
Severance, Charles

http://hdl.handle.net/2027.42/64959
Unless otherwise noted, the content of this course material is licensed under a Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/.

Copyright © 2009, Charles Severance.

You assume all responsibility for use and potential liability associated with any use of the material. Material contains copyrighted content, used in accordance with U.S. law. Copyright holders of content included in this material should contact open.michigan@umich.edu with any questions, corrections, or clarifications regarding the use of content. The Regents of the University of Michigan do not license the use of third party content posted to this site unless such a license is specifically granted in connection with particular content. Users of content are responsible for their compliance with applicable law. Mention of specific products in this material solely represents the opinion of the speaker and does not represent an endorsement by the University of Michigan. For more information about how to cite these materials visit http://michigan.educommons.net/about/terms-of-use.

Any medical information in this material is intended to inform and educate and is not a tool for self-diagnosis or a replacement for medical evaluation, advice, diagnosis or treatment by a healthcare professional. You should speak to your physician or make an appointment to be seen if you have questions or concerns about this information or your medical condition. Viewer discretion is advised: Material may contain medical images that may be disturbing to some viewers.
Loop Structures and Booleans

Zelle - Chapter 8

Charles Severance - www.dr-chuck.com
Repeated Steps

Program:

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

Output:

```
0
1
2
3
4
```
Definite Loops
Definite Loops

• Loops that run a fixed (aka definite) number of times

• Loops that “iterate” through an ordered set

• Loops that run “for” a number of times

for abc in range(5) :
    print “Hi”
    print abc

Hi
0
Hi
1
Hi
2
Hi
3
Hi
4
Definite Loops

- Loops that run a fixed (aka definite) number of times
- Loops that “iterate” through an ordered set
- Loops that run “for” a number of times

```python
for abc in range(5):
    print "Hi"
    print abc
```

Colon (:) defines the start of a block. Indenting determines which lines belong to the block.
Looking at \textbf{In}...

- The iteration variable “iterates” though the sequence (ordered set)
- The block (body) of code is executed once for each value in the sequence
- The iteration variable moves through all of the values in the sequence

\texttt{for \, \texttt{abc} \, \texttt{in} \, \texttt{range(5)}} :
\begin{verbatim}
  ... block of code ...
\end{verbatim}

Five-element sequence \[ [0, 1, 2, 3, 4] \]
In a FlowChart

- The iteration variable “iterates” though the sequence (ordered set)
- The block (body) of code is executed once for each value in the sequence
- The iteration variable moves through all of the values in the sequence
Program:

```
for i in range(4):
    print i
```

Loop body is run repeatedly.
What is range(10)?

- range(10) is a built-in function that returns a sequence of numbers.
- The for statement can iterate through any sequence.
- A sequence can have values of different types.

```python
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for i in [0, 1, 2]:
    ... print(i)
...
0
1
2
>>> for i in [0, "abc", 9, 2, 3.6]:
    ... print(i)
...
0
abc
9
2
3.6
```
File Processing
File Processing

- A text file can be thought of as a sequence of lines
Opening a File

• Before we can read the contents of the file we must tell Python which file we are going to work with and what we will be doing with the file.

• This is done with the `open()` function.

• `open()` returns a “file handle” - a variable used to perform operations on the file.

• Kind of like “File -> Open” in a Word Processor.
Using open()

- `handle = open(filename, mode)`
  
  - returns a handle use to manipulate the file
  
- filename is a string
  
- mode is “r” if we are planning on reading the file and “w” if we are going to write to the file.

http://docs.python.org/lib/built-in-funcs.html
File Handle as a Sequence

- A file handle open for read can be treated as a sequence of strings where each line in the file is a string in the sequence.

- We can use the for statement to iterate through a sequence.

- Remember - a sequence is an ordered set.

```python
xfile = open("mbox.txt", "r")
for cheese in xfile:
    print cheese
```
Counting Lines in a File

- Open a file read-only

- Use a for loop to read each line

- Count the lines and print out the number of lines

```python
pizza = open("mbox.txt", "r")
howmany = 0
for slice in pizza:
    howmany = howmany + 1
print howmany
```
What We Do in Loops

Note: Even though these examples are simple the patterns apply to all kinds of loops
Patterns in Loops

- Counting in loops
- Summing in loops
- Averaging in loops
- Searching in loops
- Detecting in loops
- Largest or smallest
- Using break in a loop
- Using Continue in a loop
Looping through a Set

```python
print "Before"
for thing in [3, 41, 12, 9, 74, 15] :
    print thing
print "After"
```

```bash
$ python basicloop.py
Before
3
41
12
9
74
15
After
```
What is the Largest Number?

• 3  41  12  9  74  15
What is the Largest Number?
What is the Largest Number?

largest_so_far

-13  41  74
Making “smart” loops

- The trick is “knowing” something about the whole loop when you are stuck writing code that only sees one entry at a time

- Favorite dog food…

Set some variables to initial values

For thing in data:

Look for something or do something to each entry separately, updating a variable.

Look at the variables.
Finding the **largest** value

Largest = -1
print “Before”, largest
For value in [3, 41, 12, 9, 74, 15]:
    if value > largest:
        largest = value
    print largest, value
Print “After”, largest

$ python largest.py
Before -1
3 3
41 41
41 12
41 9
74 74
74 15
After 74

We make a **variable** that contains the **largest value** we have seen so far. If the current **value** is larger, it becomes the new **largest value** we have seen so far.
Counting in a Loop

zork = 0
print "Before", zork
for thing in [3, 41, 12, 9, 74, 15] :
  zork = zork + 1
  print zork, thing
print "After", zork

$ python countloop.py
Before 0
1 3
2 41
3 12
4 9
5 74
6 15
After 6

To count how many times we execute a loop we introduce a counter variable that starts at 0 and we add one to it each time through the loop.
Summing in a Loop

```
zork = 0
print "Before", zork
for thing in [3, 41, 12, 9, 74, 15] :
    zork = zork + thing
    print zork, thing
print "After", zork
```

$ python countloop.py

```
Before 0
3 3
44 41
56 12
65 9
139 74
154 15
```

```
After 154
```

To **add up** a **value** we encounter in a loop, we introduce a **sum variable** that starts at 0 and we add the **value** to the sum each time through the loop.
Finding the Average in a Loop

count = 0
sum = 0
print "Before", count, sum
for value in [3, 41, 12, 9, 74, 15] :
    count +=1
    sum += value
    print count, sum, value
print "After", count, sum, sum / count

$ python averageloop.py
Before 0 0
1 3 3
2 44 41
3 56 12
4 65 9
5 139 74
6 154 15
After 6 154 25

An average just combines the counting and sum patterns and divides when the loop is done.
We use an `if` statement in the loop to catch the values we are looking for.
Did we encounter a value?

```python
found = 0
print "Before", found
for value in [3, 41, 12, 9, 74, 15] :
    if value == 9:
        found = 1
        print found, value
print "After", found
```

If we just want to search and know if a value was found - we use a variable that starts at zero and is set to one as soon as we find what we are looking for.

```
$ python search1.py
Before 0
0 3
0 41
0 12
1 9
1 74
1 15
After 1
```
Using a **Boolean Variable**

```python
found = False
print "Before", found
for value in [3, 41, 12, 9, 74, 15] :
    if value == 9:
        found = True
    print found, value
print "After", found
```

If we just want to search and **know if a value was found** - we use a **variable** that starts at zero and is set to one as soon as we **find** what we are looking for.

```
$ python search1.py
Before False
False 3
False 41
False 12
True 9
True 74
True 15
After True
```
Remembering where...

```python
found = False
where = -1
count = 0
print "Before", found
for value in [3, 41, 12, 9, 74, 15] :
    count = count + 1
    if value == 9:
        found = True
        where = count
        print found, where, value
print "After", found, where
```

$ python search1.py
Before False
False -1 3
False -1 41
False -1 12
True 4 9
True 4 74
True 4 15
After True 4
Finding the **largest** value

```
largest = -1
print "Before", largest
for value in [3, 41, 12, 9, 74, 15] :
    if value > largest :
        largest = value
        print largest, value
print "After", largest
```

$ python largest.py

```
Before -1
3 3
41 41
41 12
41 9
74 74
74 15
After 74
```

We make a **variable** that contains the **largest** value we have seen so far. If the current **value** is larger, it becomes the new **largest** value we have seen so far.
count = 0
print "Before"
for value in [9, 41, 12, 3, 74, 15] :
    if count == 0 :
        smallest = value
    count = count + 1
    if value < smallest :
        smallest = value
print smallest, value
print "After", smallest

Finding the smallest value

$ python smallest.py
Before
9 9
9 41
9 12
3 3
3 74
3 15
After 3

We still have a variable that is the smallest so far. The first time through the loop we take the first value to be the smallest.
Breaking out of a loop

print "Before"
for value in [3, 41, 12, 9, 74, 15] :
    print "Loop top", value
    if value == 12 :
        break
    print "Loop bottom", value
print "After"

$ python breakloop.py
Before
Loop top 3
Loop top 41
Loop top 12
After

Break immediately terminates the current loop and jumps out of the loop.
print "Before"
for value in [3, 41, 12, 9, 74, 15] :
    print "Loop top", value
    if value == 12 :
        break
    print "Loop bottom", value
print "After"

$ python breakloop.py
Before
Loop top 3
Loop bottom 3
Loop top 41
Loop bottom 41
Loop top 12
After

Break immediately terminates the current loop and jumps out of the loop.
found = False
where = -1
count = 0
print "Before", found
for value in [3, 41, 12, 9, 74, 15] :
    count = count + 1
    if value == 9:
        found = True
        where = count
        break
print found, value
print "After", found, where
Continuing with the next iteration

print "Before"
for value in [3, 41, 12, 9, 74, 15] :
    print "Loop top", value
    if value > 10 :
        continue
    print "Loop bottom", value
print "After"

$ python breakloop.py
Before
Loop top 3
Loop bottom 3
Loop top 41
Loop top 12
Loop top 9
Loop bottom 9
Loop top 74
Loop top 15
After

Continue immediately terminates the current loop iteration and jumps to the top of the loop and starts the next iteration of the loop.
Continuing with the next iteration

```python
print "Before"
for value in [3, 41, 12, 9, 74, 15] :
    print "Loop top", value
    if value > 10 :
        continue
    print "Loop bottom", value
print "After"
```

$ python breakloop.py
Before
Loop top 3
Loop bottom 3
Loop top 41
Loop top 12
Loop top 9
Loop bottom 9
Loop top 74
Loop top 15
After

**Continue** immediately terminates the current loop **iteration** and jumps to the top of the loop and starts the next **iteration** of the loop.
Nested Loops

for out in [1, 2, 3] :
    print "Top", out
    for nest in ["X", "Y"] :
        print out, nest
    print "Bottom", out

Each time the outer loop runs once, the inner loop runs completely through the loop.

$ python nested.py
  Top 1
  1 X
  1 Y
  Bottom 1
  Top 2
  2 X
  2 Y
  Bottom 2
  Top 3
  3 X
  3 Y
  Bottom 3
Boolean Operators and Expressions
Boolean Operations

• We can do calculations with boolean variables just like with integer variables

• The boolean operations are: and or not

• Comparison operators < > <= >= == != return boolean (True or False)
Boolean Operators

\[
\begin{array}{cccc}
<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P \text{ and } Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
\end{array}
\]

\[(x == 4) \text{ and } (y == 2)\]

True if both expressions are true.

\[
\begin{array}{ccc}
<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P \text{ or } Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
\end{array}
\]

\[(x == 4) \text{ or } (y == 2)\]

Evaluates to true if either expression is true.

\[
\begin{array}{cc}
<table>
<thead>
<tr>
<th>P</th>
<th>\text{not } P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
\end{array}
\]

\[\text{not } (x == 4)\]

Not “flips” the logic
- True becomes False and False becomes True.
import string

for str in ["bob", "bark at the moon", "where at"]:
    words = string.split(str)
    if len(words) >= 2 and words[1] == "at":
        print "+++++", str
    else:
        print "-----", str

$ python findat.py
----- bob
+++++ bark at the moon+++++
where at
Summary

- Loops over a set
- File Loops
- Counting in loops
- Summing in loops
- Averaging in loops
- Searching in loops
- Detecting in loops
- Largest or smallest
- Using break in a loop
- Using continue in a loop
- Boolean operations and, or, not