVOID RANGE(LEN())

When you see

```
for i in range(len(L)):     # not recommended!
    # do stuff with L[i]
```

in Python code, it should usually be replaced with

```
for x in L:
    # do stuff with x
```

or

```
for i,x in enumerate(L):
    # do stuff with x and/or i
```

For and while loops allow you to write programs that process a collection of data / events / etc.

If/elif/else allow processing to be customized to the data.

Together these constructs give a lot of control over program execution.

Example: simplecalc.py, one-digit calculator. Usage:

```
$ python simplecalc.py
> add 2 5
7
> sub 8 3
5
> mul 7 6
42
> div 7 2
3.5
> exp 2 5
32
> exit
$
```

Example: simplecalc.py, one-digit calculator. Code:

```
while True:
    s = input("> ")
    if s == "exit":
        break
    cmd = s[:3]     # 3 char command
    x = int(s[4])   # 1 digit operand
    y = int(s[6])   # 1 digit operand
    if cmd == "add":
        print(x+y)
    elif cmd == "sub":
        print(x-y)
    elif cmd == "mul":
        print(x*y)
    elif cmd == "div":
        print(x/y)
    elif cmd == "exp":
        print(x**y)
    else:
        print("ERROR: Unknown command",cmd)
```

Example: rot13.py. Code:

```python
clear  = "abcdefghijklmnopqrstuvwxyz "
cipher = "nopqrstuvwxyzabcdefghijklm "

intext = input("Message: ")
outtext = ""

for c in intext:
    for i,d in enumerate(clear):
        if c == d:
            outtext = outtext + cipher[i]
            break  # exits the inner for loop

print("Encoded:",outtext)
```

Example: rot13.py. Usage:

```
$ python rot13.py
Message: hello world
Encoded: uryyb jbeyq

$ python rot13.py
Message: uryyb jbeyq
Encoded: hello world
```

REFERENCES

- In *Downey*:
    - Chapter 7 is devoted to a detailed discussion of loops
    - Section 8.3 and Section 10.3 contain additional examples of for loops.

REVISION HISTORY

- 2020-09-09 Correct iteration count in break example
- 2020-09-08 Initial publication