Lists

Aidan Hunt, University of Washington

Learning Objectives

After this lesson, students will be able to

- Create simple lists in Python
- Use indexing and slicing to extract data from lists
- Use indexing and slicing to alter data in lists
- Understand list reference semantics
- Utilize useful list functions to work with lists

Announcements

- Homework 1 posted, due Friday
- Read the Style Guide
- Post questions on the discussion board

Motivation

Last time: using functions to factor out concrete tasks and reduce redundancy in our code.

This time: consider the following program for counting the number of letters in names:

In [47]:
```
def countLetters(nameIn):
    '''
    Given an input name (as a string), prints how many letters are in the name.
    '''
    numLetters = len(nameIn)
    print('The name', nameIn, 'has', numLetters, 'letters in it!')

# Define names of people in our class
name1 = 'Ty'
name2 = 'Saghar'
name3 = 'Madeline'
name4 = 'Ben'

# Call the function for each name
countLetters(name1)
countLetters(name2)
countLetters(name3)
countLetters(name4)
```

```
The name Ty has 2 letters in it!
The name Saghar has 6 letters in it!
```

```
The name Madeline has 8 letters in it!
The name Ben has 3 letters in it!
```

Observations:

- Even though we are using `countLetters` to factor out a common operation that we want to perform, this code includes several repeated calls to `countLetters` on similar data.
- Additionally, `name1` through `name4` represent data that are related to one another (names of people in ME 498 K). It would be ideal if we could store this data together in a single container, and "batch process" the names in the container.

In pseudocode, something like:

In [48]:
```
# roster = names of people in the class

# countLetters(roster)

# Desired output:
# The name Ty has 2 letters in it!
# The name Saghar has 6 letters in it!
# The name Madeline has 8 letters in it!
# The name Ben has 3 letters in it!
```

We need two things: 1) A way to organize our related data in one "container". 2) A way to perform the `countLetters` function on each piece of data in our container.

What we need for 1) is a data structure. Data structures allow us to represent our data in the computer in a way that reflects the real world.

There are many types of data structures, and today we're going to take a deep dive into the most commonly used data structure, the list.

Introducing Lists

Lists are the simplest data structure: a sequence of data!

In [49]:
```
# Create a list using square brackets
nameList = ['Ty', 'Saghar', 'Madeline', 'Ben']
print(nameList)
```

```
['Ty', 'Saghar', 'Madeline', 'Ben']
```

Now, instead of using 4 variables to store 4 names, we've put all 4 names in one container to represent that the data are related! This also better matches what we might use in the physical world to keep track of students in a class (a course roster).

What can go in a list? Anything!

In [50]:
```
# A list of numbers
numberList = [1, 2, 3, 6, 8, 9, 100, 122]
print(numberList)
```

```
[1, 2, 3, 6, 8, 9, 100, 122]
```

In [51]:
```
# A list of text
```

```
textList = ['hello', 'how', 'are', 'you']
print(textList)
```

```
['hello', 'how', 'are', 'you']
```

In [52]:
```
# A list of numbers and text
mixList = [1, 'dog', 5, 7, 'cat']
print(mixList)
```

```
[1, 'dog', 5, 7, 'cat']
```

In [53]:
```
# A list of lists!
listList = [numberList, textList, mixList]
print(listList)
```

```
[[1, 2, 3, 6, 8, 9, 100, 122], ['hello', 'how', 'are', 'you'], [1, 'dog', 5, 7, 'cat']]
```

Pause for questions

Again, lists are so handy because they allow us to organize our related data in a single "container", rather than a bunch of separate containers.

Indexing Lists

So, we've put a bunch of data into a list. Now how do we get it out?

In [54]:
```
nameList = ['Ty', 'Saghar', 'Madeline', 'Ben']
print(nameList)
```

```
['Ty', 'Saghar', 'Madeline', 'Ben']
```

We can visualize the list as a container with a bunch of slots that our data fits into.

Draw example list on the board

Each piece of data in the list is called an "element". To access an element of `nameList`, we use the square brackets:

`nameList[ind]`

where `ind` is the index of the element we want.

So, if I want to access the first element in the list, what index do I use? (Prompt students to hold up their fingers to indicate an answer).

In [55]:
```
nameList[1] # Actually accesses the second element
```

Out[55]:
```
'Saghar'
```

In [56]:
```
nameList[0] # Actually accesses the first element
```

Out[56]:
```
'Ty'
```

Python starts counting at 0!

Draw indices of example list on the board

So, to access the last element of the list, which element do we want?

In [57]:
```
nameList[3] # For list of length 4, use index of 3
```

Out[57]:
```
'Ben'
```

In [58]:
```
# What if our list was 6 names long?
nameList = ['Ty', 'Saghar', 'Madeline', 'Ben', 'Asad', 'Julie']
nameList[5] # Use index of 5
```

Out[58]:
```
'Julie'
```

In [59]:
```
# Pattern, always use length of the list - 1! Use built in len() function:
nameList[len(nameList) - 1]
```

Out[59]:
```
'Julie'
```

Pause for questions

If we are thinking of our list as a container for a bunch of data, indexing is how we access pieces of data in the container.

In [60]:
```
# Sneaky Python trick: index of -1 pulls the last element!
nameList[-1]
```

Out[60]:
```
'Julie'
```

draw negative indices on board

Getting and Setting via Indexing

Question: Have any of the indexing operations above changed our list?

In [61]:
```
nameList
```

Out[61]:
```
['Ty', 'Saghar', 'Madeline', 'Ben', 'Asad', 'Julie']
```

No, indexing is just pulling out a copy of the array element. The original contents remain unchanged.

In [62]:
```
# Assigning a variable to the value in a list
x = nameList[3]
print('x = ' + x)
print(nameList)
```

```
x = Ben
['Ty', 'Saghar', 'Madeline', 'Ben', 'Asad', 'Julie']
```

In [63]:
```
# Show that original list is not changed
x = 'Patrick Swayze'
print('x = ' + x)
print(nameList)
```

```
x = Patrick Swayze
['Ty', 'Saghar', 'Madeline', 'Ben', 'Asad', 'Julie']
```

But sometimes we do want to change the contents of a list.

To do so, we can combine variable assignments with the indexing methods we just learned.

In [64]:
```
# Changing an element in the list
print(nameList)
nameList[0] = 'Keanu Reeves'
print(nameList)
```

```
['Ty', 'Saghar', 'Madeline', 'Ben', 'Asad', 'Julie']
['Keanu Reeves', 'Saghar', 'Madeline', 'Ben', 'Asad', 'Julie']
```

Getting and Setting via Slicing

Slicing (or subsetting) is a way of pulling multiple values out of a list at once.

- Use start index (inclusive) to end index (exclusive)
- Another list is returned (if slice yields more than one index)

In [65]:
```
# Get names at indices 1, 2, and 3
nameList[1:4]
```

Out[65]:
```
['Saghar', 'Madeline', 'Ben']
```

Some shortcuts:

In [66]:
```
# Get all elements from beginning of a list up to index 4 (exclusive)

nameList[:4] # Will slice from index 0 (inclusive) to 4 (exclusive)
```

Out[66]:
```
['Keanu Reeves', 'Saghar', 'Madeline', 'Ben']
```

In [67]:
```
# Get all elements starting from an index (inclusive) to the end of the list

nameList[2:] # Will slice from index 2 (inclusive) to end of list
```

Out[67]:
```
['Madeline', 'Ben', 'Asad', 'Julie']
```

We can also use splicing to *set* list values:

In [68]:
```
# Setting list values based on another list
nameList[0:2] = ['Name1', 'Name2']
print(nameList)
```

```
['Name1', 'Name2', 'Madeline', 'Ben', 'Asad', 'Julie']
```

In [69]:
```
# Same thing as above, but just with comma separated values
nameList[0:2] = 'Name1', 'Name2'
print(nameList)
```

```
['Name1', 'Name2', 'Madeline', 'Ben', 'Asad', 'Julie']
```

You can imagine that setting list values with slicing happens in two steps.

1. Existing elements at the provided indices are removed
2. New elements are placed in the "gap"

This means that if the number of new values does not match the number of indices in the slice, your list can actually grow or shrink! So be mindful of the relationship between the indices and new values to assign.

In [70]:
```
# [0:3] is 3 indices (0, 1, and 2), but only two new values are being assigned.
# So, the resulting list is one element shorter.
nameList[0:3] = ['Test1', 'Test2']
print(nameList)
```

```
['Test1', 'Test2', 'Ben', 'Asad', 'Julie']
```

A few other notes on lists:

Lists are NOT vectors

Lists are not built for vector calculations - we need NumPy for that.

In [71]:
```
# Using '+' with lists concatenates two lists, it doesn't add the elements

[1, 2, 3] + [4, 5, 6]
```

Out[71]:
```
[1, 2, 3, 4, 5, 6]
```

In [72]:
```
# Trying to multiply a list by a constant - just concatenates the list to itself 3 times

[1, 2, 3, 4, 5] * 3
```

Out[72]:
```
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

In [73]:
```
# Trying to multiply two lists together - yields error

[1, 2, 3] * [4, 5, 6]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_1092\1215247481.py in <module>
      1 # Trying to multiply two lists together - yields error
      2
----> 3 [1, 2, 3] * [4, 5, 6]

TypeError: can't multiply sequence by non-int of type 'list'
```

Strings are like lists!

Both strings and lists are sequences, meaning that you can do many of the same operations with strings as with lists:

In [74]:
```
# Get the first three characters of a string:

testString = 'complicated'
testString[0:3]
```

Out[74]:
```
'com'
```

In [75]:
```
# Strings behave like lists when multiplied
testString * 3
```

Out[75]:
```
'complicatedcomplicatedcomplicated'
```

Lists are mutable

Recall that variables that are created using the values in other variables are not linked to those variables.

In [76]:
```
# Assign some variables and print their values
x = 2
y = 5
z = x + y

print("x =", x)
print("y =", y)
print("z =", z)
print()

# Change x and y: z does not change
x = 100
y = -50

print("x =", x)
print("y =", y)
print("z =", z)
```

```
x = 2
y = 5
z = 7

x = 100
y = -50
z = 7
```

Is this the same for lists?

In [77]:
```
# Create a list, set list1 = list2
list1 = [2, 4, 6, 8]
list2 = list1
print(list1)
print(list2)

# Change list 2, does list 1 change?
list2[0] = -500
print(list1)
print(list2)
```

```
[2, 4, 6, 8]
[2, 4, 6, 8]
[-500, 4, 6, 8]
[-500, 4, 6, 8]
```

Both lists change because lists are **mutable**, meaning that the data in the memory correpsonding to the list can change after it is created.

- Reference semantics: variables `list1` and `list2` both point to the same place in memory.
- When that memory changes, it is reflected in both variables.
- Basic data types like `int`, `float`, `str` are programmed to not do this.

Immutable version of a list: **tuple**. Tuples use parentheses for creation instead of square brackets.

In [78]:
```
# Create a tuple, set tup1 = tup2
tup1 = (1, 2, 3, 4)
tup2 = tup1
print(tup1)
print(tup2)
```

```
(1, 2, 3, 4)
(1, 2, 3, 4)
```

In [79]:
```
# Accessing tuple is okay...
tup1[0]
```

Out[79]:
```
1
```

In [80]:
```
#...but trying to change it yields an error
tup1[0] = 50
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_1092\1827647133.py in <module>
      1 #...but trying to change it yields an error
----> 2 tup1[0] = 50

TypeError: 'tuple' object does not support item assignment
```

In [81]:
```
# Strings are also immutable!
testString = 'immutable'
testString[0] = 'A'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_1092\766018959.py in <module>
      1 # Strings are also immutable!
      2 testString = 'immutable'
----> 3 testString[0] = 'A'

TypeError: 'str' object does not support item assignment
```

Question: can we change a list (mutable) that is inside a tuple (immutable)?

In [82]:
```
# Create a list (mutable) and put it in a tuple with some other data
exampleList = [1, 2, 3, 4]
exampleTuple = (exampleList, 55, 'hello')
print(exampleTuple)
```

```
([1, 2, 3, 4], 55, 'hello')
```

In [83]:
```
# Now if we change the list outside the tuple, does it update in the tuple?
exampleList[0] = -5000
print(exampleList)
print(exampleTuple)
```

```
[-5000, 2, 3, 4]
([-5000, 2, 3, 4], 55, 'hello')
```

Yes, we can! The tuple contains a *reference* to the list. The reference to the list cannot be changed, but the contents of the list that the reference points to can indeed be changed!

If we attempt to change the reference itself (e.g. assign the first element of the tuple to a different list) we get an error.

In [84]:
```
exampleTuple[0] = [5, 6, 7, 8]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_1092\631764299.py in <module>
----> 1 exampleTuple[0] = [5, 6, 7, 8]

TypeError: 'tuple' object does not support item assignment
```

Traversing Lists

So with all this in mind, let's return to our original example. We can now store related data (e.g., our classmates names) in a list, with the goal of performing the `countLetters` operation on each one.

In [85]:
```
def countLetters(nameIn):
    '''
    Given an input name (as a string), prints how many letters are in the name.
    '''
    numLetters = len(nameIn)
    print('The name', nameIn, 'has', numLetters, 'letters in it!')

# Define names of people in our class
#name1 = 'Ty'
#name2 = 'Saghar'
#name3 = 'Madeline'
#name4 = 'Ben'
roster = ['Ty', 'Saghar', 'Madeline', 'Ben']

# Call the function for each name
#countLetters(name1)
#countLetters(name2)
#countLetters(name3)
#countLetters(name4)
countLetters(roster)
```

```
The name ['Ty', 'Saghar', 'Madeline', 'Ben'] has 4 letters in it!
```

Or not....looks like our function is currently counting the number of items in the list, not the number of letters in each name!

We need to explicitly access each element of the list and perform the countLetters operation on each element. Enter: the **for loop** (next time).

Bonus: List methods

- Lists have many useful built-in methods (AKA functions that belong to the list data type)
  - **See Python documentation Section 5.1**
- Note that many of these methods could also be accomplished via indexing/slicing

Examples

Use list methods to solve these three problems in groups!

In [86]:
```
# 1) Repeat the list
# [1, 2, 3, 4, 5] ---> [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

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

```
l.extend(l)
print(l)

# Note that append does not work because it inserts l into the list
```

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

In [87]:
```
# 2) Repeat the list, but in reverse order
# This gets at the fact that these functions don't return anything --- they change the l

l = [1, 2, 3, 4, 5]
l2 = l.copy()
l2.reverse()
l.extend(l2)
print(l)
```

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

In [88]:
```
# 3) Reverse the order of the given list, except for the first element
# [1, 2, 3, 4, 5] ---> [1, 5, 4, 3, 2]

# One solution: remove the first element, reverse the list, and add it back in
l = [1, 2, 3, 4, 5]
temp = l.pop(0)
l.reverse()
l.insert(0, temp)
print(l)

# Another solution: remove the first element, append it to the list, and then reverse th
l = [1, 2, 3, 4, 5]
temp = l.pop(0)
l.append(temp)
l.reverse()
print(l)
```

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