CS 1
Introduction to Computer Programming

Lecture 24: December 5, 2012
Advanced topics, part 2
Last time

- Advanced topics, lecture 1
  - recursion
  - first-class functions
  - lambda expressions
  - higher-order functions
    - map, filter, reduce
Today

- Advanced topics, lecture 2
  - command-line arguments
  - list comprehensions
  - iterators
  - generators
- Course wrap-up
Admin notes

- This is the last lecture! 😞
  - or maybe 😊?
- The final will be ready by Friday
  - due Friday, December 14th at 9 AM
Admin notes

- There is a course feedback form online
- I'd really appreciate it if you'd fill it out!
- Also, there is the "official" course feedback form (TQFR) which I would also ask you fill out
- Reason for two forms: mine is far more detailed!
Command-line arguments

- Most of the time, we've been running programs in one of two ways:
  1. importing a module directly into WingIDE and running it there
  2. running it from the terminal command line
- However, this is a very limited way of running programs
- Sometimes we need to pass information to the program at the moment we run it
Example: We are writing a program called `capitalize.py` that will
- take a text file
- create a new file which has the same contents as the original file, but capitalized

How do we write this program so that it works from the command line?
Command-line arguments

- Given what we know now, we would probably write it using `raw_input` to get the name of the original file and the name of the file we want to write, e.g.

```
% python capitalize.py
Name of input file: infile.txt
Name of output file: outfile.txt
```

- and the program would read from the input file `infile.txt` and write capitalized text to the output file `outfile.txt`
Command-line arguments

- An alternative (and simpler) approach is to make the names of the input and output file into command-line arguments:
  ```
  python capitalize.py infile.txt outfile.txt
  ```
- and the program will work the same way, without the calls to `raw_input`
Command-line arguments

```
% python capitalize.py infile.txt outfile.txt
```

- This entire line (which runs `python` on the program file `capitalize.py`) is called a **command-line**
  - *i.e.* a line containing a command
- The command-line is a feature of the terminal's command interpreter, *not* of Python
- However, Python can access the command-line from inside a Python program
Command-line arguments

```
python capitalize.py infile.txt outfile.txt
```

- The command part can be viewed as just `python` or Python and the program that Python runs (`capitalize.py`)
  - we will consider the command to be the latter (`python capitalize.py`)
Anything that comes after the command are the command-line arguments, i.e. the arguments to the command

- analogous to the arguments to a function, where the command is like a function call given to the Unix terminal

Here, the command-line arguments are:

- infile.txt
- outfile.txt
Command-line arguments

% python capitalize.py infile.txt outfile.txt

• Writing programs to use command-line arguments is usually simpler than using `raw_input` if all you need to do is give some initial information to the program
  • here, names of files to work on
• But how do we actually use command-line arguments from inside the program?
Command-line arguments

- Inside our program, we would have:

```python
import sys
if len(sys.argv) != 3:
    print >> sys.stderr, \ 
    'Not enough arguments!' 
    sys.exit(1)
infile = open(sys.argv[1], 'r')
outfile = open(sys.argv[2], 'w')
# Then do the rest of the program
```
Command-line arguments

- The `sys` module contains functions to help us work with the external "system" that a Python program runs on.
- We need to understand:
  - `sys.argv` (command-line argument list)
  - `sys.exit` (function to exit the program)
sys.exit

- **sys.exit** is basically the same as the **quit** function; it exits the program immediately
  - could just as well use **quit** here
- Normally, we give it an integer argument indicating whether or not the program exited successfully
  - 0 means "everything went well"
  - a nonzero value means "an error happened"
- Here, we give it the value **1**, meaning that an error happened
The value we pass as an argument to `sys.exit` is "the return value of the entire program"

Normally, we don't care about this, but the operating system can use this in various ways
sys.argv

- `sys.argv` is where the command-line arguments are stored every time a Python program runs.
- It is a list of strings.
- Each command-line argument (separated by spaces) is a separate string in the list.
- The first item in the list is the name of the program.
- The rest are the command-line arguments.
sys.argv

- When we run this Python program from the command-line:
  
  % python capitalize.py infile.txt outfile.txt

- Then `sys.argv` in the program is:
  
  ['capitalize.py', 'infile.txt', 'outfile.txt']

- `sys.argv[0]` is the name of the program (`capitalize.py`, without `python`)
  - normally don't need this

- Rest of `sys.argv` are the command-line arguments, which we do need
Usually, we only need `sys.argv[0]` if something goes wrong.

It's good practice to print a *usage message* informing the user that they called the program incorrectly:

- e.g. didn't specify the input or output filenames

as well as how to call the program correctly.

This code might look like this (next slide).
import sys

usage = 'usage: python %s input_file output_file'

if len(sys.argv) != 3:
    print >> sys.stderr, usage % sys.argv[0]
    sys.exit(1)

infile = open(sys.argv[1], 'r')
outfile = open(sys.argv[2], 'w')

# rest of program...
If an incorrect number of command-line arguments are given, you will see this:

```
% python capitalize.py
usage: python capitalize.py input_file output_file
```

This tells you how the program is supposed to be used, so you can use it correctly next time.
New topic!
List comprehensions

- Python has a very general way of creating lists that have particular properties called *list comprehensions*
- The idea: you declare what kind of values you want your list to contain, and Python makes it for you
List comprehensions

- List comprehensions have three components:
  - The values from which our list elements are built
  - The values we don't want in our list
  - How we combine the good values to create the list elements

- This is easier to show than to describe
  - so let's see some examples!
List comprehensions

- Simple list comprehension:

```python
>>> [2 * x for x in range(5)]
[0, 2, 4, 6, 8]
```

- A list comprehension is some Python code (with a particular structure) inside list brackets.

- Here, we have only two of the three components:
  - where the values come from (`for x in range(5)`) 
  - how to compute the list elements (`2 * x`)
List comprehensions

\[[2 \times x \text{ for } x \text{ in } \text{range}(5)]\]

- The values come from here
- We are looking at values \(x\) that are taken from the list \(\text{range}(5)\) (i.e. \([0, 1, 2, 3, 4]\))
- So the value of \(x\) is \(0\), then \(1\), then \(2\), then \(3\), then \(4\)
List comprehensions

\[ [2 \times x \text{ for } x \text{ in range}(5)] \]

- The list elements are computed from \( x \) using the expression \( 2 \times x \)
- So the value of \( 2 \times x \) is 0, then 2, then 4, then 6, then 8
- These values are collected together to give the final list: \([0, 2, 4, 6, 8]\)
List comprehensions

- List comprehensions thus provide a very compact way of creating lists with particular properties.
- We can also specify which of the values we don't want in the list by including an if statement inside the list comprehension.
List comprehensions

```python
>>> [2 * x for x in range(5) if x % 2 == 0]
[0, 4, 8]
```

- This says:
  - take all elements `x` from the list `range(5)`
  - but only if `x % 2 == 0` i.e. `x` is even i.e. `x` is either 0, 2, or 4
  - and use those `x` values to compute `2 * x`
- So the result is `[0, 4, 8]`
List comprehensions

- Another way to look at list comprehensions:
  
  \[
  [2 \times x \text{ for } x \text{ in } \text{range}(5) \text{ if } x \mod 2 == 0]
  \]

- means the same thing as:

  ```python
  result = []
  for x in range(5):
    if x % 2 == 0:
      result.append(2 * x)
  ```

- where `result` will have the final list value
List comprehensions

You can have more than one "value generator" in a list comprehension:

```python
>>> [(x, y) for x in range(3) \n    for y in [True, False]]
[(0, True), (0, False),
 (1, True), (1, False),
 (2, True), (2, False)]
```
More examples

- Create a list of all the pairs \((x, y)\) where \(x\) and \(y\) are positive and \(x + y == 5\)

```python
>>> [(x, y) for x in range(6) \
    for y in range(6) \
    if x + y == 5]
[(0, 5), (1, 4), (2, 3), (3, 2), (4, 1), (5, 0)]
```
More examples

- Create a list of all numbers between 2 and 100 which are not divisible by 2, 3, 5, or 7:

```python
>>> [n for n in range(2, 101) 
  if n % 2 != 0 
  if n % 3 != 0 
  if n % 5 != 0 
  if n % 7 != 0]
```

- (all prime numbers between 8 and 100)
map and filter

- Note that list comprehensions can also be used instead of map and filter:

```python
>>> map(lambda x: x ** 2, [1, 2, 3, 4, 5])
[1, 4, 9, 16, 25]

>>> [x ** 2 for x in [1, 2, 3, 4, 5]]
[1, 4, 9, 16, 25]

>>> filter(lambda x: x % 2 == 0, [1, 3, 4, 6, 7])
[4, 6]

>>> [x for x in [1, 3, 4, 6, 7] if x % 2 == 0]
[4, 6]
```
List comprehensions

- List comprehensions are very convenient, but not an essential feature of Python
- They don't allow you to do anything you couldn't do before
- They often *do* allow you to create a list with particular values much more concisely than you could have done it before
- Use them as you see fit
Interlude

- A classic clip!
Iterators

- We've seen that a lot of data types can be looped over inside a `for` loop:
  - lists (by list elements)
  - strings (by characters)
  - dictionaries (by keys)
  - files (by lines in the file)
- What if we have our own special data type that we want to loop over?
- What if we want to loop over a standard data type in a non-standard way?
Iterators

- What we need is a way of saying "this is how we can loop over this data type in this particular way"
- In Python, we handle this problem by creating an object called an iterator
  - i.e. "something that we can loop over in a for loop"
- Many data types already have iterators built-in to them, but we can define new ones as well
Iterators

An iterator is a special kind of Python object that can be used in a for loop:

```
for <item> in <iterator>:
    # do something with <item>
```
Iterators

- Any Python object can be an iterator if it contains two methods:
  - __iter__
    - This returns the object itself
  - next
    - This returns the "next thing" in the object
    - If there is no "next thing", this raises the StopIteration exception
- Any object that contains these two methods can be looped over in a for loop
The __iter__ method may seem useless, and it is for iterator objects. However, non-iterator objects (those that do not have a next method) can also define __iter__ to return an iterator object that iterates over the non-iterator object. Example: a list object has the __iter__ method but not the next method. Calling the __iter__ method on the list returns an iterator over the list elements.
__iter__ and next

```python
>>> lst = [1, 2, 3]
>>> i = lst.__iter__()
>>> i
<listiterator object at 0x100496a90>
>>> i.next()
1
>>> i.next()
2
>>> i.next()
3
>>> i.next()
StopIteration
```
__iter__ and next

- Iterators explain why so many different data structures (lists, strings, dictionaries, files) can work correctly in for loops.
- When you see this code:
  ```python
  for item in object: ...
  ```
  What Python is really doing is using `object.__iter__()` instead of `object` to get an iterator over the object and calling the `next` method on the iterator to get `item` every time the loop body is executed.
Examples of iterators

- Looping over a list starts at the beginning of a list and continues to the end.
- What if we want to start at the end of a list and continue back to the beginning?
- We don't want to alter the list, so using the `reverse` method is out.
- Let's define an iterator class to do this for us.
Examples of iterators

class ReverseListIterator:
    def __init__(self, lst):
        if type(lst) is not list:
            raise TypeError('need a list argument')
        self.lst = lst[:]  # copy the list
    def __iter__(self):
        return self
    def next(self):
        if self.lst == []:  # no more items
            raise StopIteration
        return self.lst.pop()
Examples of iterators

- The `ReverseListIterator` class stores a copy of a list.
- Every time it's asked for a new element (when the `next` method is called) it pops an element off the end of the list using the `pop` method on lists.
- If there are no more elements in the list, the `StopIteration` exception is raised.
- Let's see how we would use this.
Examples of iterators

```python
>>> li = ReverseListIterator([1, 2, 3, 4, 5])
>>> for i in li:
...    print i
5
4
3
2
1

• We have just extended what the `for` loop can do to handle our new iterator class
  • cool!
```
Examples of iterators

- In fact, the ReverseListIterator class is useful enough that Python provides a built-in function called reversed which creates an iterator just like this:

```python
>>> for i in reversed([1, 2, 3, 4, 5]):
...     print i
5
4
3
2
1
```
Examples of iterators

- Another example: iterating over a file character-by-character
- Recall: using a file in a `for` loop iterates over the file line-by-line
  - usually what we want, but not always
- Let's define a file iterator class to allow us to iterate over files by characters
Examples of iterators

class FileCharIterator:
    def __init__(self, file):
        self.file = file
        self.current = []
    def __iter__(self):
        return self
    def next(self):
        if self.current == []:
            nextline = self.file.readline()
            if nextline == '':
                raise StopIteration
            self.current = list(nextline)
        return self.current.pop(0)  # return first char
Examples of iterators

- The `__init__` method stores a file object in the iterator and stores a "current line" field called `current` that is initially the empty list
  - `current` will hold the current line of the file, as a list of characters
- The `__iter__` method just returns the iterator object itself
- The `next` method is where all the action is
  - so let's look at it again
Examples of iterators

class FileCharIterator:
    # ... stuff left out ...
    def next(self):
        if self.current == []:  # no more characters
            # Try to get another line from the file.
            nextline = self.file.readline()
            if nextline == '':  # end of file
                raise StopIteration
            # Convert the line to a list of characters
            self.current = list(nextline)
        # Remove (pop) the first character from current
        # and return it.
        return self.current.pop(0)
Examples of iterators

- Using the new iterator:

```python
f = open('foo.txt', 'r')
fi = FileCharIterator(f)
for char in fi:
    print char
# Prints every character of the file, on a separate line
```
Last topic!
Generators

- We take it for granted that when we return from a function, we are done with that call to the function.
- But what if it was possible to return from a function "temporarily", so we could "pick up where we left off" later?
- Python has this feature: it's called a generator.
  - because it "generates" values for us.
Generators and yield

- **The idea**: instead of using `return` to return from a function, use the new keyword `yield`
- When you `yield` a result, you are saying "here is the result you wanted, but I'm ready to keep going whenever you want more results"
- A generator is basically an iterator which is constructed automatically from a function
def fib():
    (a, b) = (0, 1)
    while True:
        yield a
        (a, b) = (b, a + b)

• This is a function that returns a generator object (because of the `yield` statement)
• The generator will generate all fibonacci numbers (0, 1, 1, 2, 3, 5, 8, 13, ...) in order
  • forever!
Generator example

- Let's see how we can use it:

  ```python
  >>> gen = fib()
  >>> gen
  <generator object at 0x5c6a30>
  >>> gen.next()
  0
  >>> gen.next()
  1
  >>> gen.next()
  1
  >>> gen.next()
  2
  ```
Generator example

Let's print the first ten fibonacci numbers:

```python
>>> gen = fib()
>>> for i, e in enumerate(gen):
...    if i >= 10:
...        break
...    print e
0
1
1
2
3
5 ...
```
Generator example 2

- Let's create a generator which will generate all prime numbers.
- A prime number is an integer \( \geq 2 \) which is only divisible by itself or 1.
- We'll use the generator to print out all primes < 100.
Generator example 2

def primes():
    prev = []  # previously-seen primes
    i = 2
    while True:  # infinite loop!
        prime = True  # assume i is prime
        for p in prev:
            if i % p == 0:  # i is not a prime
                prime = False
                break
        if prime:
            prev.append(i)
            yield i
        i += 1  # try the next integer
Generator example 2

- Using the primes generator to generate all primes below 100:

```python
>>> gen = primes()
>>> for p in gen:
...     if p >= 100:
...         break
...     print p
```
Generator example 2

- This prints:
  
  2  
  3  
  5  
  7  
  11 
  13 
  17 
  19 
  23 
  29 
  ...
Python

- Iterators and generators are two of the coolest features of Python
- Python has many more features than I could cover in this course
- The online documentation is excellent! Get familiar with it!
Python 3.x

- The version of Python we have been using is version 2.7.3
- The most recent version is version 3.3.0
- Versions 3.0 and up have quite a few (mostly non-essential) differences from the version we have been using
- Everything you need to know about this is on the Python website: www.python.org
Wrapping up
Where to go from here

- There are several courses you can take after CS 1
- CS 2 will teach more about algorithms, data structures, and give you practice with larger programming projects and application areas
  - using Java (I think)
- CS 11 will teach you specific languages
  - C, C++, Java, Erlang, Ocaml, Haskell, whatever!
  - taught by Donnie and me
Where to go from here

- **CS 4** will be a more abstract/theoretical course focusing on the big ideas of computer programming
- It will use the *Scheme* and *Ocaml* languages and will be significantly harder than CS 1
  - good for hard-core programmer types and/or current or future CS majors
- It will be awesome!
  - Oh yeah, I'm teaching that too 😊
Finally...

- I hope you enjoyed the course!
  - and learned a lot!
- If you've done well,
- if you really like programming,
- if you think you'd like teaching...
I want YOU to be a CS 1 TA!
CS 1 TAs

- If you're interested, email me
- No rush, but no later than Spring term
- Lots of work
  - but good money (~$30/hour currently)
  - and GREAT teaching experience!
- Probably 3-4 open slots at least
And...

- Thanks for letting me teach you!
- One final clip...
[End]
class ReverseListIterator:
    def __init__(self, lst):
        if type(lst) is not list:
            raise TypeError('need a list argument')
        self.lst = lst[:]
    def __iter__(self):
        return self
    def next(self):
        if self.lst == []:  # no more items
            raise StopIteration
        return self.lst.pop()
class FileCharIterator:
    # ... stuff left out ...
    def next(self):
        if self.current == []:  # no more characters
            # Try to get another line from the file.
            nextline = self.file.readline()
            if nextline == '':  # end of file
                raise StopIteration
            # Convert the line to a list of characters
            self.current = list(nextline)
        # Remove the first character from current and # return it.
        return self.current.pop(0)
def fib():
    (a, b) = (0, 1)
    while True:
        yield a
        (a, b) = (b, a + b)
def primes():
    prev = []  # previously-seen primes
    i = 2
    while True:
        prime = True  # assume i is prime
        for p in prev:
            if i % p == 0:  # i is not a prime
                prime = False
                break
        if prime:
            prev.append(i)
            yield i
        i += 1  # try the next integer