Lab 3: Lists and Strings

CSE/IT 107

NMT Computer Science

"Each decision we make, each action we take, is born out of an intention."

— Sharon Salzberg

"Programming is learned by writing programs."

— Brian Kernighan

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

— Richard Hamming, 1962

---

1 Introduction

In the first two 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 and how to deal with strings. You have seen strings before, but we will be showing you how to do neat things to them.

2 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.

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

2.1 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
```

2.2 Values in Arrays

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 (though 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 won't 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.

2.3 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.')
```

2.4 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.

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

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

2.5 List 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 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 functions 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'
```

2.6 Summing 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]
```

2.7 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.

- Slicing can be used to create a new list from an existing one.
- `.append()` adds an item to the end of a list, `.insert()` adds an item at a given index, `.pop()` returns and removes a value from the list, `len()` will give the length of a list, and `sum()` sums all values in a list.

2.8 Exercises

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.

Sample:

```
Input a number: 12
Input a number: 5.2
Input a number: 73
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, 73.0, 100.0]
```

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
```

3 Strings

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

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

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
```

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

```
>>> s = "The cat in the hat"
>>> 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
```

If you want strings to go on for multiple lines, 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."""
```

3.1 Summary

3.2 Exercises

4 For Loops Again

4.1 Lists

`for` loops in python are designed to primarily work with lists. 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 initual 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))
```

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
```

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))
```

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]
sums = []

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

print("Initial values: {}".format(values))
print("Sums: {}".format(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(10):
    print("Hello.")
```

4.2 Strings

4.3 Summary

- A `for` loop can be used to iterate through every value in a list.
- 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.

4.4 Exercises

navigate2.py Modify that `navigate.py` from last week so that, rather than performing each action as it is input, it stores the inputs in a list and runs them all at once after the "stop" command has been given.

5 Submitting

Files to submit:

- sorted.py (see Section 2.8)
- navigate2.py (see Section 4.4)
- parity.py (see Section 4.4)

You may submit your code as either a tarball (instructions below) or as a .zip file. Either one should contain all files used in the exercises for this lab. The submitted file should be named either `cse107_firstname_lastname_lab3.zip` or `cse107_firstname_lastname_lab3.tar.gz` depending on which method you used.

For Windows, use a tool you like to create a `.zip` file. The TCC computers should have `7z` installed. For Linux, look at lab 1 for instructions on how to create a tarball or use the "Archive Manager" graphical tool.

Upload your tarball or .zip file to Canvas.