Announcements & Logistics

- **Lab 3** graded feedback was returned this morning
- **Homework 5** due Mon Oct 11 at 10 pm
- No lecture on Mon Oct 11. No lab on Mon Oct 11 or Tue Oct 12
- **Lab 4 Part 2** due Oct 13/14 at 10 pm.
  - Herd meetings and TA /instructor hours as usual (your herd TA will contact you to reschedule if necessary)
  - Highly recommend attending your herd meetings this week!
  - Check course calendar for office hours for Mon/Tue
- **Midterm reminder**: Wed Oct 20th: Slots: 6 - 7:30 pm, 8 - 9:30 pm
- **Midterm review**: Monday, Oct 18th 7 - 8 pm

Do You Have Any Questions?
Last Time

• Learned about writing and appending to files
• Reviewed useful list methods:
  • Methods that don't modify lists: `.index()`, `.count()`
  • Methods that do modify lists:
    • `.append()`, `.extend()`, `.insert()`, `.remove()`, `.pop()`, `.sort()`
• Discussed `sorted()` function: returns a new sorted list
  • Strings sorted using the ASCII values of their characters
• Started discussion on mutability and aliasing in Python
Recap: Sorting Strings

- Strings are sorted based on the **ASCII values** of their characters.
- ASCII stands for “American Standard Code for Information Interchange”
- Common character encoding scheme for electronic communication (that is, anything sent on the Internet!)
- Special characters come first, followed by capital letters, then lowercase.
- Characters encoded using integers from 0–127.
- Can use Python functions **ord()** and **chr()** to work with these:
  - **ord(str)**: takes a character and returns its ASCII value as **int**.
  - **chr(int)**: takes an ASCII value as **int** and returns its corresponding character (**str**).
<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[NULL]</td>
<td>32</td>
<td>20</td>
<td>[SPACE]</td>
<td>64</td>
<td>40</td>
<td>@</td>
<td>96</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>[START OF HEADING]</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>[START OF TEXT]</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>[END OF TEXT]</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>[END OF TRANSMISSION]</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>[ENQUIRY]</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>[ACKNOWLEDGE]</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>[BEL]</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>[BACKSPACE]</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>[HORIZONTAL TAB]</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>[VERTICAL TAB]</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>[FORM FEED]</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>[CARRIAGE RETURN]</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>[SHIFT OUT]</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>[SHIFT IN]</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>[DATA LINK ESCAPE]</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>[DEVICE CONTROL 1]</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>[DEVICE CONTROL 2]</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>[DEVICE CONTROL 3]</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>[DEVICE CONTROL 4]</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>[NEGATIVE ACKNOWLEDGE]</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>[SYNCHRONOUS IDLE]</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>[ENG OF TRANS. BLOCK]</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>[CANCEL]</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>[END OF MEDIUM]</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>[SUBSTITUTE]</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>[ESCAPE]</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>[</td>
<td>123</td>
<td>7B</td>
<td>{</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>[FILE SEPARATOR]</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
<td>124</td>
<td>7C</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>[GROUP SEPARATOR]</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
<td>125</td>
<td>7D</td>
<td>}</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>[RECORD SEPARATOR]</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>[UNIT SEPARATOR]</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>_</td>
<td>127</td>
<td>7F</td>
<td>[DEL]</td>
</tr>
</tbody>
</table>
**sorted()** returns a list

- **sorted()** always returns a new list!
- **Note:** **sorted(string)** returns a sorted list (not string!)

```python
In [1]: sorted('shikha')
Out[1]: ['a', 'h', 'h', 'i', 'k', 's']

In [2]: sorted('jeannie')
Out[2]: ['a', 'e', 'e', 'i', 'j', 'n', 'n']

In [3]: sorted('Hello World')
Out[3]: [', ', 'H', 'W', 'd', 'e', 'l', 'l', 'l', 'l', 'o', 'o', 'r']
```
Today’s Plan

• Briefly discuss while loops
  • Needed for ranked-choice voting on Lab 4 Part 2
• Continue discussion on aliasing in Python: a side-effect of mutability
While Loops
For loops in Python

- **For loops** in Python are meant to iterate directly over a **fixed sequence** of items
  - No need to know the sequence's length ahead of time
- Interpretation of for loops in Python:
  
  ```
  for each item in given sequence:
    (do something with item)
  ```
- This is inherently different from for loops in other languages, such as Java, where you iterate over "indices" and have a stopping condition on the indices (e.g., length of sequence) built into the loop header
- **Takeaway**: For loops in Python are meant to iterate directly over each item of a given iterable object
What If We Don’t Know When to Stop?

• Stopping condition of for loop: **no more elements in sequence**

  ```
  ['A', 'lovely', 'Fall', 'day']
  ```

• What if we don’t know when to stop?
  • Suppose you had to write a program to ask a user to enter a name, repeatedly, until the user enters "quit" in which case you stop asking for input and print “Goodbye"
While Loops

• For loops iterate over a pre-determined sequence and stop at the end of the sequence.

• On the other hand, `while` loops are useful when we don't know in advance when to stop.

• While loop syntax:
  
  ```python
  while (boolean expression evaluates to true):
    # keep repeating the following
    # statements in loop body
  ```

• A while loop will keep iterating until the condition in the parentheses is satisfied (is true) and will halt if the condition fails to hold (is false).

  • Kinda like a repeating if statement.
**While Loops**

```
while loops are a fundamental mechanism for expressing iteration
```

**keyword indicating while loop**

```
while
```

**body of loop = actions to perform if the continuation condition is true**

```
body = statement1, ..., statementN
```

```
while (continuation_condition):
    # keep repeating the following statements in loop body
```

Image Source: (http://cs111.wellesley.edu/spring19)
While Loop Example

- Example of a while loop that depends on user input

```python
prompt = 'Please enter a name (type quit to exit): '
name = input(prompt)

while (name.lower() != 'quit'):
    print('Hi,', name)
    name = input(prompt)
print('Goodbye')
```

- See notebook for more examples
While Loop to Print Halves

- Given a number, keep dividing it until it becomes smaller than 0 and print all the “halves”

```python
def printHalves(n):
    while n > 0:
        print(n)
        n = n//2

printHalves(100)
```
Moving on…

Back to
Mutability and Aliasing
Value vs Identity

- Python is an **object oriented language**
- An **object’s identity** never changes in Python once it has been created; think of it as the object’s **address** in memory
- The `id()` function returns an integer representing an object's identity (or address)
- An **object’s value** is the value assigned to the object when it is created

```
In [1]: num = 5
```
```
In [2]: id(num)
```
```
Out[2]: 4486937008
```

**num**

**identity:** mem address where **5** is stored

**value:** 5
Value vs Identity

• An object’s identity never changes in Python once it has been created; think of it as the object’s address in memory.

• On the other hand, an object’s value can change.

  • Objects whose values can change are called mutable; objects whose values cannot change are called immutable.

```
In [1]: num = 5
In [2]: id(num)
Out[2]: 4486937008
```

Variable names like `num` point to memory addresses of stored value.
Comparing Value vs Identity

- The `==` operator compares the value of an object (i.e., are the contents of the objects the same?)
- The `is` operator compares the identity of two objects (i.e., do they have the same memory address?)
  - `var1 is var2` is equivalent to `id(var1) == id(var2)`

```python
In [1]: num = 5
In [2]: id(num)
Out[2]: 4486937008
```

Variable names like `num` point to memory addresses of stored value

```
num
id: 4486937008
```

Memory address
Mutability in Python

**Strings, Ints, Floats are Immutable**

- Once you create them, their value **cannot** be changed
- All attempts (functions/methods) to change mutable objects create (return) **new objects** and **do not modify** the original object

**Lists are Mutable**

- List values **can** be changed
- We have seen how we can mutate/change what’s in a list in many ways
- The mutability of lists has many implications such as **aliasing**, which can cause trouble if we are not careful!
Ints, Floats are Immutable

In [1]: num = 5
In [2]: id(num)
Out[2]: 4486937008
In [3]: num = num + 1
In [4]: id(num)

What happens behind the scenes?

Will this stay the same?

Attempts to change an immutable object creates a new object
Ints,Floats are Immutable

In [1]: num = 5
In [2]: id(num)
Out[2]: 4486937008
In [3]: num = num + 1
In [4]: id(num)
Out[4]: 4486937040

Identity of ints cannot be changed, num assumes a **new** identity!

Attempts to change an immutable object creates a new object
Strings are Immutable

Attempts to change an immutable object creates a new object

In [1]: word = "Williams"

In [2]: college = word

In [3]: word == college

Out[3]: True

In [4]: print(id(word), id(college))

4518582576 4518582576

In [5]: word is college

Out[5]: True

id: mem addr (4518582576)

“Williams”

variable names point to memory addresses of stored value

Even though word and college have the same identity and value now, if we update one of them, it just assumes a new identity!
Strings are Immutable

Attempts to change an immutable object creates a new object
Sequence Operations Return New Sequences

- The following operations, that can be performed on both lists and strings, and always return a new list/string
  - `sorted(sequence)`: returns a new sorted sequence
  - `[::]` slicing operator: returns a new sliced sequence
  - assignment of a new sequence to a variable
    - `names = 'Shikha and Jeannie'`
    - `myList = [1, 2, 3]`
  - concatenation (+) always creates a new sequence
  - functions like `len()`, accessing an element using an index, etc do not modify the sequence (they just provide information about an existing sequence)
Sequence Methods Return New Sequences

- Any sequence operation, e.g., slicing, that works on strings always creates a new object
  - Slicing a list returns a new list, slicing a string returns a new string
- String methods like `.lower()`, `.upper()` return a new string

```python
In [1]: name = "gryffindor"

In [2]: id(name)
Out[2]: 4574657776
```
Sequence Methods Return New Sequences

- Any sequence operation, e.g., slicing, that works on strings always creates a new object.
- Slicing a list returns a new list, slicing a string returns a new string.
- String methods like `.lower()`, `.upper()` return a new string.

```python
In [1]: name = "gryffindor"

In [2]: id(name)
Out[2]: 4574657776

In [3]: name = name[4:8]

In [4]: id(name)
Out[4]: 4574684720
```
Lists are Mutable

Value of list objects can change, keeping identity the same.

In [1]: myList = [1, 2, 3]

In [2]: id(myList)

Out[2]: 4418551104

In [3]: myList.append(4)

In [4]: id(myList)

Out[4]: 4418551104

Notice: value changes, identity stays the same!
Aliasing: Side Effect of Mutability

- Because list objects **can change**, this leads to a side effect: **aliasing**
- Any assignment or operation that “points” to a list implicitly creates an **alias** (a new name) to the same list.

```python
In [1]: list1 = [1, 2, 3]
list2 = list1

In [2]: list1 is list2
Out[2]: True
```

We are not creating a separate copy, but rather creating a **pointer** to the original list; **list1 is an alias of list2**
Aliasing: Side Effect of Mutability

• Changing the value of `list1` will also change the value of `list2`:
  • They are both names that point to the same list!

```
In [1]: list1 = [1, 2, 3]
    list2 = list1

In [2]: list1 is list2
Out[2]: True

In [3]: list1.append(4)

In [4]: list2
Out[4]: [1, 2, 3, 4]
```
Aliasing: Side Effect of Mutability

- A new assignment to a variable **creates a new list**

```
In [1]: list1 = [1, 2, 3]
    list2 = list1
    myList = [1, 2, 3]

In [2]: # same values?
    myList == list1 == list2

Out[2]: True  # True; values are the same

In [3]: # same identities?
    myList is list1

Out[3]: False  # False; identities are different
```
Understanding Aliasing
Aliasing Examples

In [1]:
nums = [23, 19]
words = ['hello', 'world']
mixed = [12, nums, 'nice', words]

In [2]:
words.append('sky')

In [3]:
mixed

Out[3]:
[12, [23, 19], 'nice', ['hello', 'world', 'sky']]

What is going on here?
**Aliasing Examples**

In [1]:
```
nums = [23, 19]
words = ['hello', 'world']
mixed = [12, nums, 'nice', words]
```
Aliasing Examples

```python
In [2]: words.append('sky')
```
Aliasing Examples

In [1]:
nums = [23, 19]
words = ['hello', 'world']
mixed = [12, nums, 'nice', words]

In [2]:
words.append('sky')

In [3]:
mixed

Out[3]: [12, [23, 19], 'nice', ['hello', 'world', 'sky']]

In [4]:
mixed[1].append(27)

In [5]:
nums

Out[5]: [23, 19, 27]

In [6]:
mixed

Out[6]: [12, [23, 19, 27], 'nice', ['hello', 'world', 'sky']]
Aliasing Examples

```python
In [4]: mixed[1].append(27)
```

```
[23, 19, 27]
['hello', 'world', 'sky']
[12, , 'nice', ]
mixed
```
Aliasing Examples

```
In [9]: def foo(someList):
   ...:     someList.append('*')
   ...:
   ...:     newList = ['#']
   ...:     bar = foo(newList)

In [10]: newList
Out[10]: ['#', '*']

In [12]: def foo(someList):
   ...:     print(id(someList))
   ...:     someList.append('*

   ...:     newList = ['#']
   ...:     print(id(newList))
   ...:     bar = foo(newList)

   ...: print(id(newList))
   ...: bar = foo(newList)

4462680064
4462680064
```
Conclusion

• We **cannot change** the value of **immutable** objects such as strings
  • Aliasing is not an issue!
  • Attempts to change the object just creates a new object

• We **can change** the value of **mutable** objects such as lists
  • Need to be careful about aliasing!
  • Unintended aliases can cause problems...
  • You can create a true (new) copy of a list using slicing:
    `newList = myList[:]`