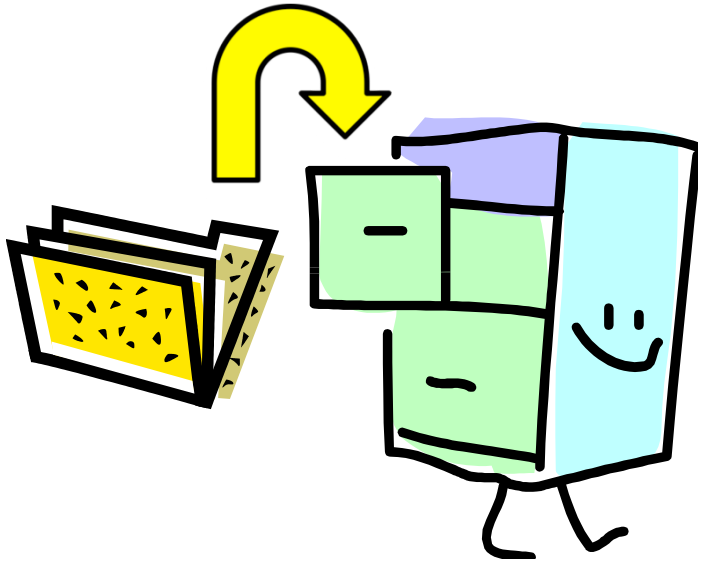
CS 1
Introduction to Computer Programming

Lecture 7: October 14, 2015

Loops and Files



Last time

- Loops using the `for` statement
- Decision-making with `if`/`elif`/`else`

Today

- **`while`** loops
- The **`break`** statement
- Files
  - opening and closing
  - the **`readlines()`** and **`readline()`** methods on file objects

More on loops

- Last time, we saw the `for` loop
- `for` is natural when working with lists
  - does something with each element of the list
- Sometimes, we're not working with lists
- Sometimes, we don't have a fixed number of things to loop over
- Sometimes, we don't know in advance how many times we will have to go through the loop

The `while` loop

- Python has a more primitive (simple) loop statement called a **`while`** loop
- Structure:

```
while <boolean expression>:
    <block of code>
```

The `while` loop

- Note similarities with `if` and `for` forms

```
while <boolean expression>:
    <block of code>
```

The `while` loop

- Note similarities with `if` and `for` forms

```
while <boolean expression>:
    <block of code>
```

- Statement starts with the keyword `while`

The `while` loop

- Note similarities with `if` and `for` forms

```
while <boolean expression>:
    <block of code>
```

- There is a colon (`:`) at the end of the first line
- It *must* be there, or else a syntax error!

The `while` loop

- Note similarities with `if` and `for` forms

```
while <boolean expression>:
    <block of code>
```

- There is an indented block of code
  - which can be one or multiple lines

The `while` loop

- Evaluation of **`while`** loop:

```
while <boolean expression>:
    <block of code>
```

1. Evaluate the **`<boolean expression>`**
2. If it evaluates to **`True`**, evaluate the **`<block of code>`** and repeat from the beginning
3. Otherwise, continue with the next line after the **`while`** loop

Example

- Starting at the number 10, print all the numbers from 10 down to 1

```
>>> num = 10
>>> while num > 0:
...     print num
...     num -= 1
10
9
```

... until reach `1`

Example

- This is equivalent to writing

```
num = 10
if num > 0:
    print num
    num -= 1
    if num > 0:
        print num
        num -= 1
        if num > 0: ...
```

(*etc.* forever)

Example

- When `num` is no longer > 0, the loop ends and execution continues on the line following the `while` loop

```
num = 10
while num > 0:
    print num
    num -= 1
print 'done with the while loop!'
```

A bad example?

- This example is unrealistic
- Could easily write this with a `for` loop:

```
for num in [10,9,8,7,6,5,4,3,2,1]:
    print num
```

- We know how many times through the loop in advance (10)

A bad example?

- Can rewrite using the `range` function to make it shorter:

```
for num in range(10, 0, -1):
    print num
```

- `range(10, 0, -1)` is equal to `[10,9,8,7,6,5,4,3,2,1]`
  - We'll see more of `range` next lecture

A better example

- Use `raw_input` to read numbers from the user and print them, stopping when a negative number is read
- In this case, we cannot know how many times we will have to go through the loop
  - because we can't control what the user does!
- This is a much more natural situation in which to use a `while` loop

A better example

```
num = int(raw_input('Enter a number: '))
while num > 0:
    print 'Your number was: %d' % num
    num = int(raw_input('Enter a number: '))
print 'Done!'
```

Sample run

```
Enter a number: 10
Your number was: 10
Enter a number: 3
Your number was: 3
Enter a number: 1729
Your number was: 1729
Enter a number: 2716057
Your number was: 2716057
Enter a number: -91
Done!
```

Ugly code

```
num = int(raw_input('Enter a number: '))
while num > 0:
    print 'Your number was: %d' % num
    num = int(raw_input('Enter a number: '))
print 'Done!'
```

- Why is this code ugly?

Ugly code

```
num = int(raw_input('Enter a number: '))
while num > 0:
    print 'Your number was: %d' % num
    num = int(raw_input('Enter a number: '))
print 'Done!'
```

- Why is this code ugly?
- The exact same line is repeated twice!
  - a "programming sin"

D.R.Y.

```
num = int(raw_input('Enter a number: '))
while num > 0:
    print 'Your number was: %d' % num
    num = int(raw_input('Enter a number: '))
print 'Done!'
```

- *Programming principle*: **D.R.Y.**
- Stands for **D**on't **R**epeat **Y**ourself

D.R.Y.

```
num = int(raw_input('Enter a number: '))
while num > 0:
    print 'Your number was: %d' % num
    num = int(raw_input('Enter a number: '))
print 'Done!'
```

- Repeated code usually means there is a better way to write the code
- Here, it allows us to introduce some new tricks

Infinite loops

```
while True:
    <block of code>
```

- This is an *infinite loop*
- There is no way for the program to complete the `while` loop
  - (at least, no way that we know yet)
  - so it just goes on running
  - can halt it by typing `Control-C`

Infinite loops

- Infinite loops are not useless!
- But we need some way to tell the loop when to stop executing
- Let's build up to that

New version of the example

```
while True:
    num = int(raw_input('Enter a number: '))
    print num
```

- This code is like previous code, except won't halt if `num < 0`
- Good thing: didn't have to write the `raw_input` line twice
- Bad thing: this never halts!
- Need a way to tell it when to stop

The `break` statement

```
while True:
    num = int(raw_input('Enter a number: '))
    if num < 0:
        break
    else:
        print num
```

- A **`break`** statement says "get out of this loop NOW!"
- It's a way to force a loop to end when some condition is met

The `break` statement

- A **`while`** loop works well when the condition to be tested can be tested before the body of the loop begins:

```
while <loop is not yet done>:
    <body of the loop>
```

The `break` statement

- A **`while`** loop needs a **`break`** statement if the condition to be tested occurs in the *middle* of the body of the loop:

```
while True:
    <body of the loop, part 1>
    if <loop is done>:
        break
    else:
        <body of the loop, part 2>
```

The `break` statement

- Here, we have:

```
while True:
    <read a number>
    if <the number is negative>:
        break
    else:
        <print the number>
```

The `break` statement

- **`break`** statements are not needed often
- **`break`** statements are never "necessary"
  - can always re-write without using **`break`**
- But when a test naturally falls in the middle of a loop body, **`break`** can make code much cleaner (less repetition, not violating **`D.R.Y.`**)
- So, when appropriate:
  - give yourself a **`break`**!

The `break` statement

- Can also write this code as:

```
while True:
    num = int(raw_input('Enter a number: '))
    if num < 0:
        break
    print num  # not inside an else
```

- Why is this OK?
- This code is preferable to previous version
  - the `else` is unnecessary so should be left out

Quick admin note

- No ombuds lunch tomorrow

Interlude

- TV clip!

Files

- Data that programs act on can be either temporary or permanent
- Values of program variables are temporary
  - they disappear when program exits
- Often want to work with more permanent data

Files

- Data that is stored on computer's hard drive is generally in the form of files
- Files are said to be "persistent"
  - Still around after the program exits
  - Still around after the computer turned off!
- Files are massively useful
  - Most programs need to store some data

Binary files and text files

- We can distinguish two kinds of files based on the way the data is formatted
- Files that contain only textual (character) data we call text files
- Files that contain "raw data" we call binary files (raw → just 0s and 1s)
- (This is an oversimplification)
- We will mostly work with text files

Example

- Let's assume there is a file containing temperature data taken once/day at noon
  - file called `temps.txt`
- Can be very large (> 1000 entries)
- Assume one number per line
- Want to read this data and compute values from it

Example

- The `temps.txt` file contains textual (character) data which can be interpreted as numeric:

```
78.2
68.3
59.0
88.1
49.5
99.0
```

(etc.)

Opening a file

- Files in Python are represented as file objects
  - *i.e.* objects which represent a file on the hard drive
  - These are not the same thing as the file, just a way to interact with the file from Python
- Before we work with a file, we have to create a file object in Python that is linked to the real file
- After that, file operations are just methods of the file object
- File objects are created using the `open` function

Opening a file

```
>>> temps = open('temps.txt', 'r')
>>> temps
<open file 'temps.txt', mode 'r' at
  0x559f8>
```

- Now the file `temps.txt` has a corresponding Python object called `temps`
- Method calls on `temps` will do something to the file `temps.txt`

Ways of opening a file

- The `open` function looks like this:

```
open(<name of file>, <mode>)
```

- `open` returns a value: a Python file object
- `<name of file>` is just the file's name, as a string
- `<mode>` determines how you can interact with the file object that `open` returns

Ways of opening a file

- Three typical values of `<mode>`:
- `'r'` – means that the file has been opened "**r**ead-only"
- `'w'` – means that the file has been opened for "**w**riting"
- `'a'` – means that the file has been opened for "**a**ppending"
- Some other modes exist (won't discuss now)

Ways of opening a file

- Read-only mode: use on an existing file you don't want to change
  - file must already exist
- Write mode: use when creating a new file from scratch
  - if file exists, it will be wiped out and overwritten!
- Append mode: use when adding to the end of an existing file
  - file must already exist, will be changed

Ways of opening a file

- Here, we will be reading from an existing file called `temps.txt`, but not writing to it
  - so we need to use the `'r'` (read-only) mode
- If we try to write to a read-only file, an error will occur
- Similarly, trying to read from a write-only file will result in an error

Closing a file

- Once we're done working with a file object, we should close it
  - prevents further actions from occurring to the file
- If we forget, file object will be closed anyway when program exits
  - but this is sloppy and bad programming practice

Closing a file

- Assume we created the `temps` file object before, corresponding to the file `temps.txt`
- To close it, we do:

```
temps.close()
```

- This causes the `close` method of the file object `temps` to be called
  - which closes the file `temps.txt`

Closing a file

- After you close a file, you can't do anything to it!
  - can't read from it or write to it
- So make sure you are truly finished with the file object before you `close` it!

Pattern for files

- Code that interacts with files will typically have this pattern:

```
f = open('temps.txt', 'r')
# do something with file object f
f.close()
```

- (May use a different file name or mode, of course)

Reading from text files

- When handling text files, can think of the file as a bunch of lines (strings ending in newline (`'\n'`) character)
- Python methods for reading from text files:
  - **`readline`** – read a single line from a text file
  - **`readlines`** – read *all* the lines of the text file

readlines

- Simplest pattern:

```
temps = open('temps.txt', 'r')
# Read all lines in file:
lines = temps.readlines()
temps.close()
```

- Now `lines` is a list of strings, one for every line in the file

readlines

- The `lines` list will look like this:

```
['78.2\n', '68.3\n', '59.0\n',
 '88.1\n', '49.5\n', '99.0\n', ...]
```

- Notes:
  1. Each element of the list is a string; need to convert it to a number before using it
  2. Each string ends in a newline character (`'\n'`)
  3. Changing the elements of the list will *not* cause the contents of the file to change (the list and the file are independent once `readlines` completes)

readlines

- The `lines` list will look like this:

```
['78.2\n', '68.3\n', '59.0\n',
 '88.1\n', '49.5\n', '99.0\n', ...]
```

- Problem 1:
  - To do anything useful with this list, need to convert all the strings into floats
- Problem 2:
  - If the file is extremely large, you now have a very large list in memory (may be more than computer can handle)

readlines

- It would be nice if we could read in lines from a file one at a time instead of all at once
- Then, could convert to numbers right after reading
- Could also process right away instead of storing into a single list
  - (if that's possible)
- Python has another useful method: `readline` (without the `'s'` at the end)

`readline` (without the `'s'`)

- The **`readline`** method reads a single line (ending in a newline (**`'\n'`**) character) and returns the line read (with the newline)
- If there are no more lines (at end of file), readline returns the empty string
  - but does *not* report an error!
- Note: an empty line in the file will return a line which consists only of the newline character
  - only way to return the empty string is at end of file

Sample problem

- Read all the lines of the file
- Assume each line contains a floating-point number
  - (won't do any error checking)
- Compute the sum of all the numbers

readline

- `readline` used with our `temps.txt` file:

```
>>> sum_nums = 0.0
>>> temps = open('temps.txt', 'r')
>>> sum_nums += float(temps.readline())
>>> sum_nums += float(temps.readline())
>>> sum_nums += float(temps.readline())
```

(etc.)

- Problem: how do we know when to stop?

`readline`

- Idea:
  - read a line
  - if the line is not empty (not at end of file), convert to `float`, add to `sum_nums` and keep reading
- Let's translate that into code

readline

- Could use a **while** loop:

```
temps = open('temps.txt', 'r')
sum_nums = 0.0
line = temps.readline()
while line != '':  # '' is empty string
    sum_nums += float(line)
    line = temps.readline()
```

- This works, but something is still not great

D.R.Y. again

- Repeated code:

```
temps = open('temps.txt', 'r')
sum_nums = 0.0
line = temps.readline()
while line != '':
    sum_nums += float(line)
    line = temps.readline()
```

- There should be a better (DRYer) way

D.R.Y. again

- Last time we used an infinite loop and a **`break`** statement to DRY up our code
- Can the same approach work this time?
- Let's give it a try...
  - (give it a *D.R.Y.*...)

D.R.Y. again

- With an infinite loop:

```
temps = open('temps.txt', 'r')
sum_nums = 0.0
while True:
    line = temps.readline()
    if line == '':
        break
    sum_nums += float(line)
```

- No repetition, very DRY

The code, in words

1. Open the file
2. Initialize `sum_nums` to zero
3. Repeat:
   a) read a line from the file
   b) if the line is empty, we're at the end of the file, so the loop is done
   c) otherwise, convert the line to a float and add to `sum_nums`

Another advantage

- Using `readline()` instead of `readlines()` to get the values in the file one-by-one means that you don't have to copy a very large file into the computer's memory
- Can use this code on arbitrarily large files without having your computer run out of memory
  - important for large data sets

Summary

- A **`while`** loop is useful for cases where the number of times the loop body will execute is not known in advance
- If you need to be able to decide to terminate a **`while`** loop in the middle of the body of the loop, you can use **`break`** with an infinite loop
- Files are represented as file objects in Python
- Files must be "opened" before use (creates a file object) and "closed" after use.
- The **`readlines()`** and **`readline()`** methods can be used to read data from a file line-by-line

Summary

- Design guideline: do not needlessly repeat lines of code (obey the *D.R.Y.* principle)!

Next time

- More on booleans
- More on loops and files
  - looping over files with `for` instead of `while`
- The `range` and `enumerate` functions