Lists and For Loops

CSE/IT 107L

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 Weizsácker

Figure 1: `http://xkcd.com/1185`

Introduction

We have discussed several types of Python values. These include integers, floating point numbers, strings, and booleans. This lab marks the start of your introduction containers for Python values. These are lists, tuples, strings, sets, and dictionaries. They are used to organize Python values and are called *data structures*. In this lab, you will learn about lists, an ordered collection of Python values such as `[1, 2, 3]`. You will learn how to create, modify, and slice lists. You will also learn more about `for` loops, a powerful Python feature for accessing values in data structures.

Contents

1 Lists: Ordered Collections of Python Values **1**
- 1.1 Nested Lists: Lists Containing Lists . . . 1
- 1.2 Accessing Elements Using Indexing . . . 2
  - 1.2.1 Access Elements From the End Using Negative Indices . . . 3
  - 1.2.2 A Potential Error: Index Out of Range . . . 3
- 1.3 Checking List Properties With `len`, `sum` . . . 4
  - 1.3.1 Lists to Booleans Using `in`, `==`, `!=` . . . 4
- 1.4 Modifying Lists and Their Elements . . . 5
  - 1.4.1 Add an Element Using `list.append(value)` . . . 5
  - 1.4.2 Add an Element to Any Index Using `list.insert(index, value)` . . . 5
  - 1.4.3 Remove Elements Using `list.pop()` and `list.pop(index)` . . . 6
  - 1.4.4 Joining Two Lists With + . . . 6
  - 1.4.5 Sorting and Reversing Lists . . . 6

2 Accessing Sublists by Slicing Lists **7**
- 2.1 Access All Elements After *M* Using `[M:]` . . . 7
- 2.2 Access All Elements Before *N* Using `[:N]` . . . 7
- 2.3 Access All Elements Between *M* and *N* Using `[M:N]` . . . 8
- 2.4 Skip Every *S* Elements Using `[M:N:S]`, `[:N:S]`, `[M::S]`, and `[::S]` . . . 8

3 Lists are Mutable: Side-Effects of Modifying Elements **9**

4 Traversing Lists Using `for` Loops **9**
- 4.1 Access the Index of Current Elements With `enumerate()` . . . 10
- 4.2 Traverse Multiple Lists Using `zip()` . . . 11
- 4.3 Creating New Lists . . . 11

5 Exercises **12**

Submitting **14**

1 Lists: Ordered Collections of Python Values

One of the most important data types in Python is the list. A list is an ordered collection of Python values. For example, we might create a list of numbers representing data points on a graph or create a list of strings representing names of students in a class. To create a Python list value, we simply place comma-separated values inside square brackets. The values within a list are called elements. The number of values within a list is called the length of a list.

```
python_list = [value1, value2, value3, ...]
```

For example, the following list assigned to the variable `values` contains the integers 1, 2, 3, and 4; they're in this order.

```
>>> values = [10, 20, 30, 40, 50]
>>> print(values)
[10, 20, 30, 40, 50]
```

Here is a list of strings:

```
>>> strings = ['Mercury', 'Pluto', 'Luna']
>>> strings
['Mercury', 'Pluto', 'Luna']
```

How do you print the strings within the list? How do you modify the elements of a list? How do you combine two lists? Can you delete an element? As we will see, you can perform many operations on their elements or on the list itself.

1.1 Nested Lists: Lists Containing Lists

A Python list is just another type of Python value. Lists can store any type of Python value, therefore you may create lists that contain lists, also known as nested lists .

For example, this is a 2 dimensional list that represents a 4×5 2-color picture:

```
>>> picture = [[0, 1, 0, 1, 0],
...            [0, 0, 0, 0, 0],
...            [1, 0, 0, 0, 1],
...            [0, 1, 1, 1, 0]]
>>> print(picture)
[[0, 1, 0, 1, 0], [0, 0, 0, 0, 0], [1, 0, 0, 0, 1], [0, 1, 1, 1, 0]]
```

The triple-dots ... under the >>> indicate a multi-line code entry in the Python interactive shell. Use Shift+Enter to type multiple lines.

This is a 2 dimensional list of strings. Notice that sublists do not have to be of the same length.

```
>>> systems = [['Venus'], ['Earth', 'Moon'], ['Mars', 'Phobos', 'Deimos']]
>>> print(systems)
[['Venus'], ['Earth', 'Moon'], ['Mars', 'Phobos', 'Deimos']]
```

Finally, note that you can nest lists as deep as needed:

```
>>> balancedleaves = [[[1, 3], [4, 6]],
...                   [[11, 10, 8], [[15, 17, 41], [50], [55, 52, 53]]]]
```

1.2 Accessing Elements Using Indexing

To access (extract or look at) an element within a list, you may use indexing. To index the n^{th} element in a list, write the list or a variable containing the list and then write square brackets with the number $n-1$. This is because lists are 0-indexed, meaning the first element of a list is at index 0 and the last element is at index N-1 where N is the number of elements in the list. For example: the first year in a decade ends in 0, but the last year ends in 9. Here's a picture that shows two 0-indexed lists:

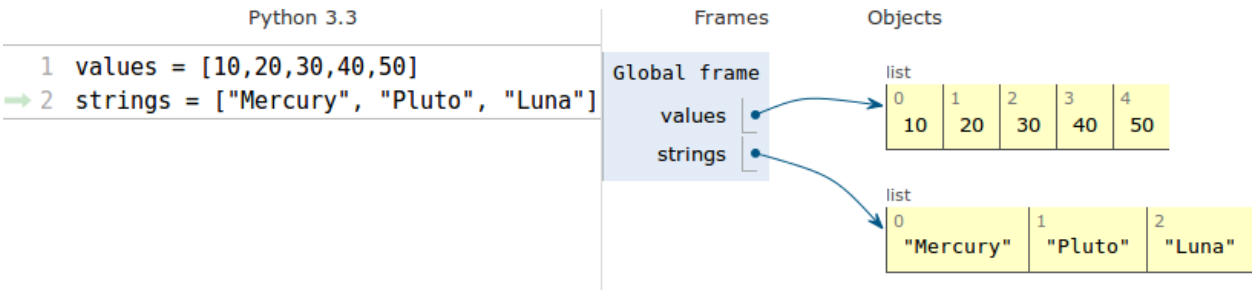

Figure 2: Two lists visualized using pythontutor.com. One is a list of integers, the other list contains strings. Note, each element has its index written directly above it. The first element in each list has index 0.

In these samples, we take values out of previously introduced lists:

```
>>> values = [10, 20, 30, 40, 50]
>>> values[0]
10
>>> values[3]
40
>>> values[5]  # does not exist
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
    values[5]  # does not exist
IndexError: list index out of range
>>> print('the first and last elements are {} and {}'.format(values[0], values[4]))
the first and last elements are 10 and 50
>>> strings = ['Mercury', 'Pluto', 'Luna']
>>> strings[0]
```

```
'Mercury'
>>> strings[2]
'Luna'
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
    values[5]  # does not exist
IndexError: list index out of range
```

Python lists contain Python values, so we can assign the elements of a list to a variable. For example,

```
>>> values = [1, 2, 3, 4]
>>> first = values[0]
>>> print(first)
1
```

1.2.1 Access Elements From the End Using Negative Indices

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
```

1.2.2 A Potential Error: Index Out of Range

What happens if you try to get the element at index 5 in a list of 5 elements?

```
>>> values = [10, 20, 30, 40, 50]
>>> values[5]
Traceback (most recent call last):
  File ``<stdin>'', line 1, in <module>
IndexError: list index out of range
Traceback (most recent call last):
  File ``<stdin>'', line 1, in <module>
IndexError: list index out of range
```

Python will raise an `IndexError` with the message "list index out of range". To avoid such an error, check the length of every list with an unknown length by using the `len` function before you try to access its elements.

This error also occurs when using negative indexing and the index is too small; such as when trying to access the -6^{th} element in a list of 5 elements.

1.3 Checking List Properties With `len`, `sum`

The `len(l)` function returns the length of the list passed to it.

```
>>> values = [1, 2, 3, 12, 123]
>>> print(len(values))
5
>>> empty = []
>>> print(len(empty))
0
```

The `sum(l)` function sums of every value in a list, so long as every value in the list is a number. If any values are not numbers, 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'
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

1.3.1 Lists to Booleans Using `in`, `==`, `!=`

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. Lists make it easier to add or remove passwords.

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

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

```
4
5  if user_in in passwords:
6      print('Correct password. Welcome!')
7  else:
8      print('Incorrect password.')
```

1.4 Modifying Lists and Their Elements

The elements of a list are similar to variables. Think of each index as a "slot" where a Python value can be stored. Because of this, we can reassign the elements of a list by using the assignment operator.

```
1  >>> values = [1, 2, 3]
2  >>> values[0] = 16
3  >>> values
4  [16, 2, 3]
5  >>> values[2] = 1
6  >>> values
7  [16, 2, 1]
8  >>> values[3] = 10
9  Traceback (most recent call last):
10   File "<stdin>", line 1, in <module>
11 IndexError: list assignment index out of range
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14 IndexError: list assignment index out of range
```

1.4.1 Add an Element Using `list.append(value)`

Many functions exist to help manipulate lists. One of these is called `list.append(value)`. This adds a new value `value` onto the end of the list `list`. Do not name any of your variables `list`, this name has special significance in Python.

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

1.4.2 Add an Element to Any Index Using `list.insert(index, value)`

`list.insert(index, value)` 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.

```
1  >>> values = [-1, 0, 1]
2  >>> values.insert(2, 10)
```

```
>>> print(values)
[-1, 0, 10, 1]
```

1.4.3 Remove Elements Using `list.pop()` and `list.pop(index)`

The `list.pop()` function is returns the Python value at the end of a list, but it also removes the value from the list. If given a value, then `list.pop(index)` will remove the value at that index.

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

1.4.4 Joining Two Lists With +

Two lists can be merged (joined or concatenated) together in order to combine them into one list using the + operator.

```
>>> values = [1, 2, 3]
>>> more_values = [3, 2, 111]
>>> combined = values + more_values
>>> print(combined)
[1, 2, 3, 3, 2, 111]
```

The + operator preserves the order of the two lists.

1.4.5 Sorting and Reversing Lists

To sort a list, use the `list.sort()` function or the `sorted(l)` function. Both functions sort lists of strings in dictionary-order, and lists of numbers in ascending order.

The keyword argument `reverse` can be set to `False` to sort the list in descending order. This looks like: `list.sort(reverse=False)`. We will discuss keyword arguments in later labs.

```
>>> values = [10, 20, 30, 40, 50]
>>> values.sort(reverse=True)
>>> values
[50, 40, 30, 20, 10]
>>> values.sort()
>>> values
[10, 20, 30, 40, 50]
```

The `list.reverse()` function reverses the given list.

```
>>> strings = ['Mercury', 'Pluto', 'Luna']
>>> strings.reverse()
>>> strings
['Luna', 'Pluto', 'Mercury']
```

2 Accessing Sublists by Slicing Lists

A sublist contains all of the elements of another list and no other elements. For example, `[1,2,3]` is a sublist of `[1,2,3,4,5]`. We can access many different sublists from a Python list by using slicing. Four of slicing methods are shown in the following subsections.

A list is a sublist of itself, and the empty list `[]` is a sublist of every list. This means slices will always result in either the original list itself, the empty list, or something between the two extremes.

Everything that can be done with a slice can also be done using for-loops or while-loops; but slices are a useful Python feature for writing concise code.

2.1 Access All Elements After M Using `[M:]`

The slice of the list `l`, `l[m:]`, will contain every element after the index `m`. Note that `l[m:]` includes `l[m]`. For example,

```
>>> strings = ['Mercury', 'Venus', 'Earth', 'Mars']
>>> strings[2:]
['Earth', 'Mars']
>>> strings[0:]
['Mercury', 'Venus', 'Earth', 'Mars']
```

2.2 Access All Elements Before N Using `[:N]`

Similarly, the slice `l[:n]` will create a sublist that contains every element before the index `n`. Note that `l[:n]` does not include `l[n]`.

```
>>> strings = ['Mercury', 'Venus', 'Earth', 'Mars']
>>> strings[:0]
[]
>>> strings[:1]
['Mercury']
>>> strings[:3]
['Mercury', 'Venus', 'Earth']
```

If out-of-bounds indices are used, the slice returns a list instead of raising an exception.

```
>>> values = [10, 20, 30, 40, 50]
>>> values[:100000]
```

```
[10, 20, 30, 40, 50]
>>> values[100000:]
[]
```

2.3 Access All Elements Between M and N Using `[M:N]`

To access all elements between two indices `m` and `n`, slice the list using `l[m:n]`. Here are some examples:

```
>>> values = [3, 5, 7, 9, 11, 13, 17, 19, 23, 25, 29, 31]
>>> values[1:5]
[5, 7, 9, 11]
>>> values[100:115]
[]
>>> values[-4:-10]
[]
>>> values[-10:-4]
[7, 9, 11, 13, 17, 19]
```

2.4 Skip Every S Elements Using `[M:N:S]`, `[:N:S]`, `[M::S]`, and `[::S]`

To skip elements, you can use a third index `s` that defines the step size. To skip every other element between two indices, use the slice `l[m:n:2]`. To skip every `s` element, use this slice: `l[m:n:s]`. This Python interactive shell session demonstrates the concept:

```
>>> values = [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
>>> values[0:9:2]
[3, 9, 15, 21, 27]
>>> values[1:10:2]
[6, 12, 18, 24, 30]
>>> values[1:10:4]
[6, 18, 30]
>>> values[0:10:4]
[3, 15, 27]
>>> values[0:10:3]
[3, 12, 21, 30]
```

You can omit either the first or second index to include either every element starting from the beginning or from the end. For example,

```
>>> values = [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
>>> values[:9:2]
[3, 9, 15, 21, 27]
>>> values[1::4]
[6, 18, 30]
>>> values[::4]
[3, 15, 27]
```

3 Lists are Mutable: Side-Effects of Modifying Elements

What does this program print?

```
x = [4, -4]
y = x
x[0] = x[0] + 1
print y
print x
```

Both variables x and y store the same list, not a copy of the list. Therefore, when you modify one variable, they both change.

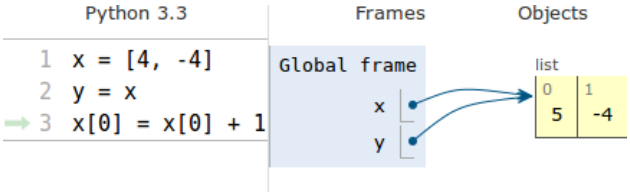

Figure 3: A visualization of the result of modifying a list when two variables are storing it. Try to run the code.

4 Traversing Lists Using `for` Loops

In Lab 1, we saw how to use `for` loops. These were only the most basic type of `for` loop. In fact, Python has many methods for traversing (or iterating) through data in a controlled way, thus avoiding the potentially unpredictable behavior of `while` loops. Most of those methods involve a `for` loop. This is how you access each element in a list:

```
values = [1, 2, 3, 4]
for value in values:
    # code that uses the `value`
```

Suppose you have a list of strings called `strings` that you need to print. You could write a while loop to print all `len(strings)` elements, or you could use this for-loop:

```
lines = ['# arithmetic.py', 'x = 7', 'y = 5 + x',
         'print(y)', 'print(x)']
for line in lines:
    print(line)
```

Like any other loop, you can have any type of Python code under the indentation. For example, you could check the current element at every iteration to see if it is larger than the largest known element and in this way you can find the maximum element in the list.

```
elements = [10, 14, -3, 5, -100, 12, 4]
maximum = elements[0]  # first guess
for element in elements:
    if element > maximum:
        maximum = element
```

If you have a nested list, you can traverse it using nested for loops:

```
picture = [[0, 1, 0, 1, 0],
           [0, 0, 0, 0, 0],
           [1, 0, 0, 0, 1],
           [0, 1, 1, 1, 0]]
for row in picture:
    for bit in row:
        if bit != 0:
            print(bit, end='')  # print without the newline
    print()  # print newline
```

4.1 Access the Index of Current Elements With `enumerate()`

If you have a list and need to access the index and the element at the index at the same time, you should consider the `enumerate(l)` function. This example prints the elements in a list, their index, and the next element in the list (if it exists).

```
>>> values = [10, 20, 30, 40, 50]
>>> for index, element in enumerate(values):
...     if index < len(values) - 1:
...         next_element = values[index + 1]
...         print('{} at {}, next: {}'.format(element, index, next_element))
...     else:
...         print('{} at {}'.format(element, index))
10 at 0, next: 20
20 at 1, next: 30
30 at 2, next: 40
40 at 3, next: 50
50 at 4
```

As another example, suppose you have a list of strings that represent lines of Python code. You can print each string and the line number by using this code:

```
>>> lines = ['# arithmetic.py', 'x = 7', 'y = 5 + x',
...          'print(y)', 'print(x)']
>>> for index, line in enumerate(lines):
...     print('{}: {}'.format(index + 1, line))
1: # arithmetic.py
2: x = 7
3: y = 5 + x
```

```
4: print(y)
5: print(x)
```

4.2 Traverse Multiple Lists Using `zip()`

The `zip(l1, l2)` function can be used to traverse multiple lists element-by-element. Here are some examples:

```
>>> group1 = [1, 10, 100, 1000, 10000]
>>> group2 = [1, 2, 4, 8, 16]
>>> for element1, element2 in zip(group1, group2):
...     print("10s: {}, 2s: {}".format(element1, element2))
10s: 1, 2s: 1
10s: 10, 2s: 2
10s: 100, 2s: 4
10s: 1000, 2s: 8
10s: 10000, 2s: 16
```

You could use `for index, (e1, e2, e3) in enumerate(zip(l1, l2, l3))` to iterate through multiple lists and access the current index.

4.3 Creating New Lists

One way to create a new list based on a previously exisitng list is to use a for-loop to check each element of the existing list and append an element to the new list. For example, the following function takes a list and returns a list without any negative numbers and with every non-negative number replaced by its square root:

```
>>> import math
>>> def sqrt_of_elements(elements):
...     result = []
...     for elem in elements:
...         if elem >= 0:
...             result.append(math.sqrt(elem))
...     return result
...
>>> sqrt_of_elements([-1, -2])
[]
>>> test_list = [-1, 0, 1, 4]
>>> r = sqrt_of_elements(test_list)
>>> r
[0.0, 1.0, 2.0]
```

5 Exercises

Exercise 5.1 (navigate2.py).
Modify `navigate.py` from lab 3 so that, instead of performing each action as it is entered, the program accepts user input without drawing anything until the "stop" command is given all while ignoring invalid inputs. After the user finishes, the program should run the drawing commands all at once.

There are many ways to approach this problem. One way is to accept the user's input and append the strings to a list. For example, if the user enters "forward, left, 50, forward, right, 20, forward, stop," you might have the following list:

```
['forward', 'left', '50', 'forward', 'right', '20', 'forward']
```

Exercise 5.2 (statistics107.py).
Write the functions listed below. They should that take a list as a parameter and return the following numbers.

`max(elements)` maximum element.

`min(elements)` minimum element.

`sum(elements)` the sum of every element in the list.

Although you may find them useful for testing, do not use the built-in functions `max`, `min`, `sum`. Also, write the following functions that take a list and return a list with the following elements:

`odds(elements)` all odd elements of the input list.

`evens(elements)` all even elements.

`every_other(elements)` finds every other element, starting from the 0^{th}.

`every_other_odd(elements)` finds every other element then returns only the odd elements.

`every_other_even(elements)` finds every other element then returns only the even elements.

You do not need to accept user input, just write the functions and use the function `run_tests` to check them.

Remember you can call your own functions from any part of your program, even from other functions that you write. The goal is to reduce repetition. What's the minimum amount of code you'll have to write?

```
>>> import statistics107
>>> statistics107.every_other([1,2,3,4])
[1, 3]
>>> statistics107.every_other_even([1, 100, 45, 23, 10, 2, 4, 13])
```

```
[100, 2]
>>> statistics107.evens([1,2,3,4])
[2, 4]
>>> statistics107.sum([1,2,3,4])
10
>>> sum([1,2,3,4])  # write your own!
10
```

Index of New Functions and Methods

+, 6
..., 1
[], 7
0-indexing, 2

assignment operator, =, 5

element access, 2
enumerate(l), 10

for, 9

in, 4
IndexError, 3
indexing, 2

l[:n], 7
l[m:], 7
l[m:n:s], 8
l[m:n], 8
length, 1
list, 1
list elements, 1
list.append(value), 5
list.insert(index, value), 5
list.pop(), 6
list.pop(index), 6
list.reverse(), 7
list.sort(), 6
list.sort(reverse=False), 6

nested lists, 1
not in, 4

sorted(l), 6

zip(l1, l2), 11

Submitting

You should submit your code as a tarball that contains all the exercise files 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

5.1 Exercise (navigate2.py) 12
5.2 Exercise (statistics107.py) 12

Exercises start on page 12.