

CS 134:

Lists and Loops

Announcements & Logistics

- **Homework 3** is out on GLOW, due Monday 10 pm
- **Lab 1** graded feedback was released on Wed
 - Any problems?
- **Lab 3** will be released today at noon
 - Watch pre-lab video with your herd and discuss before lab
 - Lab 3 is a collection of word puzzles: can use our newly acquired knowledge of strings, functions and loops to solve them

Do You Have Any Questions?

Last Time

- Started discussing *sequences* in Python
 - Focused on **strings** (sequences of characters)
 - Discussed *slicing* and *indexing* of strings
 - Learned about **in** operator to test membership:
 - Note: there is also a **not in** operator
 - Also learned about string methods **.lower()** and **.upper()**
 - There are also string methods **.islower()** and **.isupper()** that return True if string is in lowercase/uppercase, else False
- (Briefly) Introduced **for loops** as a mechanism to iterate over sequences

Today's Plan

- Discuss for loops in more detail
- Introduce a new sequence: **Lists**
 - Apply indexing, slicing, **in** operator to lists
- Build a collection of functions that iterate over lists and strings
- Build a module for working with sequences

Recap: Iterating with **for** Loops

- The **loop variable** (char and var in the examples below) takes on the value of each of the elements of the sequence one by one

```
for var in seq:  
    # loop body  
    (do something)
```

```
# simple example of for loop
```

```
word = "Williams"
```

```
for char in word:  
    print(char)
```

```
W  
i  
l  
l  
i  
a  
m  
s
```

Recap: countVowels

- **Problem:** Write a function `countVowels()` that takes a string `word` as input, counts and returns the number of vowels in the string.

```
def countVowels(word):  
    '''Returns number of vowels in the word'''  
    pass
```

```
>>> countVowels('Williamstown')
```

```
4
```

```
>>> countVowels('Ephelia')
```

```
4
```

(Bad) Attempt with Conditionals

- Using conditionals as shown is repetitive and does not generalize to arbitrary length words
- Note that `val += 1` is shorthand for `val = val + 1`

```
In [35]: word = 'Williams'
counter = 0
if isVowel(word[0]):
    counter += 1
if isVowel(word[1]):
    counter += 1
if isVowel(word[2]):
    counter += 1
if isVowel(word[3]):
    counter += 1
if isVowel(word[4]):
    counter += 1
if isVowel(word[5]):
    counter += 1
if isVowel(word[6]):
    counter += 1
if isVowel(word[7]):
    counter += 1
print(counter)
```


Counting Vowels Revisited

- Let's use a for loop to finish implementing our `countVowels()` function correctly

```
def countVowels(word):  
    '''Takes a string as input and returns  
    the number of vowels in it'''  
  
    count = 0 # initialize the counter  
  
    # iterate over the word one character at a time  
    for char in word:  
        if isVowel(char): # call helper function  
            count += 1  
    return count
```

Count is an **accumulator** variable, since we accumulate the value as we go through the loop.

Counting Vowels: Tracing the Loop

- How are the local variables updated as the loop runs?

```
def countVowels(word):
```

```
    '''Takes a string as input and returns the number  
    of vowels in it'''
```

```
    count = 0
```

```
    for char in word:
```

```
        if isVowel(char):
```

```
            count += 1
```

```
    return count
```

Loop variable

countAllVowels('Boston')

word

'Boston'

count

~~0~~ 1 2

char

~~'B'~~ ~~'o'~~ ~~'s'~~ ~~'t'~~ ~~'o'~~ 'n'

Exercise: vowelSeq

- Define a function **vowelSeq()** that takes a string **word** as input and returns a string containing all the vowels in word in the same order as they appear. (Hint: we can use `isVowel()` from last class)

```
def vowelSeq(word):  
    '''returns the vowel subsequence in word'''  
    pass
```

```
>>> vowelSeq("Chicago")
```

```
"iao"
```

```
>>> vowelSeq("protein")
```

```
"oei"
```

```
>>> vowelSeq("rhythm")
```

```
""
```

Exercise: vowelSeq

- Define a function **vowelSeq()** that takes a string **word** as input and returns a string containing all the vowels in word in the same order as they appear. (Hint: we can use `isVowel()` from last class)

```
def vowelSeq(word):  
    '''returns the vowel subsequence in word'''  
    vowels = "" # accumulation variable  
    for char in word:  
  
    return vowels
```

Exercise: vowelSeq

- Define a function **vowelSeq()** that takes a string **word** as input and returns a string containing all the vowels in word in the same order as they appear. (Hint: we can use `isVowel()` from last class)

```
def vowelSeq(word):  
    '''returns the vowel subsequence in word'''  
    vowels = "" # accumulation variable  
    for char in word:  
        if isVowel(char): # if vowel  
            vowels += char # accumulate characters  
    return vowels
```

Moving on: Lists

- **Lists** are another type of **sequence** in Python
- Definition: A list is a comma separated sequence of values
- Unlike strings, which can *only contain characters*, lists can be collections of **heterogenous** objects (strings, ints, floats, etc)
- Today we'll focus on **iterating** over lists (i.e., looking at the elements sequentially) using for loops
- Next week we'll focus on manipulating and using lists to store dynamic sequences of objects

Lists

- Lists are:
 - **Comma separated sequences** of values
 - **Heterogenous** collections of objects
 - **Mutable** (or “changeable”) objects in Python. In contrast, strings are immutable (they cannot be changed).
 - We will discuss mutability in more detail soon!

```
In [1]: # Examples of various lists:

wordList = ['What', 'a', 'beautiful', 'day']
numList = [1, 5, 8, 9, 15, 27]
charList = ['a', 'e', 'i', 'o', 'u']
mixedList = [3.145, 'hello', 13, True] # lists can be heterogeous
```

```
In [2]: type(numList)
```

```
Out[2]: list
```

Operations on Sequences

- We already saw several string operators and functions last time
- Most of these apply to lists as well
- We can do the following on lists:
 - Indexing elements of lists using `[]`
 - Using `len()` function to find length
 - Slicing lists using `[:]`
 - Testing membership using `in/not in` operators
 - Concatenation using `+`

Operations on Sequences

```
In [1]: wordList = ['What', 'a', 'beautiful', 'day']  
wordList[3]
```

```
Out[1]: 'day'
```

```
In [2]: wordList[-1]
```

```
Out[2]: 'day'
```

```
In [3]: len(wordList)
```

```
Out[3]: 4
```

```
In [4]: nameList = ["Aamir", "Beth", "Chris", "Daxi", "Emory"]
```

```
In [5]: nameList[2:4]
```

```
Out[5]: ['Chris', 'Daxi']
```


Membership in Sequences

- Recall: The **in** operator in Python is used to test if a given sequence is a subsequence of another sequence; returns True or False

```
In [20]: nameList = ["Anna", "Beth", "Chris", "Daxi", "Emory", "Fatima"]
```

```
In [28]: "Anna" in nameList # test membership
```

```
Out[28]: True
```

```
In [30]: "Jeannie" in nameList
```

```
Out[30]: False
```

Sequences: **not in** operator

- The **not in** operator in Python returns True if and only if the given element is **not** in the sequence

```
In [20]: nameList = ["Anna", "Beth", "Chris", "Daxi", "Emory", "Fatima"]
```

```
In [28]: "Anna" in nameList # test membership
```

```
Out[28]: True
```

```
In [30]: "Jeannie" in nameList
```

```
Out[30]: False
```

```
In [31]: "Jeannie" not in nameList # not in returns true if el not in seq
```

```
Out[31]: True
```

```
In [33]: "a" not in "Chris"
```

```
Out[33]: True
```

Strings to Lists: `split()`

- It is often useful to be able to convert strings to lists, and lists to strings.
- The `split()` method splits strings at “spaces” (the default separator) and returns a list of (sub)strings
- Can optionally specify other **delimiters** as well

```
In [5]: phrase = "What a lovely day"
```

```
In [6]: phrase.split()
```

```
Out[6]: ['What', 'a', 'lovely', 'day']
```

```
In [7]: newPhrase = "What a *lovely*      day!"  # multiple spaces or punctuations dont matter
```

```
In [8]: newPhrase.split()
```

```
Out[8]: ['What', 'a', '*lovely*', 'day!']
```

```
In [9]: commaSepSpells = "Impervius, Portus, Lumos, Reducio, Protego" #comma separated strings
```

```
In [10]: commaSepSpells.split(',')
```

```
Out[10]: ['Impervius', ' Portus', ' Lumos', ' Reducio', ' Protego']
```

List to Strings: `join()`

- Given a list of strings, the `join()` string method, when applied to a character **char**, concatenates the strings together with the character **char** between them

```
In [11]: wordList = ['Everybody', 'is', 'looking', 'forward', 'to', 'the', 'weekend']
```

```
In [12]: '*' .join(wordList)
```

```
Out[12]: 'Everybody*is*looking*forward*to*the*weekend'
```

```
In [13]: '_' .join(wordList)
```

```
Out[13]: 'Everybody_is_looking_forward_to_the_weekend'
```

```
In [14]: ' ' .join(wordList)
```

```
Out[14]: 'Everybody is looking forward to the weekend'
```

Looping over Lists

- We can loop over lists the same way we loop over strings
- As before, the **loop variable** iteratively takes on the values of each item in the list, starting with the 0th item, then 1st, until the last item
- The following loop iterates over the list, printing each item in it

```
In [15]: numList = [0, 2, 4, 6, 8, 10]
```

```
In [16]: for num in numList:  
         print(num)
```

```
0  
2  
4  
6  
8  
10
```

Exercise: countItem

- Let's write a function `countItem()` that takes as input a sequence `seq` (can be a string or a list), and an element `el`, and returns the number of times `el` appears in the sequence `seq`.

```
def countItem(seq, el):  
    """Takes seq as input, and returns the number of times  
    el appears in seq"""  
    pass
```

Exercise: countItem

- Let's write a function `countItem()` that takes as input a sequence `seq` (can be a string or a list), and an element `el`, and returns the number of times `el` appears in the sequence `seq`.

```
def countItem(seq, el):  
    """Takes seq as input, and returns the number of times  
    el appears in seq"""  
    count = 0 # initialize counter  
  
    for item in seq:  
        if item == el: # if this item matches el  
            count += 1 # increment counter  
        # else do nothing, go to next item  
    return count
```

Another accumulator
variable!

Exercise: wordStartEnd

- Write a function that iterates over a given list of words `wordList`, returns a (new) list containing all the words in `wordList` that start and end with the same letter (ignoring case).

```
def wordStartEnd(wordList):  
    '''Takes a list of words wordList and returns a list  
    of all words in wordList that start and end with the same letter'''  
    pass
```

```
>>> wordStartEnd(['Anna', 'banana', 'salad', 'Rigor', 'tacit', 'hope'])  
['Anna', 'Rigor', 'tacit']  
>>> wordStartEnd(['New York', 'Tokyo', 'Paris'])  
[]  
>>> wordStartEnd(['*Hello*', '', 'nope'])  
['*Hello*']
```


Exercise: wordStartEnd

- **Step by step approach (organize your work):**
 - Go through every word in wordList
 - Check **if word starts and ends at same letter**
 - If true, we need to “collect” this word (remember it for later!)
 - Else, just go on to next word
 - Takeaway: need a new list to **accumulate** desirable words
- **Break down bigger steps (decomposition!)**
 - If word starts and ends at same letter:
 - Can do this using string **indexing**
 - Think about **corner cases**: what if string is empty? what about case?

Exercise: wordStartEnd

- Write a function that iterates over a given list of words `wordList`, returns a (new) list containing all the words in `wordList` that start and end with the same letter (ignoring case).

```
def wordStartEnd(wordList):  
    '''Takes a list of words and returns a list of words in it  
    that start and end with the same letter'''  
    # initialize accumulation variable (of type list)  
    result = []  
    for word in wordList: # iterate over list  
  
        #check for empty strings before indexing  
        if len(word) != 0:  
            if word[0].lower() == word[-1].lower():  
                result += [word] # concatenate to resulting list  
    return result # notice the indentation of return
```


Exercise: palindromes

- Write a function that iterates over a given list of strings `sList`, returns a (new) list containing all the strings in `sList` that are palindromes (i.e., read the same backward and forward).

```
def palindromes(sList):  
    '''Takes a list of words wordList and returns a list  
    of all words in wordList that start and end with the same letter'''  
    pass
```

```
>>> palindromes(['Anna', 'banana', 'kayak', 'rigor', 'tacit', 'hope'])  
['Anna', 'kayak']  
>>> palindromes(['1313', '1110111', '0101'])  
['1110111']  
>>> wordStartEnd(['Level', 'Stick', 'Gag'])  
['Level', 'Gag']
```

Exercise: palindromes

- **Step by step approach (organize your work):**
 - Go through every word in wordList
 - Check if **word is same forward and backwards**
 - If true, we need to collect this word (remember it for later!)
 - Else, just go on to next word
 - Takeaway: need a new list to **accumulate** desirable words
- **Break down bigger steps (decomposition)**
 - If word is same forward and backwards:
 - Can do using slicing with optional step
 - Think about **corner cases**: what is string is empty? what about case?

Exercise: palindromes

- Write a function that iterates over a given list of strings `sList`, returns a (new) list containing all the strings in `sList` that are palindromes (i.e., read the same backward and forward).

```
def palindromes(sList):  
    '''Takes a list of words and counts the  
    number of words in it that start and end  
    with the same letter'''  
    # initialize accumulation variable (of type list)  
    result = []  
    for word in sList: # iterate over list  
        wLower = word.lower()  
        if wLower[::-1] == wLower:  
            result += [word] # concatenate to resulting list  
    return result
```


Putting Our Functions to Use

- We have written several helpful functions for working with sequences
- We can collect them in a module called **sequenceTools**
 - Then, whenever we want to use our functions, we can import this module
- **pydoc3 sequenceTools** gives an overview of all functions in it

```
cs134 — less · pydoc3 sequenceTools — 67x23
NAME
    sequenceTools - This module contains different functions that i
    terate over sequences such as strings and list.

DESCRIPTION
    You can query this documentation using pydoc3 sequenceTools fro
    m the terminal

FUNCTIONS
    countItem(seq, el)
        Takes as input a sequence seq (can be a string or a list),
        and an element el and returns the number of times el appear
        s
        in the sequence seq.

    countVowels(word)
        Returns number of vowels in the given word.

    palindromes(sList)
        Takes a list of words and counts the
        number of words in it that start and end
        with the same letter
    :
```


Importing a Module

- If the variable starts/ends with “__” it is a special variable in Python
 - Where have we already seen this?
- `__all__` is another special variable
 - A list of strings of function names (or other public objects) that are intended to be imported when the user types:

```
from moduleName import *
```

- Note: Any specific function/variable/etc. in the module can also be explicitly imported as:

```
from moduleName import explicitVariableName
```

Testing Functions: Doctests

- We have already seen two ways to test a function (what were they??)
- Python's **doctest** module allows you to embed test cases and expected output directly into a function's docstring
- To use the doctest module, we must import it using:
from doctest import testmod
- To make sure the test cases are run when the program is run as a script from the terminal, we then need to call **testmod()**.
- To ensure that the tests are not run in interactive Python or when the module is imported, we place the command within a guarded if block:
if __name__ == '__main__':

Testing Functions: Doctests

```
def isVowel(char):  
    """Takes a letter as input and returns true if and only if it is a vowel.  
    >>> isVowel('e')  
    True  
    >>> isVowel('U')  
    True  
    >>> isVowel('t')  
    False  
    >>> isVowel('Z')  
    False  
    """  
    return char.lower() in 'aeiou'
```

```
if __name__ == '__main__':  
    # the following code tests the tests in the docstrings ('doctests').  
    # as you add tests, re-run this as a script to test your work  
    from doctest import testmod # this import is necessary when testing  
    testmod() # test this module, according to the doctests
```

**Run the doctests only when file is
executed as a script**

Extra ..

Range Function

- When the **range** function is given two integer arguments, and it returns a **range object** of all integers starting at the first and up to, *but not including*, the second
- To see the values included in the range, we can pass it to the list function which returns a **list** of them
- A **list** is a new Python type: stores a sequence of any values, delimited by square brackets, and separated by commas

```
In [1]: range(0, 10)
```

```
In [2]: range(0, 10)
```

```
Out [2]: list(range(0,3))
```

```
In [3]: list(range(3))    #missing first arg defaults to 0
```

```
Out [3]: [0,1,2]
```

Strings to Lists: `list()`

- It is often useful to be able to convert strings to lists, and lists to strings.
- `list` function when given a string returns a list of characters the string is composed of

```
In [1]: spell = "Avada Kedavra!"
```

```
In [2]: list(spell) # can turn a string into a list of its characters
```

```
Out[2]: ['A', 'v', 'a', 'd', 'a', ' ', 'K', 'e', 'd', 'a', 'v', 'r', 'a', '!']
```

```
In [3]: phrase = "What a lovely day"
```

```
In [4]: wordList = phrase.split()
```

```
In [5]: wordList
```

```
Out[5]: ['What', 'a', 'lovely', 'day']
```


Loops to Repeat Tasks

- Sometimes we might use a loop, not to iterate over a sequence but just to repeat a task over and over. The following loops print a pattern to the screen.

```
for i in range(5): # for loops to print patterns
```

```
    print('$' * i)
```

```
for j in range(5):
```

```
    print('*' * j)
```

```
for _ in range(10):
```

```
    print('Hello World!')
```

\$

\$ \$

\$ \$ \$

\$ \$ \$ \$

*

**

Try this out in interactive python! When loop variable is not needed in body, can use _ as variable

Mutability

Strings are Immutable

- Once you create a string, it cannot be changed!
- All functions that we have seen on strings return a new string and do not modify the original string

Lists are mutable

- Lists are mutable sequences
- As we saw, you can append to a list
- You can modify a list in many other ways: we will see this in the next lecture

Summary: Sequences Operations

Operation	Result
x in seq	True if an item of seq is equal to x
x not in seq	False if an item of seq is equal to x
seq1 + seq2	The concatenation of seq1 and seq2*
seq*n, n*seq	n copies of seq concatenated
seq[i]	i'th item of seq, where origin is 0
seq[i:j]	slice of seq from i to j
seq[i:j:k]	slice of seq from i to j with step k
len(seq)	length of seq
min(seq)	smallest item of seq
max(seq)	largest item of seq

* Concatenation is not supported on range objects

Summary: String Methods

```
word = 'Williams College'
```

```
word.split()
```

```
word.upper()
```

```
word.lower()
```

```
word.replace('iams', 'eslley')
```

```
word.replace('tent', 'eselley')
```

```
newWord = '    Spacey College    '
```

```
newWord.strip()
```

```
myList = ['Williams', 'College']
```

```
' '.join(myList)
```

Returned value

```
['Williams', 'College']
```

```
'WILLIAMS COLLEGE'
```

```
'williams college'
```

```
'Willeslley College'
```

```
'Williams College'
```

```
'Spacey College'
```

```
'Williams College'
```

Remember. none of these operations change/affect the original string, they all return a new string

Lots More String Functions

- `word.find(s)`
 - Return the first (or last) position of string `s` in `word`. Returns `-1` if not found.
- `char.isspace()`
(or `islower`, `isupper`, `isalpha`, `isdigit`, `isalnum`).
 - Returns `True` if `s` is not empty and `s` is composed of white space (or lowercase, uppercase, or alphabetic letters, or digits, or either letters or digits).
- `word.count(s)`
 - Returns the number of (non-overlapping) occurrences of `s` in `word`
- Many more: see `pydoc3 str`

Sorted Function

- The built-in function `sorted` which takes a sequence as input, creates and returns a new list where items are ordered in ascending order.

```
In [1]: numbers = [35, -2, 17, -9, 0, 12, 19]
sorted(numbers)
```

```
Out[1]: [-9, -2, 0, 12, 17, 19, 35]
```

- Notice that the original list is unchanged

```
In [2]: numbers
```

```
Out[2]: [35, -2, 17, -9, 0, 12, 19]
```

Sorted Function on Strings

- Strings can be sorted the same way: the ordering used for the sorting is dictated by the ASCII values of the characters.

```
In [3]: phrase = 'Red Code 1'  
sorted(phrase)
```

```
Out[3]: [' ', ' ', '1', 'C', 'R', 'd', 'd', 'e', 'e', 'o']
```

- Notice that spaces and special characters are first, following by numbers, followed by capital letters, and finally lower case
- You can check the ASCII value of any character using the **ord** function

```
In [4]: ord(' ')
```

```
Out[4]: 32
```

← ASCII value of space

Why Sort Strings?

- Gives us a canonical form, useful to find other strings made up of the same characters!
- Remember that when comparing strings, we should always make sure they are in the same case (which is why we use `.lower()` often)
- Motivating example. **Anagrams.**
 - Finding anagrams of a given word among a list of words
 - What do anagrams have in common?

Dormitory = Dirty room
School master = The classroom
Listen = Silent
Funeral = Real fun