Data structures pt I: Lists

Lesson 4 – 9/16/16

With notes!

Today's schedule

1. Lists
2. File parsing with `.split()`

1. Lists

What is a list?

- A **list** is a built-in data structure in Python (along with sets, tuples, and dictionaries)
- **What's a data structure**? It is basically a way of storing large amounts of data (numbers, strings, etc) in an organized manner, making storage and retrieval easier

Note for people who have used other programming languages:
Lists are similar to what other programming languages call "arrays". There are actually some subtle (but important) differences between lists and arrays (lists are closer to what is usually called a *linked list*), but for most purposes they perform the same role. The most obvious difference you might notice is that you don't need to specify ahead of time how large your list will be. This is because the size of the list grows dynamically as you add things to it (it also shrinks automatically as you take things out).

What is a list?

We've already seen an example of lists when we used the `range()` function:

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

>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(0,11,2)
[0, 2, 4, 6, 8, 10]
```

What is a list?

Last time we used `range()` to create `for` loops.
In fact, we can use any list to create `for` loops!

```
for i in [1, 20, 3, 19, 6]:
    print i
```

Output:

```
1
20
3
19
6
```

Practice with lists

What will the following code print?

```
for i in ["cat", "dog", "mouse", "human"]:
    print "I am a", i
```

Practice with lists

What will the following code print?

```
for i in ["cat", "dog", "mouse", "human"]:
    print "I am a", i
```

Result:

```
I am a cat
I am a dog
I am a mouse
I am a human
```

Practice with lists

What will the following code print?

```
myStuff = ["cat", 2, True, 99.5]
for i in myStuff:
    print i
```

Practice with lists

What will the following code print?

```
myStuff = ["cat", 2, True, 99.5]
for i in myStuff:
    print i
```

Result:

```
cat
2
True
99.5
```

How lists work

The lists we've seen so far look like this:

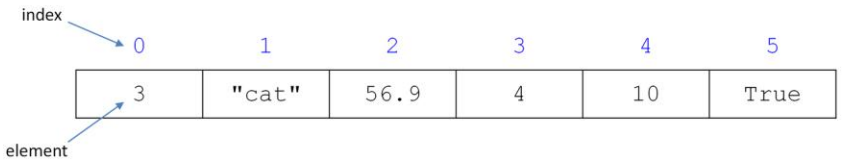
```
[3, "cat", 56.9, 4, 10, True]
```

How lists work

The lists we've seen so far look like this:

```
[3, "cat", 56.9, 4, 10, True]
```

However, it may be more helpful to think of it like this:



where each **element** is given an **index**, starting at 0.

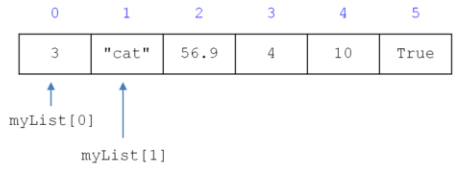
Accessing elements in a list

We use only one variable name to refer to the whole list. For example:

```
myList = [3, "cat", 56.9, 4, 10, True]
```

To access a specific element in the list, we use the following syntax: *listName*[*index*]

```
>>> myList[0]
3
>>> myList[1]
cat
```



Practice with list indexing

What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[1]
```

Practice with list indexing

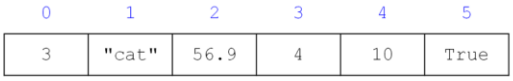
What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[1]
```

Result:

```
cat
```

Practice with list indexing



What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[4]
```

Practice with list indexing

What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[4]
```

Result:

```
10
```

Practice with list indexing

What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[6]
```

Practice with list indexing

What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[6]
```

Result:

```
Traceback (most recent call last):
  File "L5_test.py", line 2, in <module>
    print myList[6]
IndexError: list index out of range
```

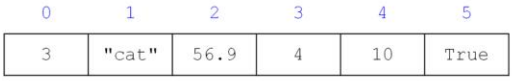
This is an "index out of bounds" error. You cannot access an index that does not exist!

Practice with list indexing

What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[-1]
```

Practice with list indexing



What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[-1]
```

Result:

```
True
```

Yep, this works! This comes in handy when you know you want the last element, but you don't know what the index of the last element is.

Practice with list indexing

What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[-2]
```

Practice with list indexing

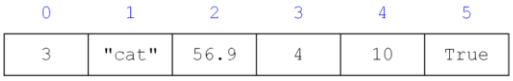
What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
print myList[-2]
```

Result:

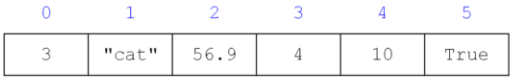
```
10
```

Practice with list indexing



How would you get the third element?

Practice with list indexing

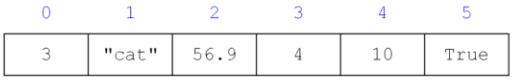


How would you get the third element?

Answer:

```
myList[2]
```

Practice with list indexing



What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
myList[0] = "dog"
print myList
```

Practice with list indexing

What will this code print?

```
myList = [3, "cat", 56.9, 4, 10, True]
myList[0] = "dog"
print myList
```

Result:

```
['dog', 'cat', 56.9, 4, 10, True]
```

This is an easy way to overwrite list elements

Creating a list

Create an empty list:

```
myList = []
```

Create a list with elements:

```
myList = [2, 7, 8]
```

Automatically create a list of numbers:

```
myList = range(5, 50, 10)
```

Adding to a list

After creating a list, you can add additional elements to the **end** using `.append()`.

Syntax:

```
list.append(newElement)
```

Example:

```
>>> myList = [2, 4, 6, 8]
>>> myList.append(10)
>>> print myList
[2, 4, 6, 8, 10]
```

Important to note:

Most of the functions we've seen so far do not modify variables directly -- they simply "return" a value. (e.g. `line.rstrip('\n')` does nothing to the original string -- it just returns a modified version. You have to say `line = line.rstrip('\n')` to actually change `line`.)

`.append()` is different. When you say `mylist.append()`, you are directly modifying `mylist`. We'll see several examples of this type of function today.

Notice that we don't have to say
myList = myList.append()
for this to work. Functions that work this way are said to be "in place" operations, because they modify the variable itself, instead of simply returning a value that must be captured. "In place" functions are not very common, but it's good to look up any new functions you use to check if they are, since you do not want to modify your variables without realizing it.

Removing from list

After creating a list, you can remove elements from it using `.pop()`.

Syntax:

```
list.pop(index)
list.pop()
```

This in-place function removes the element at the specified index, or if no index is given, removes the last item.

It also returns the removed item.

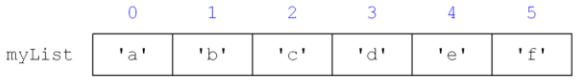
Example:

```
>>> myList = [22, 44, 66, 88]
>>> myList.pop(2)
>>> print myList
[22, 44, 88]
```

Elements that come after will be moved up one index, so that there are no empty spaces in the list.

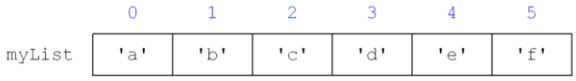
.pop() is also an in-place function, but it returns something as well: the element that was "popped"

Practice with lists



How do I add an 'g' to the end?

Practice with lists

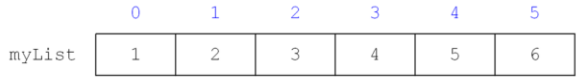


How do I add an 'g' to the end?

Answer:

```
myList.append('g')
```

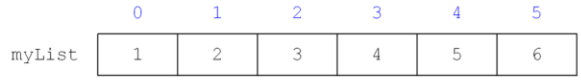
Practice with lists



What will this code print?

```
myList.pop(4)
print myList
```

Practice with lists



What will this code print?

```
myList.pop(4)
print myList
```

Answer:

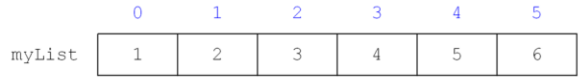
```
[1, 2, 3, 4, 6]
```

Practice with lists

What will this code print?

```
myList.pop()
print myList
```

Practice with lists



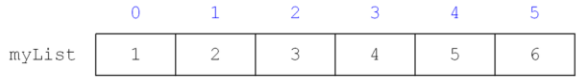
What will this code print?

```
myList.pop()
print myList
```

Answer:

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

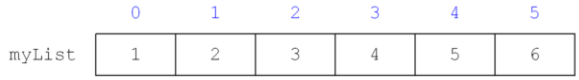
Practice with lists



What will this code print?

```
item = myList.pop()
print item
```

Practice with lists



What will this code print?

```
item = myList.pop()
print item
```

Answer:

```
6
```

Checking if something is in the list

To check if a particular element is in a list, you can just use `in`, as we've seen before:

```
myList = [22, 44, 66, 88]
if (66 in myList):
   print "found it!"
```

Code output:

```
found it!
```

Iterating through a list

Again, we've actually already done this. It's as simple as using a `for` loop:

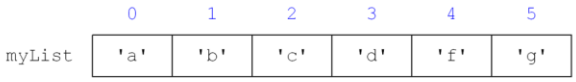
```
myList = ["Joe", "Sally", "George", "Mike"]
for name in myList:
    print "Hello", name
```

Code output:

```
Hello Joe
Hello Sally
Hello George
Hello Mike
```

List slicing

Sometimes you may want to extract a certain subset of a list.



Syntax: `list[begin:end]` returns from index `begin` to `end-1`

```
>>> myList = ['a', 'b', 'c', 'd', 'f', 'g']
>>> myList[2:] #get from 2 to the end
['c', 'd', 'f', 'g']
>>> myList[:4] #get from the beginning to 3
['a', 'b', 'c', 'd']
>>> myList[2:4] #get from 2 to 3
['c', 'd']
```

Side note: indexing strings like lists

Strings are NOT lists. But we can index into strings like we do lists:

```
>>> name = "Sarah"
>>> name[0]
'S'
>>> name[-1]
'h'
>>> name[1:4]
'ara'
```

Side note: indexing strings like lists

Strings are immutable (cannot be changed), so none of these operations are allowed:

```
>>> name = "Sarah"

>>> name[0] = "T"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> name.append("s")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'append'
```

Yes, strings really aren't allowed to be changed. Whenever we've "changed" strings in the past, what we really did was overwrite the variable holding the string with a new version of the string. This is a subtle but important distinction.

Useful list functions

Lists come with several other helpful functions:

- `.sort()` - sorts **in place** (overwrites the list). Can sort both strings and numerical data.
- `.reverse()` - reverses order of items, **in place**
- `.index(element)` - returns index of the first occurrence of the specified element
- `.remove(element)` - Removes the first occurrence of the specified element. Elements that come after will shift down one index.
- `.insert(value, index)` - insert the value at the specified index. ELements that come after that index will shift up one index.
- `.count(element)` - returns the number of times the specified element occurs in the list

Experiment with these on your own to see how they work!

Functions that work on lists

There are also several built-in Python functions that work on lists:

- `len(list)` - returns the total number of elements in the list
- `max(list)` - returns the element in the list with the largest value
- `min(list)` - returns the element in the list with the smallest value
- `sum(list)` - returns the sum of the elements of the list

2. File parsing with `.split()`

The situation

- You have a file with multiple columns separated by tabs, commas, etc
- You want to extract certain columns of data to analyze.
- How can you do this in Python?

`.split()`

- This functions splits a string into a list based on a delimiter.
- The delimiter can be anything you want, but usually it'll be a tab, space, or comma.
- This effectively lets you chop up a file into columns!

.split()

Purpose:

Splits a string every time it encounters the specified delimiter. If no delimiter is given, splits on whitespace (spaces, tabs, and newlines). The delimiter is not included in the output. If *maxsplit* is given, splits no more than *maxsplit* times. Returns a list.

Syntax:

```
result = string.split([delimiter[,maxsplit]])
```

Example:

```
>>> sentence = "Hello, how are you today?"
>>> sentence.split()
['Hello,', 'how', 'are', 'you', 'today?']
```

Notice that the spaces are removed!

More examples

```
>>> sentence = "Hello, how are you today?"
```

```
>>> sentence.split(',')
['Hello', ' how are you today?']
```

Notice that now the comma is removed, but spaces are not!

```
>>> sentence.split(None, 2)
['Hello,', 'how', 'are you today?']
```

maxsplit must always be the second parameter. So if we don't want to specify a delimiter, we can put `None` instead as a placeholder.

Why is `.split()` important?

This is perhaps the single most useful tool for parsing a text file (for what I do, anyway).

Let's take a look at a real-life example.

A more realistic example: parsing a data file

A data file organized in rows and columns (data "table") can be easily parsed using a combination of a `for` loop and `.split()`.

What does "parsing" a data file mean? Basically, it means breaking it down into meaningful pieces--whatever that may be.

A more realistic example: parsing a data file

A data file organized in rows and columns (data "table") can be easily parsed using a combination of a `for` loop and `.split()`.

Example data file:

knownGene	Gene	InitCodon		DistCDS	Frame	InitContext	CDSLen	PeakSt	PeakWidth		#Reads	PeakScore	Codon	Product
uc007afd.1	Mrpl15	248	79	1	AATATGG	15	247	2	368	2.61	aug	internal-out-of-frame		
uc007afh.1	Lypla1	36	5	0	AACATGT	225	34	4	783	3.27	aug	n-term-trunc		
uc007afi.1	Tcea1	28	-24	0	GGCTTGT	325	27	3	446	1.43	nearcog	n-term-ext		
uc007afi.1	Tcea1	100	0	0	GCCATGG	301	99	3	3852	3.79	aug	canonical		
uc007afn.1	Atp6v1h	100	-13	-1	GCTATCC	10	99	3	728	0.77	nearcog	uorf		
uc007afn.1	Atp6v1h	149	3	0	AAGATGG	480	147	3	1407	1.36	aug	n-term-trunc		
uc007agb.1	Pcmtd1	120	-97	-1	GCGCTGG	45	119	3	65	0.75	nearcog	uorf		
uc007agb.1	Pcmtd1	265	-49	0	GCGCTGC	42	264	3	133	0.86	nearcog	uorf		
uc007agb.1	Pcmtd1	412	0	0	GTCATGG	357	411	3	246	1.60	aug	canonical		
uc007agb.1	Pcmtd1	737	108	1	ATCATGG	44	735	3	93	2.37	aug	internal-out-of-frame		
uc007agb.1	Pcmtd1	890	159	1	AGTATGA	17	889	2	87	1.32	aug	internal-out-of-frame		
uc007agk.1	Rrs1	25	-19	0	GTAGTGG	10	25	1	927	1.52	nearcog	uorf		

A more realistic example: parsing a data file

Let's say I just want to extract the 6th column of each row (in this case, the initiation context for each start site).

Code:

```
inFile = "init_sites.txt"
input = open(inFile, 'r')
input.readline() #skip header

for line in input:
    line = line.rstrip('\n')
    data = line.split() #splits line on tabs
    print data[5] #6th column = index 5

input.close()
```

A more realistic example: parsing a data file

Let's say I just want to extract the 6th column of each row (in this case, the initiation context for each start site).

Code:

```
inFile = "init_sites.txt"
input = open(inFile, 'r')
input.readline() #skip header

for line in input:
    line = line.rstrip('\n')
    data = line.split() #splits line on tabs
    print data[5] #6th column = index 5

input.close()
```

Output:

```
AATATGG
AACATGT
GGCTTGT
GCCATGG
GCTATCC
AAGATGG
GCGCTGG
GCGCTGC
GTCATGG
ATCATGG
AGTATGA
GTAGTGG
```

Appendix

Nested lists

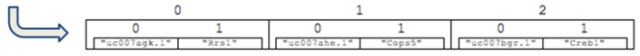
List comprehensions

Examples of .insert() and .remove()

Nested lists

A list can hold pretty much anything, including other lists:

```
>>> geneList = [["uc007agk.1", "Rrs1"], ["uc007ahe.1", "Cops5"],
["uc007bgr.1", "Creb1"]]
```




```
>>> geneList[1]
['uc007ahe.1', 'Cops5']
>>> geneList[1][0]
'uc007ahe.1'
```

You can access individual items in a list of lists using double indexing: `list[index][subindex]`

List comprehensions (advanced)

A list comprehension is just a quick, concise way of performing operations on the elements of a list. Returns a new list with the modified elements.

Syntax:

```
newList = [expression for item in list if condition]
```

Example:

```
>>> myList = [1, 2, 3, 4, 5]
>>> newList = [i * 2 for i in myList]
>>> newList
[2, 4, 6, 8, 10]
>>> newList = [i * 2 for i in myList if i > 3]
>>> newList
[8, 10]
```

List comprehensions (advanced)

Almost any function can be used as the *expression* part:

```
>>> myList = ["Joe", "Sally", "George", "Mike"]
>>> [len(i) for i in myList]
[3, 5, 6, 4]
>>>
>>> [i.upper() for i in myList]
['JOE', 'SALLY', 'GEORGE', 'MIKE']
>>>
>>> [(i == "George") for i in myList]
[False, False, True, False]
>>>
>>> [print(i) for i in myList]
  File "<stdin>", line 1
    [print(i) for i in myList]
          ^
SyntaxError: invalid syntax
```


Why doesn't this work?
The *expression* must return something that can be assigned to a list, which `print` does not do.

Inserting into a list: `.insert()`

Purpose:

Insert new element at specified index. All elements after will be pushed back one index.

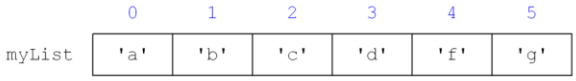
Syntax:

```
list.insert(index, newElement)
```

Example:

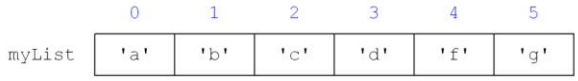
```
>>> myList = [2, 4, 6, 8]
>>> myList.insert(1, "hi!")
>>> print myList
[2, 'hi!', 4, 6, 8]
```

Practice with adding to lists



How do I insert an 'e' between the 'd' and 'f'?

Practice with adding to lists



How do I insert an 'e' between the 'd' and 'f'?

Answer:

```
myList.insert(4, 'e')
```

Remove element from list: `.remove()`

Purpose:

Removes the first occurrence of the specified element. Elements that come after will be moved up one index.

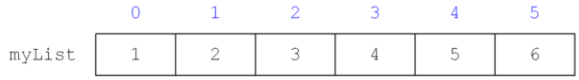
Syntax:

```
list.remove(element)
```

Example:

```
>>> myList = [22, 44, 66, 88]
>>> myList.remove(44)
>>> print myList
[22, 66, 88]
```

Practice with removing from lists



What will this code print?

```
myList.remove(4)
print myList
```

Practice with removing from lists

What will this code print?

```
myList.remove(4)
print myList
```

Answer:

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