Lab 4: Sequence Types – Lists, Tuples, Strings

CSE/IT 107

NMT Department of Computer Science and Engineering

"Programming is learned by writing programs."

— Brian Kernighan

"The purpose of computing is insight, not numbers."

— Richard Hamming, 1962

"From our own history we learn what man is capable of. For that reason we must not imagine that we are quite different and have become better."

— Richard von Weizsäcker

Introduction

In the first few labs, we showed you how to generally use Python and how to have your programs make decisions based on what the user entered. With this lab, we will be starting to show you how to do some useful things in Python: how to make lists and manipulate them – sort them, reverse them, combine them. We will also show you how to deal with strings. You have seen strings before, but we will be showing you how to manipulate them.

Contents

1 Lists

One of the most important data types in Python is the list. A list is an ordered collection of other values. For example, we might create a list of numbers representing data points on a graph or create a list of strings representing students in a class. Creating a list is simple. We simply place comma-separated values inside square brackets.

```
>>> values = [1, 2, 3, 4, 5]
>>> print(values)
[1, 2, 3, 4, 5]
```

1.1 Indices

Once we put a value in a list, we can treat the list as a single entity or access individual values. In order to access an individual element, we need to use the index of the value we want to access. The index is the position of the element in the list if we start numbering the elements from 0. When we are accessing list elements by index we can use them in any way we might use a normal variable.

```
>>> values = [23, 7, 18, 0.23, 91]
>>> print(values[0])
23
>>> print(values[2])
18
>>> print(values[3])
0.23
>>> values[3] = 7.5
>>> print(values[3])
7.5
>>> print(values)
[23, 7, 18, 7.5, 91]
>>> values[0] = values[0] + values[1]
>>> print(values)
[30, 7, 18, 7.5, 91]
```

If we wish, we can use negative array indices to reference elements starting at the end of the list. For example, -1 is the last element, -2 is the second to last element, and so on.

```
>>> values = [32, 1, 54, -3, 6]
>>> print(values[-1])
6
>>> print(values[-2])
-3
```

1.2 Values in Lists

We can use an existing variable when creating a list. If we change the value of the variable, the value in the list will stay the same.

```
>>> x = 35
>>> k = 19
>>> y = 5
>>> values = [x, k, y]
>>> print(values)
[35, 19, 5]
>>> x = 1
>>> print(values)
[35, 19, 5]
```

All values in a list do not need to be the same type. If we want, we can create a list with both numbers and strings, although this is usually a poor idea.

```
>>> values = [2, 'hello', 5.3]
>>> print(values)
[2, 'hello', 5.3]
```

If you want, you can even put a list inside a list!

```
>>> values = [1, 5, 2]
>>> more_values = [7, 'test', values]
>>> print(more_values)
[7, 'test', [1, 5, 2]]
>>> print(more_values[2])
[1, 5, 2]
```

Note that, unlike with other variables, changing an element of a list inside another list will change both values.

```
>>> values = [1, 5, 2]
>>> more_values = [7, 'test', values]
>>> print(more_values)
[7, 'test', [1, 5, 2]]
>>> values[2] = 7
>>> print(more_values)
[7, 'test', [1, 5, 7]]
>>> values = [1, 2]
>>> print(more_values)
[7, 'test', [1, 5, 7]]
```

You generally will not have to worry about this, though the reason is because we are modifying the existing value rather than create a new list. When we reassign `values`, it no longer changes the values inside inside `more_values`. This is because we are creating a new list for `values` rather than modifying the existing list.

1.3 Testing List Contents

A common operation is to test if a list contains a given value. We can do this using the `in` keyword. We can also test if a list does not contain a value using `not in`.

```
>>> values = [1, 'test', 30, 20]
>>> print(1 in values)
True
>>> print('test' in values)
True
>>> print(2 in values)
False
>>> print(2 not in values)
True
```

This could be used to simplify the example from Lab 2 involving checking user input against multiple valid passwords.

```
passwords = ['hunter2', 'hunter3', 'hunter4']

user_in = input('Please enter your password: ')

if user_in in passwords:
    print('Correct password. Welcome!')
else:
    print('Incorrect password.')
```

1.4 Slicing

Often we will want to make a new list out of part of a larger list. We do this using slicing. To do this, we specify the first and last indices we want to include in our new list, separated by a colon. The last index is used as a bookend – that is, values up to but not including that index are included in the new list. If one of the values is omitted, then Python will act as if the most extreme index on that side was enterred. Omitting both values will use the full list.

Thus, for a list `L` and two positions (indices) `B` and `E` within that list, the expression `L[ B : E ]` produces a new list containing the elements in `L` between those two positions not including `E`. Notice that `B` and `E` must be indices – whole numbers.

```
>>> L = [1, 2, 3, 4, 5]
>>> print(L[1:3])
[2, 3]
>>> print(L[3])
4
```

Note that `L[1:3]` did *not* include `L[3]`.

You may omit the starting position to slice elements from the beginning of the list up to the specified position. You may similarly omit the ending position to specify that a slice extends to the end of the list. You can even omit both of them to just get a copy of the whole list.

```
>>> L[:3]
[1, 2, 3]
>>> L[1:]
[2, 3, 4, 5]
>>> L[:]
[1, 2, 3, 4, 5]
```

Using slicing allows us to add or delete elements from lists:

```
>>> L = [1, 2, 3, 4, 5]
>>> L[2:4] = [93, 94, 95, 96]
>>> L
[1, 2, 93, 94, 95, 96, 5]
>>> L[3:6] = []
>>> L
[1, 2, 93, 5]
```

If we include an additional colon and value, we can include a step size. For example, a step size of two will create the list from every other value in the original list. A negative step size allows the list to be reversed.

Then, given a list `L` and two positions (indices) `B` and `E` as well as a step size `S`, you can use the expression `L[ B : E : S]` to obtain the elements of `L` between `B` and `E` positions in increments of `S` positions.

```
>>> values = [1, 2, 3, 4, 5]
>>> values[::2]
[1, 3, 5]
>>> values[1:4:2]
[2, 4]
>>> values[4:1:-1]
[5, 4, 3]
```

1.5 List Methods and Functions

Many functions exist to help manipulate lists. The first of these is `.append()`. This adds a new value onto the end of an existing list.

```
>>> values = [1, 2, 3]
>>> values.append(12)
>>> values.append(123)
>>> print(values)
[1, 2, 3, 12, 123]
```

`.insert()` also inserts a value, but it allows you to choose where it goes. The first argument of the function is the index you want your new value to be located. Other values will be moved to make room.

```
>>> values = [1, 2, 3, 4]
>>> values.insert(2, 10)
>>> print(values)
[1, 2, 10, 3, 4]
```

The `.pop()` function works similarly to accessing list values by index, but also removes the element from a list. If given a value, then `.pop()` will remove the value at that index. If not, it will remove the last value in the list.

```
>>> values = [1, 2, 3, 12, 123]
>>> print(values.pop())
123
>>> print(values.pop(0))
1
>>> print(values)
[2, 3, 12]
```

Using `len` returns the length of the list passed to it.

```
>>> values = [1, 2, 3, 12, 123]
>>> print(len(values))
5
```

`sum` allows the easy summing of every value in a list, so long as every value in the list is a number. If any values are not, an error will occur.

```
>>> values = [1, 2, 3, 12, 123]
>>> print(sum(values))
141
>>> values.append('test')
>>> print(sum(values))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

1.6 Combining Lists

Two lists can be added together in order to combine them into one list.

```
>>> values = [1, 2, 3]
>>> more_values = [3, 2, 1]
>>> combined = values + more_values
>>> print(combined)
[1, 2, 3, 3, 2, 1]
```

Function/method	What it does
`lst.append(x)`	appends `x` to `lst`
`lst.insert(i, x)`	inserts `x` at index `i` in `lst` (moves other elements)
`x = lst.pop()`	removes last element of `lst` and places it in `x`
`lst.reverse()`	reverses the order of elements in list `lst` in place
`lst.sort()`	sort `lst` in place
`len(lst)`	number of elements in `lst`
`sum(lst)`	sum the elements of `lst` (only works when `+` works between the elements)

Table 1: List methods and functions, where `lst` is a variable that holds a list

1.7 Summary

- A list is created from a series of comma-separated values inside square brackets.
- An element in an array can be referenced and manipulated using its index. The first element has an index of 0. Negative indices can be used to reference elements from the end of the list, with the last element having an index of -1.
- The `in` keyword can be used to test if a value exists in a list, while `not in` can be used to test if a value does not exist in a list.
- Slicing can be used to create a new list from an existing one.
- See Table 1 for functions on lists.

2 Tuples

Tuples work a lot like lists, except they cannot be modified; they are *immutable*. Instead of brackets `[]` we use parentheses `()` around the elements.

```
>>> food = ('eggs', 'bananas', 'lemonheads')
>>> food[1]
'bananas'
>>> food[1:3]
('bananas', 'lemonheads')
>>> food[1] = 'steak' # tuples are immutable!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Lists and tuples turn out to be `sequence` data types – this means they support indexing and slicing. We will see below that strings are also a sequence type.

We can also define a tuple by simply separating values by commas:

```
>>> food = 'trail mix', 'nothing'
>>> food[0]
'trail mix'
```

And of course, we can nest tuples and lists interchangably:

```
>>> foo = ([1, 2, 3], [4, 5, (7, 8, 9)], (10, 11))
>>> foo
([1, 2, 3], [4, 5, (7, 8, 9)], (10, 11))
```

Tuples, like lists, can also be empty. Use `len()` to find the length of a tuple:

```
>>> foo = ()
>>> len(foo)
0
```

We can also *unpack* tuples, which means assigning each element of a tuple to a variable. For example:

```
>>> food = 'trail mix', 'nothing'
>>> tylerfood, chrisfood = food
>>> # equivalent to, but shorter than:
... tylerfood = food[0]
... chrisfood = food[1]
>>> tylerfood
'trail mix'
>>> chrisfood
'nothing'
```

2.1 Summary

- Tuples are a like lists, but they are immutable.
- Syntax: like lists, but use parentheses `()` instead of brackets `[]`.
- Find more documentation about sequence types here:

`https://docs.python.org/3.4/library/stdtypes.html#typesseq`

3 Strings

You have seen strings in Python before. They are sequences of characters enclosed by either double quotes or single quotes; for example:

```
>>> s = "I'm a string."
>>> print(s)
I'm a string.
>>> r = 'I am also a string.'
>>> print(r)
I am also a string.
```

Notice how we did not use a single quote in the second string because it was enclosed (*delimited*) by single quotes. The proper way to use a single quote in a single quoted string or a double quote in a double quoted string goes like this:

```
>>> s = "Previously, we said \"I'm a string.\"."
>>> print(s)
Previously, we said "I'm a string.".
>>> r = 'I\'m also a string.'
>>> print(r)
I'm also a string.
```

This is called *escaping* a character. We *escaped* the double quotes and single quote respectively so that Python did not think it was the end of the string.

You do not have to escape a single quote in a double quoted string and vice versa:

```
>>> s = "I can use single quotes '' here"
>>> print(s)
I can use single quotes '' here
>>> r = 'I can use double quotes "" here'
>>> print(r)
I can use double quotes "" here
```

If you want a string to go to a new line, you use `\n` for that:

```
>>> r = 'I hear America singing, the varied carols I hear,\nThose of mechanics,' + \
...     'each one singing his as it should be blithe and strong,\nThe carpenter' + \
...     'singing his as he measures his plank or beam,'
>>> print(r)
I hear America singing, the varied carols I hear,
Those of mechanics, each one singing his as it should be blithe and strong,
The carpenter singing his as he measures his plank or beam,
```

Listing 1: Excerpt of *I Hear America Singing* by Walt Whitman

What, however, if you actually need a `\n` in a string? You will want to create a raw string. You do this by placing an `r` before the beginning quote:

```
>>> s = r'C:\Users\Files\nothing\more'
>>> print(s)
C:\Users\Files\nothing\more
```

As you can see, the \n in the middle of the string did not end up being a line break.

A lot of the operations you can do on lists also work on strings. We also saw indirectly and previously that we can concatenate strings together using the addition operator +:

```
>>> s = "The cat"
>>> r = " in the hat"
>>> t = s+r
>>> print(t)
The cat in the hat
```

In addition to that, we can repeat strings using the multiplication operator *:

```
>>> s = "Hi"
>>> r = 5*s
>>> print(r)
HiHiHiHiHi
>>> n = 3
>>> print(r + 'cat' * n)
HiHiHiHIHicatcatcat
```

Note how we can also combine both.

You previously saw *slicing* in the section on lists. Slicing works on strings, too!

```
>>> s = "The cat in the hat"
>>> print(s[2])
e
>>> print(s[4:7])
cat
>>> print(s[15:18])
hat
>>> print(s[14:18])
 hat
>>> print(s[17:14:-1])
tah
>>> print(s[17::-1])
tah eht ni tac ehT
```

As opposed to lists, strings are *immutable*. This means that the string cannot be changed at all:

```
>>> s = 'Python'
>>> s[0] = 'J'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you want to change a string, just create a new string!

```
>>> s = 'Python'
>>> r = 'J' + s[1:]
>>> print(r)
Jython
```

Just as with lists, you can test membership of a string using `in` and `not in`:

```
>>> s = 'abcde'
>>> print('c' in s)
True
>>> print('f' in s)
False
>>> print('f' not in s)
True
>>> print('ab' in s)
True
```

If you want strings to go on for two or more lines of Python code, you have to use three double quotes:

```
weizsaecker = """We in the older generation owe to young people not the
fulfillment of dreams but honesty. We must help younger people to
understand why it is vital to keep memories alive. We want to help them
to accept historical truth soberly, not one-sidedly, without taking
refuge in utopian doctrines, but also without moral arrogance. From our
own history we learn what man is capable of. For that reason we must not
imagine that we are quite different and have become better. There is no
ultimately achievable moral perfection. We have learned as human beings,
and as human beings we remain in danger. But we have the strength to
overcome such danger again and again."""
```

Listing 2: Excerpt of Richard von Weizsäcker's speech in the Bundestag to commemorate the 40th anniversary of the end of World War II.

3.1 String Methods and Functions

As with lists, there are a few functions you can use with strings.

```
>>> s = 'abcDE'
>>> print(len(s))
5
```

```
>>> print(s.upper())
ABCDE
>>> print(s.lower())
abcde
>>> print(s.capitalize())
Abcde
>>> print(s.isnumeric())
False
>>> print(s.isalpha())
True
>>> print(s.islower())
False
>>> print(s.isupper())
False
>>> r = s.replace('ab', 'more')
>>> print(r)
morecDE
>>> print(s)
abcDE
```

You can find a summary of the functions and what they do in Table 2.

String method/function	What it does
`len(s)`	Length of `s`
`r = s.upper()`	replaces all lowercase characters with uppercase and puts that in `r`
`r = s.lower()`	replaces all uppercase characters with lowercase and puts that in `r`
`r = s.capitalize()`	lower case of `s` with capital first letter and puts that in `r`
`r = s.isnumeric()`	`r` is `True` if `s` contains only numbers and `False` otherwise
`r = s.isalpha()`	`r` is `True` if `s` contains only alphabet characters and `False` otherwise
`r = s.islower()`	`r` is `True` if `s` contains only lower case characters and `False` otherwise
`r = s.isupper()`	`r` is `True` if `s` contains only upper case characters and `False` otherwise
`r = s.replace(x, y)`	replace any occurence of `x` in `s` with `y` and put result in `r`

Table 2: String methods and functions, where `s` is a string

3.2 Summary

- Syntax:

```
s = 'String'
# or
s = "String"
# or
s = """Multiline
String"""
```

- Escape single quotes in single quoted strings and double quotes in double quoted stings!

- Can slice strings just like lists, but not change them. You must make a new string if you want something different, because strings are *immutable*.
- Use `r + s` to combine (concatenate) strings `r` and `s` together and `s*n` to repeat a string `s` a number of `n` times.
- Use `r in s` to test whether `r` occurs in the string `s` and `r not in s` to test whether it does not occur in `s`.
- See string functions in Table 2.

4 Converting between Sequence Types

Just like we were able to convert between integers and floating-point types, we can convert (*cast*) lists to tuples to strings and vice versa. Use `list()`, `tuple()`, and `str()`. For example:

```
>>> food = ('nothing', 'cereal', 'lemonheads')
>>> list(food)
['nothing', 'cereal', 'lemonheads']
>>> str(food) # not particularly useful
"('nothing', 'cereal', 'lemonheads')"
>>> s = "The cat in the"
>>> list(s)
['T', 'h', 'e', ' ', 'c', 'a', 't', ' ', 'i', 'n', ' ', 't', 'h', 'e']
>>> tuple(s)
('T', 'h', 'e', ' ', 'c', 'a', 't', ' ', 'i', 'n', ' ', 't', 'h', 'e')
```

5 `for` Loops

5.1 Lists

`for` loops in Python are designed to primarily work with lists and other sequence types. A `for` loop iterates through each element in a list in order, performing the same operation for each element. The following program is a simple `for` loop that prints out every element of a list.

```
values = [1, 2, 3, 4, 5]

for i in values:
    print(i)
```

This example uses one list as input to create another list consisting of the initial list's squares.

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = []

for i in values:
    squares.append(i ** 2)

print("Initial values: {}".format(values))
print("Squares: {}".format(squares))
```

This prints:

```
Initial values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

One limitation of this approach is that we cannot easily manipulate multiple values of a list at once. We can get around this using `range`. `range` produces a list of the numbers 0 up to and not including the number passed as an argument. If given two arguments, the first is used as the lower bound rather than zero.

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

`range()` can take between one and three arguments:

`range(E)`	numbers from 0 to `E`, not including `E`
`range(B, E)`	numbers from `B` to `E`, not including `E`
`range(B, E, S)`	numbers from `B` to `E`, not including `E`, skipping every `S` number

Do some experiments with range to see what numbers it gives. For reasons that we cannot yet explain to you, you have to use the `list()` function to get it to print correctly as you can see in the following example.

```
>>> print(range(5))
range(0, 5)
>>> print(list(range(5)))
[0, 1, 2, 3, 4]
>>> print(list(range(1, 5)))
[1, 2, 3, 4]
>>> print(list(range(1, 7, 2)))
[1, 3, 5]
```

If we use `range` to produce a list of elements the same length as the array being looped over, we can use its values as indices. Here is the squares example redone using `range`.

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = []

for i in range(len(values)):
    squares.append(values[i] ** 2)

print("Initial values: {}".format(values))
print("Squares: {}".format(squares))
```

Using `range` to iterate a list

Always try to **avoid** using `range` to iterate a list as shown above. Try to iterate through the list directly if you can: after all, it is a sequence type!

In this example, using `range` does not help us a significant amount. In this next example, it allows us to sum every adjacent pair of values to create a new list.

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
adjacent_sums = []

for i in range(len(values)):
    if i != 0:
        adjacent_sums.append(values[i] + values[i - 1])

print("Initial values: {}".format(values))
print("Sums: {}".format(adjacent_sums))
```

If we simply want to do something a fixed number of times, we pass that number to `range` and put it in a loop.

```
>>> for i in range(3):
...     print("Hello.")
...
Hello.
Hello.
Hello.
```

5.2 Strings

Strings are sequence types just like lists and tuples:

```
for char in 'aHdHefg':
    if char == 'H':
        print('5')
    else:
        print('1')
```

This will print:

```
1
5
1
5
1
1
1
```

5.3 `enumerate()`

Given any sequence type, `enumerate()` will return a list of tuples containing a count and an element of the list. (Actually, `enumerate()` is what we call a generator. You need not know what that is, just remember that to get an actual list you have to cast the return value to a list.) For example:

```
>>> food = ['eggs', 'hamburgers', 'steak and potatos']
>>> list(enumerate(food))
[(0, 'eggs'), (1, 'hamburgers'), (2, 'steak and potatos')]
```

This is often used with the *tuple unpacking* syntax in for loops:

```
>>> food = ['eggs', 'hamburgers', 'steak and potatos']
>>> for count, item in enumerate(food):
...     print('{} is the {}th item.'.format(item, count))
...
eggs is the 0th item.
hamburgers is the 1th item.
steak and potatos is the 2th item.
```

5.4 Summary

- A `for` loop can be used to iterate through every value in a sequence, such as a list or a string.
- If we want access to the index number of the current value, we need to use a `range` of the same length as the list.

6 Exercises

Requirements

Remember to adhere to PEP 8, PEP 257, and the boilerplate code requirement.

Exercise 6.1 (stringfun.py).
This exercise is a modification of exercises in the Google Python class, licensed under the Apache License 2.0.

Write the following functions. *You must figure out the appropriate arguments to the functions and design a command line user interface that is not part of the prescribed functions.* Please note that the command line user interface should *not* be a part of the functions below!

`ends()` Write a function that takes in a string from the user and prints just the first two and the last two characters of the string. You may assume that any input will be at least 2 characters long.

```
== ends ==
Enter a string >>> spring
spng
```

`mix()` Write a function that takes two strings `a` and `b` and prints the two strings concatenated, but with the first two characters of each word swapped with the other word's first two characters. You may assume that any input will be at least two characters long. For example:

```
== mix ==
String a >>> german
String b >>> english
enrman geglish
```

```
== mix ==
String a >>> dog
String b >>> dinner
dig donner
```

`splitit()` Consider dividing a string into two halves. If the length is even, the front and back halves are the same length. If the length is odd, we'll say that the extra character goes in the front.

For example, in `'abcde'`, the front half is `'abc'` and the back half is `'de'`.

Write a function that takes in two strings `a` and `b` and prints a-front + b-front + a-back + b-back.

For example,

```
== splitit ==
String a >>> abcd
String b >>> efghi
abefgcdhi
```

```
== splitit ==
String a >>> this dinner is
String b >>> what am i doing1
this diwhat am nner isi doing1
```

Exercise 6.2 (fractions.py).
Any fraction can be written as the division of two integers. You could express this in Python as a tuple – (numerator, denominator).

Write functions for each of the following. They must use the tuple representation to return fractions.

1. Given two fractions as tuples, multiply them.
2. Given two fractions as tuples, divide them.
3. Given a list of fractions as a tuple, return the one that is smallest in value.

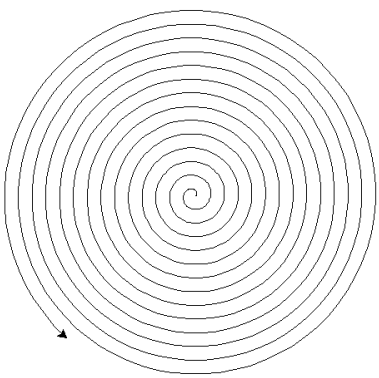
Also write a small command-line interface such that the user running your script sees something like this:

```
== Multiplication and Division ==
Enter a fraction >>> 5/3
Enter a fraction >>> 10/3
Multiplication of the fractions: 50/9
Division of the first by the second: 5/10

== Smallest fraction ==
Enter a fraction >>> 1/3
Enter a fraction >>> 10/3
Enter a fraction >>> 6/4
Enter a fraction >>> stop
Smallest fraction: 1/3
```

Exercise 6.3 (spiral.py).
Use a `for` loop, `range`, and `turtle` to draw a spiral. It does not need to perfectly resemble the example.

Sample:



Exercise 6.4 (parity.py).
Take in numbers as input until "stop" is entered. Then split the numbers into three lists: one containing all the numbers, one containing all even values, and one containing all odd. Print out all three lists, as well as each list's sum and average. Assume all input values are integers.

Sample:

```
Input a number >>> 1
Input a number >>> 5
Input a number >>> 8
Input a number >>> 2
Input a number >>> 8
Input a number >>> 100
Input a number >>> 3
Input a number >>> 7
Input a number >>> 27
Input a number >>> 5
Input a number >>> stop
All numbers: [1, 5, 8, 2, 8, 100, 3, 7, 27, 5]
Average of all numbers: 16.6
Sum of all numbers: 166
Even numbers: [8, 2, 8, 100]
Average of even numbers: 29.5
Sum of even numbers: 118
Odd numbers: [1, 5, 3, 7, 27, 5]
Average of odd numbers: 8.0
Sum of odd numbers: 48
```

Exercise 6.5 (navigate2.py).
Modify `navigate.py` from lab 2 so that, rather than performing each action as it is entered, it stores the inputs in a list and runs them all at once after the "stop" command has been given.

Exercise 6.6 (sorted.py).
Take in numbers as input until "stop" is entered. As you take in each number, insert it into a list so that the list is sorted in ascending order. That is, look through the list until you find the place where the new element belongs, then use `.insert()` to place it there. If the number is

already in the list, do not add it again. After "stop" is entered, print out the list. Do not use any of Python's built-in sorting functions.

You cannot use `.sort()` for this exercise.

Sample:

```
Input a number >>> 12
Input a number >>> 5.2
Input a number >>> 73
Input a number >>> 45
Input a number >>> 100
Input a number >>> -5
Input a number >>> 2.3
Input a number >>> stop
[-5.0, 2.3, 5.2, 12.0, 45.0, 73.0, 100.0]
```

What happens in the background:

```
Input a number >>> 12
List contains [12.0]
Input a number >>> 5.2
List contains [5.2, 12.0]
Input a number >>> 73
List contains [5.2, 12.0, 73.0]
Input a number >>> 45
List contains [5.2, 12.0, 45.0, 73.0]
Input a number >>> 100
List contains [5.2, 12.0, 45.0, 73.0, 100.0]
Input a number >>> -5
List contains [-5.0, 5.2, 12.0, 45.0, 73.0, 100.0]
Input a number >>> 2.3
List contains [-5.0, 2.3, 5.2, 12.0, 45.0, 73.0, 100.0]
Input a number >>> stop
[-5.0, 2.3, 5.2, 12.0, 45.0, 73.0, 100.0]
```

Exercise 6.7 (sets.py).
Write the following functions:

`overlap()` Given two lists, find a list of the elements common to both lists and return it.

`join()` Given two lists, join them together to be one list without duplicate elements and return that list.

Write a small command line interface that is not part of these functions that will look something like this:

```
List 1
Enter number >>> 1
Enter number >>> 2
Enter number >>> 4.5
List 1 is [1.0, 2.0, 4.5]

List 2
Enter number >>> 2
Enter number >>> 4.5
Enter number >>> 5
List 2 is [2.0, 4.5, 5.0]

Overlap is [2.0, 4.5]
Join is [1.0, 2.0, 4.5, 5.0]
```

7 Submitting

You should submit your code as a tarball. It should contain all files used in the exercises for this lab. The submitted file should be named

`cse107_firstname_lastname_lab4.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit