Week 5

Video 1: `while` loops

The general form of a `while` loop:

```
while expression:
    statements
```

The `while` condition, `num < 100`, is evaluated, and if it is `True` the statements in the loop body are executed. The loop condition is rechecked and if found to be `True`, the body executes again. This continues until the loop condition is checked and is `False`. For example:

```
>>> num = 2
>>> while num < 100:
        num = num * 2
        print(num)
4
8
16
32
64
128
```

In the example above, there are 6 *iterations*: the loop body executes 6 times.

Loops Conditions and Lazy Evaluation

The problem: print the characters of `str s`, up to the first vowel in `s`.

The first attempt at solving this problem works nicely when s contains one or more vowel, but results in an error if there are no vowels in s:

```
>>> i = 0
>>> s = 'xyz'
>>> while not (s[i] in 'aeiouAEIOU'):
        print(s[i])
        i = i + 1

x
y
z
Traceback (most recent call last):
    File "<pyshell#73>", line 1, in <module>
    while not (s[i] in 'aeiouAEIOU'):
IndexError: string index out of range
```

In the code above, the error occurs when `s` is indexed at `i` and `i` is outside of the range of valid indices. To prevent this `error`, add an additional condition is added to ensure that `i` is within the range of valid indices for `s`:

```
>>> i = 0
>>> s = 'xyz'
>>> while i < len(s) and not (s[i] in 'aeiouAEIOU'):
        print(s[i])
        i = i + 1

x
y
z
```

Because Python evaluates the and using lazy evaluation, if the first operand is `False`, then the expression evaluates to False and the second operand is not even evaluated. That prevents the `IndexError` from occurring.

Optional Book Reading

Chapter 9.6 Looping Until a Condition Is Reached

LECTURE CODE

```
def get_answer(prompt):
    ''' (str) -> str

    Use prompt to ask the user for a "yes" or "no"
    answer and continue asking until the user gives
    a valid response. Return the answer.
    '''

    answer = input(prompt)

    while not (answer == 'yes' or answer == 'no'):
        answer = input(prompt)

    return answer

def up_to_vowel(s):
    ''' (str) -> str

    Return a substring of s from index 0 up to but
    not including the first vowel in s.

    >>> up_to_vowel('hello')
    'h'
    >>> up_to_vowel('there')
    'th'
    >>> up_to_vowel('cs')
    'cs'
    '''

    before_vowel = ''
    i = 0

    while i < len(s) and not (s[i] in 'aeiouAEIOU'):
        before_vowel = before_vowel + s[i]
        i = i + 1

    return before_vowel
```

Video 2: Comments

The Why and How of Comments

As your programs get longer and more complicated, some additional English explanation can be used to help you and other programmers read your code. These explanations called *comments* document your code, much the way docstrings document your functions.

A comment begins with the number sign character (#) and goes until the end of the line. One name for this character is the hash character. Python ignores any lines that start with this character.

Comments are intended for programmers to read, and are usually there to explain the purpose of a function, as well as to describe relationships between your variables. Comments are to help you, and anyone else who is reading/using your code, to remember or understand the purpose of a given variable or function in a program.

Optional Book Reading

Chapter 2.7 Describing Code

Video 3: Type `list`

Overview

Our programs will often work with collections of data. One way to store these collections of data is using Python's type `list`.

The general form of a list is:

```
[expr1, expr2, ..., exprN]
```

For example, here is a list of three grades:

```
grades = [80, 90, 70]
```

List Operations

Like strings, lists can be indexed:

```
>>> grades[0]
80
>>> grades[1]
90
>>> grades[2]
70
```

Lists can also be sliced, using the same notation as for strings:

```
>>> grades[0:2]
[80, 90]
```

The in operator can also be applied to check whether a value is an item in a list.

```
>>> 90 in grades
True
>>> 60 in grades
False
```

Several of Python's built-in functions can be applied to lists, including: * `len(list):` return the length of `list`. * `min(list):` return the smallest element in `list`. * `max(list):` return the largest element in `list`. * `sum(list):` return the sum of elements of `list` (where list items must be numeric).

For example, here are some calls to those built-in functions:

```
>>> len(grades)
3
>>> min(grades)
70
>>> max(grades)
90
>>> sum(grades)
240
```

Types of list elements

Lists elements may be of any type. For example, here is a `list of str`:

```
subjects = ['bio', 'cs', 'math', 'history']
```

Lists can also contain elements of more than one type. For example, a street address can be represented by a `list of [int, str]`:

```
street_address = [10, 'Main Street']
```

`for` loops over `list`

Similar to looping over the characters of a string, it is possible to iterate over the elements of a list. For example:

```
>>> for grade in grades:
    print(grade)
80
90
70
```

The general form of a for loop over a `list` is:

```
for variable in list:
    body
```

Optional Book Reading

Chapter 8.1 Storing and Accessing Data in Lists
Chapter 8.3 Operations on Lists
Chapter 8.4 Slicing Lists

LECTURE CODE

```
Python 3.2.3 (v3.2.3:3d0686d90f55, Apr 10 2012, 11:25:50)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> grades = [80, 90, 70]
>>> grades[0]
80
>>> grades[1]
90
>>> grades[2]
70
>>> grades[1:2]
[90]
>>> grades[0:2]
[80, 90]
>>> 90 in grades
True
>>> 60 in grades
False
>>> len(grades)
3
>>> min(grades)
70
>>> max(grades)
90
>>> sum(grades)
240
>>> subjects = ['bio', 'cs', 'math', 'history']
>>> len(subjects)
4
>>> min(subjects)
'bio'
>>> max(subjects)
'math'
>>> sum(subjects)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
sum(subjects)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> street_address = [10, 'Main Street']
>>> for grade in grades:
    print(grade)
```

```
80
90
70
>>> for item in subjects:
    print(item)


bio
cs
math
history
>>> @
```

Video 4: `list` Methods

Methods

A method is a function inside an object. You can find out the methods in type `list` by typing `dir(list).`

Modifying Lists

The table below contains methods that modify lists.

Method	Description	Example

list.append(object)	Append object to the end of list.	>>> colours = ['yellow', 'blue']
		>>> colours.append('red')
		>>> print(colours)
		['yellow', 'blue', 'red']

list.extend(list)	Append the items in the list parameter to the list.	>>> colours.extend (['pink', 'green'])
		>>> print(colours)
		['yellow', 'blue', 'red', 'pink', 'green']

list.pop([index])	Remove the item at the end of the list; optional index to remove from anywhere.	>>> colours.pop()
		'green'
		>>> print(colours)
		['yellow', 'blue', 'red', 'pink']
		>>> colours.pop(2)
		'red'
		>>> print(colours)
		['yellow', 'blue', 'pink']

list.remove(object)	Remove the first occurrence of the object; error if not there.	>>> colours.remove('green')
		Traceback (most recent call last):
		File "<pyshell#10>", line 1, in
		colours.remove('green')
		ValueError: list.remove(x): x not in list
		>>> colours.remove('pink')
		>>> print(colours)
		['yellow', 'blue']

list.reverse()	Reverse the list.	>>> grades = [95, 65, 75, 85]
		>>> grades.reverse()
		>>> print(grades)
		[85, 75, 65, 95]

		>>> grades.sort()

list.sort()	Sort the list from smallest to largest.	>>> print(grades) [65, 75, 85, 95]

list.insert(int, object)	Insert object at the given index, moving items to make room.	>>> grades.insert(2, 80)
		>>> print(grades)
		[65, 75, 80, 85, 95]

Getting Information from Lists

The table below contains methods that return information about lists.

Method	Description	Example

list.count(object)	Return the number of times object occurs in list.	>>> letters = ['a', 'a', 'b', 'c']
		>>> letters.count('a')
		2

list.index(object)	Return the index of the first occurrence of object; error if not there.	>>> letters.index('a')
		0
		>>> letters.index('d')
		Traceback (most recent call last):
		File "<pyshell#24>", line 1, in
		letters.index('d')
		ValueError: 'd' is not in list

Optional Book Reading

Chapter 2.7 Describing Code

LECTURE NOTES

```
Python 3.2.3 (v3.2.3:3d0686d90f55, Apr 10 2012, 11:25:50)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__',
 '__delitem__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__getitem__', '__gt__', '__hash__',
 '__iadd__', '__imul__', '__init__', '__iter__', '__le__',
 '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
 '__setattr__', '__setitem__', '__sizeof__', '__str__',
 '__subclasshook__', 'append', 'count', 'extend', 'index',
 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> colours = []
>>> prompt = 'Enter another one of your favourite colours (type return to end): '
>>>
>>> colour = input(prompt)
Enter another one of your favourite colours (type return to end): blue
>>> colour
'blue'
>>> colours
[]
>>> while colour != '':
    colours.append(colour)
    colour = input(prompt)


Enter another one of your favourite colours (type return to end): yellow
Enter another one of your favourite colours (type return to end): brown
Enter another one of your favourite colours (type return to end):
>>> colours
['blue', 'yellow', 'brown']
```

```
>>> colours.extend(['hot pink', 'neon green'])
>>> colours
['blue', 'yellow', 'brown', 'hot pink', 'neon green']
>>> colours.pop()
'neon green'
>>> colours
['blue', 'yellow', 'brown', 'hot pink']
>>> colours.pop(2)
'brown'
>>> colours
['blue', 'yellow', 'hot pink']
>>> colours.remove('black')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
colours.remove('black')
ValueError: list.remove(x): x not in list
>>> if colours.count('yellow') > 0:
    colours.remove('yellow')


>>> colours
['blue', 'hot pink']
>>> if 'yellow' in colours:
    colours.remove('yellow')


>>> colours
['blue', 'hot pink']
>>> colours.extend('auburn', 'taupe', 'magenta')
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
colours.extend('auburn', 'taupe', 'magenta')
TypeError: extend() takes exactly one argument (3 given)
>>> colours.extend(['auburn', 'taupe', 'magenta'])
>>> colours
['blue', 'hot pink', 'auburn', 'taupe', 'magenta']
>>> colours.sort()
>>> colours
['auburn', 'blue', 'hot pink', 'magenta', 'taupe']
>>> colours.reverse()
>>> colours
['taupe', 'magenta', 'hot pink', 'blue', 'auburn']
>>> colours.insert(-2, 'brown')
>>> colours
['taupe', 'magenta', 'hot pink', 'brown', 'blue', 'auburn']
>>> colours.index('neon green')
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
colours.index('neon green')
ValueError: 'neon green' is not in list
>>> if 'hot pink' in colours:
    where = colours.index('hot pink')
    colours.pop(where)


'hot pink'
>>> colours
['taupe', 'magenta', 'brown', 'blue', 'auburn']
>>>
```

Video 5: Mutability and Aliasing

Mutability

We say that lists are *mutable*: they can be modified. All the other types we have seen so far (`str, int, float` and `bool`) are *immutable*: they

cannot be modified.

Here are several examples of lists being modified:

```
>>> classes = ['chem', 'bio', 'cs', 'eng']
>>>
>>> # Elements can be added:
>>> classes.append('math')
>>> classes
['chem', 'bio', 'cs', 'eng', 'math']
>>>
>>> # Elements can be replaced:
>>> classes[1] = 'soc'
>>> classes
['chem', 'soc', 'cs', 'eng', 'math']
>>>
>>> # Elements can be removed:
>>> classes.pop()
'math'
>>> classes
['chem', 'soc', 'cs', 'eng']
```

Aliasing

Consider the following code:

```
>>> lst1 = [11, 12, 13, 14, 15, 16, 17]
>>> lst2 = lst1
>>> lst1[-1] = 18
>>> lst2
11, 12, 13, 14, 15, 16, 18]
```

After the second statement executes, `lst1` and `lst2` both refer to the same list. When two variables refer to the same objects, they are *aliases*. If that list is modified, both of `lst1` and `lst2` will see the change.

Optional reading

Chapter 8.5 Aliasing: What's in a Name?

Video 6: Range

The Built-in Function: Range

Python has a built-in function called `range` that is useful to use when you want to generate a sequence of numbers. You can type `help(range)` in IDLE if you ever need a reminder.

The example below will print the integers 0 to 9, inclusive.

```
for i in range(10):
    print (i)
```

The form of `range` is:

```
range([start,] stop[, step]):
    return a virtual sequence of numbers from start to stop by step
```

Applications of Range

There are other options you can specify to `range`. One option is to let range generate the numbers corresponding to indices of a string or a list.

```
s = 'computer science'
for i in range(len(s)):
    print(i)
```

```
    print(s[i])
```

You can also tell `range` what index to start at. For instance, the example below starts at index 1 (as opposed to the default which is 0).

```
for i in range(1, len(s)):
    print(i)
```

You can even specify the "step" for `range`. The default stepping size is 1, which means that numbers increment by 1. The example below starts at index 1 and its step size is there (goes to every third index).

```
for i in range(1, len(s), 3):
    print(i)
```