Week 6: Lists

Course Website: cs.uoregon.edu/Classes/15U/cis122

Welcome Back

Only a week and a half of class remaining!

We've come a long way...

Types	Functions	Flow Control	Keywords
int	print()	branching	def
float	type()	loops	return
string	help()	break	None
bool	min()/max()	continue	import
	int()/float()	if/elif/else	for
	round()	True/False	in
	len()		while
	Input()		
	list()		

Lists

We're ready to explore our first data structure: the list. Python has 4 built-in data structures:

•Lists []

•Tuples ()

•Sets {}

•Dicts {a:b}

Lists are heterogenous mutable collections of ordered data

Tuples are heterogenous immutable collections of ordered data

Sets are heterogenous mutable collections of unordered unique data

Dicts are heterogenous mutable unordered mappings of unique keys to values

But, What do these terms mean?

Heterogenous- can have different types of data in them (strings, ints, floats, lists, bools…all in the same list if you want)

Mutable- Something that can be changed, more on this later.

Ordered- objects added to the list in a certain order stay in the same order when the list is examined.

Why do we need collections?

Let's say you wanted to write a function to check passwords against a list of too common passwords.

```
def pswrd_check(password):
        if password == "password":
                return False
        elif password == "abc123":
                return False
        elif password == "":
                return False
        ...

        else:
                return True
```

How do we update this code to account for a new common password?

An Improvement

Here’s a better working code

```
common_password = ["abc123", "password", ""]

def pswrd_check(password):
        if password in common_passwords:
                return False
        else:
                return True

def update_commons(new_password, add_or_subtract):
        if add_or_subtract:
                common_passwords.append(password)
        else:
                common_password.remove(password)
```

List Basics

- an empty list can be created by using the [] brackets like this:
  myLst = [ ]
- similarly a list can also be created with elements in it already:
  myLst = [1, "two", 3] notice elements are separated by commas

- Having created a list we can add elements to it with append() **myLst.append("four")**

- insert() allows you to place an item at a certain place in a list elements in a list have positions and can be retrieved from the list with them: **myLst[1]** will grab the second element from a list (remember zero indexing!)

- Note- this is exactly the slicing we used with strings. You can do all the same tricks (negative numbers, slicing out sequences of elements)

List: Keywords

+ concatenates two lists

myLst = [1] + ["two", True]

+= works as expected, - does not work with lists

* works as with strings, repeating a list an integer number of times

myLst * 2

*= works as expected, / does not work with lists

len() is a built in function that returns the number of elements in the list **len (myLst)**

min and **max** are built in functions that return the smallest and largest elements in the list using > and < operations min(myLst) max(myLst)

List: Keywords

- **in** tests if any element of the list matches the criteria

  "two"in myLst
- **del()** destroys a list

  del(myLst)
- **sum()** will add up all elements of a list using + operators

  sum(myList)
- **count()** is a list function that counts the number of instances of the argument in the list

  myLst.count("two")

- **index()** is a list function that finds the index of the first instance of the argument in the list

  myLst.count(True)

Why are Lists Different?

Lists work differently than everything else we've dealt with so far because they are mutable, which means they can change in place.

…which means what?

Refresher

Remember before we said that variables don't change value unless you have an assignment statement? i.e.

```
x = 3
x + 1
print(x)
```

prints out 3 not 4

This was true for ints, floats, strings, bools…

but this isn't (always) true for lists!

My life is a lie...

ints, floats, strings, bools are all immutable, which means they can't change.
but that's stupid, of course they can change:

```
x = "foo"
x += "bar"
print(x)
```

right? Didn't we just change a string from "foo" to "foobar"?

Well…no.

A look under the hood using id()

Id() returns an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id( ) value.

Here's where it's really pretty useful. i.e.

```
x = 1000000
id(x)
x += 1
id(x)
```

We didn't change the value of the int that x points toward. What we did was store a new int and made x point at it. The new int lives at a very different address from the old int. Same thing with strings and floats.

Hello Alice...

Try the same thing with a list. i.e.

```
x = [1]
id(x)
x += [2]
id(x)
```

The lists are in the same place because we didn't create a new list, we changed the old list in place (i.e. where it was).

So, what?

Here's (one reason) why it matters.

```
x = [1]
y = x
x += [2]
print(x)
print(y)
```

Why did that happen?

When we set y = x we didn't create a new copy of the list, we just pointed y at the same copy. So when we modified x we modified y.

Completely different than

```
x = 1
y = x
x+= 1
print(x)
print(y)
```

Why not just create a new list?

Why don't we just make a new copy of the list like we do with ints and floats, strings, and bools?

Lists are often very large, and take lots of memory. You don't make copies of them unless you need to.

BTW if you do need to you can with either of these commands:

y = x.copy() or

y = x[:] (slicing creates a new list)

Some observations worth noting

So you need to understand what operations create new lists and which operations change a list in place.

concatenation creates a new list:

```
 x = []
y = x
x + [1]
print(x)
print(y)
```

meanwhile append() modifies a list in place:

```
x = []
y = x
x .append(1)
print(x)
print(y)
```

Spot the error

What will this give us?

```
x = [1, "two"]
x = x .append(3)
print(x)
```

Spot the error

What will this give us?

```
x = [1, "two"]
x = x .append(3)
print(x)
```

append() changes a list in place, and hence it returns None. When you set x = x.append(3) you nuke your list and replace it with the value None. You probably didn't want to do that.

Another observation

- If a function sorts in-place then it will return None (because it did its work on the list itself)

- Similarly, if a function does not work on the list in-place then it will return a new list (because otherwise you couldn't use it)

Cranial Explosions Expected

...or the usual *groan*

Two ways to sort a list:

```
myLst = [2, 1, 6, 0]
#method 1
print( sorted(myLst) )
print( myLst )
#method 2
myLst.sort()
print(myLst)
```

notice one of these returns a new list and one sorts in place.

Moving on..

We covered most of the basics about lists and some of their tricky aspects, now we’ll look at some useful aspects of lists

The split

You can make lists out of strings quite easily with the string split() method

```
x = "Hello, my name is Gautam!"
y = x.split()
print(y)
```

So that's kind of cool.

Note: So by default the split() would assume split on '<space> ' This is why, while trying to split a string on an empty grade separator we would not get individual characters; for this the list() convertor works better

More on the split

```
x = "Hello, my name is Gautam!"
y = x.split()
print(y)
```

What's more, by splitting the string up into pieces and storing them in a list it becomes easy to then loop through the pieces

```
count = 0
for item in y:
    if y.islower():
        count += 1
print( count )
```

split()

We don't have to split on white space.

```
x = "Hello, my name is Gautam!"
y = x.split(",")
print(y)
```

splitting on commas is particularly useful when you are looking at .csv files (comma separated values).

We'll talk tomorrow about handling text files including csv files with Python.

A few corrections

I had said that the list method .append returns either true, false or None. In actuality it returns None when it changes the list in place.

The other thing is, split() can take 2 arguments at most and the delimiter is checked using pattern matching.For more info: https://docs.python.org/3.4/library/stdtypes.html?highlight=string%20methods#str.split

The .strip() method on the other hand looks for all possible combinations of a regular expression.

More on strip() later..

Changing values in a list

What if we wanted to double each item of a list? Will this work:

```
example = [1, 2, 3, 4]
for item in example:
    item = item * 2
```

let's look at it.

Changing Values

Well that sucked.

What about this:

```
for i in range( len(example) ):
        example[i] = example[i] * 2
        # or
        #example[i] *= 2
```

Nested loops with Lists

I want to write the ultimate popular sci-fi show but I'm not that creative, so I've written a program to help me come up with a title based on previous examples.

```
def mashup():
    firsts = ["fire", "battle", "star", "far"]
    seconds = ["wars", "trek", "star", "fly", "scape"]
    for afirst in firsts:
        for asecond in seconds:
            print(afirst + asecond)
```

Some more nesting

I have a list of names and I want to just get first initials. How hard is that?

Not very.

```
def initials_from_list(alist):
    for i in range( len(alist) ):
        print( alist[i][0] )
```

But, why did this happen?

Sequenception!

Some Problems

I have a list of names and I want to just get first initials, but now the list consists of first names followed by a space and then the last name all as a single string.

Moaaaar Difficulty!

Lets make this slighter harder..

I have a list of names and I want to just get first initials, but now the list consists of first names followed by a space and then the last name all as a single string. What if instead of printing we want to return a list of the initials?

Text Files

Python with it’s rich library makes accessing files really easy. We’ll look at a couple of commands that will allow us to use Python’s great interface

open()

The open() method is the most well known method to access files in python.

open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

we will only require to use the first 2 arguments i.e.

Path of the file and,

the mode.

What do you think are the different modes?

Reading files

We can easily open a file for reading like this:

fin = open("afile.txt", "r")

"afile.txt" is the name of the file to be opened.

"r" indicates we are opening the file in read only mode.

What do you think is the type of fin?

An important point to note regarding addressing

There are two ways to refer to a file address/location

- Relative and,
- Absolute

For this we must delve a little deeper.

the os package allows to access a host of methods that can interact with the operating system

type the following:
import os
os.getcwd()

What do you think cwd stands for?

This is important with regards to saving/accessing files.

Reading Files

To actually read the file we have several options.

- **fin.read(size)**

  which reads some quantity of data and returns it as a string or bytes object. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned
- **fin.readline()**

  reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.
- **fin.readlines()**
  reads every line and puts each as a separate string into a list

- For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

  ```
  for line in f:
          print(line)
  ```

Pointers

To understand how files are read you have to understand that Python maintains a pointer that keeps track of where you are in a file. To begin with this is set at the start of the file.

Writing to Files

Opening a file for writing is easy too:

fout = open("afile.txt", "w")

or,

fout = open("afile.txt", "a")

"afile.txt" is the name of the file to be opened.

"w" indicates we are opening the file in write mode.

The pointer is at the start of the file.

"a" indicates we are opening the file in append mode. The pointer is at the end of the file.

Writing to Files

Once the file is open writing is as easy as:

```
fout.write("This is some text to write")
```

or,

```
txt = "Here's some text"
fout.write(txt)
```

and then,

```
fout.close()
```

close(), shocker, closes the file and makes the changes.