Lists, Memory Diagrams & Mutable vs. Immutable Sequences

Why Lists (and other sequences)?
Lists (and other sequences) are useful to represent collections, especially where order matters.

- list of all public tweets ever posted on Twitter, in time order
- course information for all Wellesley courses (714)
- Complete works of Maya Angelou:
  - As a single string
  - As a list of books, poems, sentences, verses, words, etc.

Lists: glue for many values

```
# Lists returned from builtin functions and methods
oddcies = range(1,10,2) # [1,3,5,7,9]
lyrics = 'call me on my cell'.split() # ['call', 'me', 'on', 'my', 'cell']
letters = list('happy') # ['h', 'a', 'p', 'p', 'y']

# Literal list definitions
primes = [2, 3, 5, 7, 11, 13, 17, 19]
bools = [1<2, 1==2, 1>2]
houses = ['Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin']
strings = ['ab' + 'cd', 'ma'*4]
counts = [1, 2, 3] + [4, 5]
animalLists = [['fox', 'raccoon'],
               ['duck', 'raven', 'gosling'], [], ['turkey']]

# A heterogeneous list
stuff = [17, True, 'foo', None, [42, False, 'bar']]

# An empty list
empty = []
```

Homogenous, heterogeneous, nested lists

Lists in which all elements have the same type are called **homogeneous**. Most of the lists we'll use will be homogeneous.

```
# List of primes less than 20
[2, 3, 5, 7, 11, 13, 17, 19]
```

Lists can also contain other lists as elements!

```
# List of string lists
[['fox', 'raccoon'], ['duck', 'raven', 'gosling'], [], ['turkey']]
```

Python also allows **heterogeneous** lists in which elements can have different types. In general, you should avoid heterogeneous lists unless you have a good reason to use them. (They make programs harder to reason about.)

```
[17, True, 'Wendy', None, [42, False, 'computer']]
```
How to represent list values: Memory Diagrams [0]

Big # 4: Models

- list slot indices
- list slots
- list indices
- start at 0, not 1

Numbers, booleans, and None are “small enough” to fit directly in variables and list slots.

All other values are drawn outside the variable/list slot, with an arrow pointing to them.

How to represent list values: Memory Diagrams [1]

animalLists = [['fox', 'raccoon'], ['duck', 'raven', 'gosling'], [], ['turkey']]

animalLists = [[['fox', 'raccoon'], ['duck', 'raven', 'gosling'], [], ['turkey']]

List indexing and slicing (review)

In[1]: houses = ['Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin']

In[2]: houses[0]  # List indexing
Out[2]: 'Gryffindor'

In[3]: houses[3]
Out[3]: 'Slytherin'

In[4]: houses[4]  # IndexError Traceback (most recent call last)
              <ipython-input-4-834fac18ce76> in <module>()
              ----> 1 houses[4]
              IndexError: list index out of range

In[5]: houses[-3]  # Negative indexing
Out[5]: 'Gryffindor'

In[6]: mammals = animalLists[0]  # List of lists.

In[7]: mammals[2]
Out[7]: ['raven']

In[8]: mammals[2][1]  # Nested list indexing
Out[8]: 'raven'

Nested list indexing (is not special!)

Write a 1-line Python expression to get 'raven' from animalLists.

Write a 1-line Python expression to get 'turkey' from animalLists.

Challenge: Write two new expressions that also get 'raven' and 'turkey' using different indices than before.
Lists are sequences.

Immutable sequence operations

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

Lists are mutable.

Lists are mutable, meaning that their contents can change over time.

Lists can change in two ways:

1. The element at a given index can change over time. That is, the slot in a list at a particular index behaves as a variable, whose contents can change over time.

2. The length of a list can change over time as new slots are added or removed.

List slot mutability example

```
shoesizes = [8, 8.5, 12.5, 10]

shoesizes[3] = 11.5
```

List slot mutability larger example

```
myList = [17, 3.141, True, None, ['I', 'am', 'Sam'], Circle(50, Point(200, 100))]
```
List slot mutability larger example [1]

The value in any named or numbered box can change over time.
For example, the values in list slots can be changed by assignment.

myList[1] = myList[0] + 6
myList[4][1] = 'was'

append: add a new slot to the end of a list

myList.append(42)

myList[4].append('Adams')

More list mutability

pop
(remove an element from a list)

insert
(adding a new element to a list)

"Aliasing"
(same object stored in multiple variables and slots)
**pop**: remove slot at an index and return its value

- `myList.pop(3) ➞ False` # Indices of slots after 3 are decremented
- `myList[3].pop(2)` # Index of previous slot 3 is decremented
- `myList.pop()` # When no index, last one is assumed

Draw updates:

```plaintext
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>23</td>
<td>True</td>
<td>False</td>
<td></td>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>
```

Final memory diagram:

```plaintext
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>23</td>
<td>True</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**insert**: add a slot, add an index

- `myList.insert(0, 98.6)` # Indices of previous slots 0 and above are incremented
- `myList[4].insert(2, 'not')` # Index of previous slot 2 is incremented

Draw updates:

```plaintext
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>23</td>
<td>True</td>
<td></td>
<td>98.6</td>
</tr>
</tbody>
</table>
```

Final memory diagram:

```plaintext
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>98.6</td>
<td>17</td>
<td>23</td>
<td>True</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList
circ = list2[5]
myList[1][3] = circ
```

Draw updates:

Final memory diagram:

What is the final value of `c[0]`?

```
a = [15, 20]
b = [15, 20]
c = [10, a, b]
b[1] = 2*a[0]
c[1][0] = c[0]
c[0] = a[0] + c[1][1] + b[0] + c[2][1]
```

Lists are mutable. What about strings?

Strings are sequences:

```
In [6]: name = 'Gryffindor'
In [7]: name[2]  # 'y'
In [8]: name[4:8]  # 'find'
In [9]: 'do' in name  # True
```

Mutation operations do not work on strings:

```
In [10]: name[4] = 't'  # what happens?
```

```
TypeError ... name[0] = 't'
```

```
In [11]: name.append('s')  # what happens?
```

```
AttributeError ... name.append('s')
```

```
AttributeError: 'str' object has no attribute 'append'
```
Strings are immutable sequences.

Once you create a string, it cannot be changed.

In[13]: college = 'WELLESLEY'

Once you create a string, it cannot be changed.

In[14]: college.lower()

Out[14]: 'wellesley'

Immutability means that the data is not changed after it is created.

myCollege = college.lower()

Tuples are immutable sequences.

Like strings, tuples support all sequence operations that do not involve mutation.

In[32]: houseTuple = ('Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin')

In[33]: houseTuple[2]

Out[33]: 'Gryffindor'

In[34]: houseTuple[1:3]

Out[34]: ('Hufflepuff', 'Ravenclaw')

In[35]: houseTuple.count('Slytherin')

Out[35]: 1

In[36]: 'Ravenclaw' in houseTuple

Out[36]: True

In[37]: houseTuple * 2 + ('12 Grimmauld Place',)

Out[37]: ('Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin', 'Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin', '12 Grimmauld Place')

Tuples are written as comma-separated values delimited by parentheses.

# A homogeneous tuple of five integers (a 4-tuple)
(5, 8, 7, 1, 3)

# A homogeneous tuple of four strings
('Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin')

# A heterogeneous tuple of three elements (a 3-tuple)
(42, 'Hello', False)

# A pair is a tuple with two elements (a 2-tuple)
(7, 3)

# A tuple with one element must use a comma to avoid being confused with a parenthesized expression
()

# A tuple with 0 values

Mutation operations do not work on tuples.

In [38]: houseTuple[0] = '4 Privet Drive'

---------------------------------------------------------------------------
TypeError...
houseTuple[0] = '4 Privet Drive'
TypeError: 'tuple' object does not support item assignment

In [39]: houseTuple.append('The Shrieking Shack')

---------------------------------------------------------------------------
AttributeError...
houseTuple.append('The Shrieking Shack')
AttributeError: 'tuple' object has no attribute 'append'

In [40]: houseTuple.pop(1)

---------------------------------------------------------------------------
AttributeError...
houseTuple.pop(1)
AttributeError: 'tuple' object has no attribute 'pop'
Conversion between sequence types

The built-in functions `str`, `list`, `tuple` create a new value of the corresponding type.

In [41]: word = "Wellesley"
In [42]: list(word)
Out[42]: ['W', 'e', 'l', 'l', 'e', 's', 'l', 'e', 'y']
In [43]: tuple(word)
Out[43]: ('W', 'e', 'l', 'l', 'e', 's', 'l', 'e', 'y')

In [44]: numbers = range(5, 15, 2)
In [45]: str(numbers)
Out[45]: '[5, 7, 9, 11, 13]'

Tuple Assignment

Suppose `harryInfo` is a tuple of three values:

In [46]: harryInfo = ('Harry Potter', 11, True)

Then we can extract three named values from `harryInfo` by a single assignment to a tuple of three variable names:

In [47]: (name, age, glasses) = harryInfo

This so-called tuple assignment is just a shorthand for three separate assignments:

name = harryInfo[0]
age = harryInfo[1]
glasses = harryInfo[2]

We can now use these names like any other variables:

In [47]: print name.lower(), age + 6, not glasses
harry potter 17 False

Parens are not necessary in a tuple assignment; above, we could also have written:

In [47]: name, age, glasses = harryInfo

Enumerations

When called on a sequence, the `enumerate` function returns a sequence of pairs of indices and values.

In [46]: list(enumerate('boston'))
Out[49]: [(0, 'b'), (1, 'o'), (2, 's'), (3, 't'), (4, 'o'), (5, 'n')]

In [47]: list(enumerate([7, 2, 8, 5]))
Out[47]: [(0, 7), (1, 2), (2, 8), (3, 5)]

In [48]: for (index, char) in enumerate('boston'):
...:     print index, char
0 b
1 o
2 s
3 t
4 o
5 n