Lists, Memory Diagrams & Mutable vs. Immutable Sequences

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

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

Complete works of Maya Angelou:
- As a single string
- As a list of books, poems, sentences, verses, words, etc.

Homogenous, heterogeneous, nested lists
Lists in which all elements have the same type are called homogenous. Most of the lists we'll use will be homogenous.

# 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']]

Lists: glue for many values
# Lists returned from builtin functions and methods
odds = 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', 'y']

# Literal list definitions
primes = [2, 3, 5, 7, 11, 13, 17, 19]
bools = [1<2, l==2, l>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 = []
How to represent list values: Memory Diagrams [0]

- A variable starts at 0, not 1.
- List indices start at 0, not 1.
- List slots work from end.
- Negative indices work from end.
- List indices start at 0, not 1.
- All other values are drawn outside the variable/list slot, with an arrow pointing to them.

- Numbers, booleans, and `None` are "small enough" to fit directly in variables and list slots.

List indexing and slicing (review)

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

- **Indexing:** get one element from the given position (index) in the list.

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

```
In[3]: houses[3]  # List indexing
Out[3]: 'Slytherin'
```

```
In[4]: houses[4]  # IndexError: list index out of range
```

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

```
In[6]: houses[1:3]  # List slicing
Out[6]: ['Hufflepuff', 'Ravenclaw']
```

```
In[7]: houses[2:]  # List slicing
Out[7]: ['Ravenclaw', 'Slytherin']
```

```
In[8]: houses[:2]  # List slicing
Out[8]: ['Gryffindor', 'Hufflepuff']
```

How to represent list values: Memory Diagrams [1]

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

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

```
animalLists[0][1]  # Nested list indexing
```

```
animalLists[0][1]  # Nested list indexing
```

- **Slicing:** get a new list of all list elements at indices in the given range.

```
In[2]: animalLists[0][1]  # Nested list indexing
Out[2]: 'raccoon'
```

```
In[3]: mammals = animalLists[0]
```

```
In[4]: mammals
Out[4]: ['fox', 'raccoon']
```

```
In[5]: mammals[1]  # Nested list indexing
Out[5]: 'raccoon'
```

```
In[6]: animalLists[1][2]  # Nested list indexing
Out[6]: 'gosling'
```

```
In[7]: animalLists[2][1]  # Nested list indexing
Out[7]: 'raccoon'
```

```
In[8]: animalLists[3][0]  # Nested list indexing
Out[8]: 'turkey'
```

- Nested list indexing is not special! It is just repeated list indexing.

```
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>x in seq</td>
<td>True if an item of seq is equal to x</td>
</tr>
<tr>
<td>x not in seq</td>
<td>False if an item of seq is equal to x</td>
</tr>
<tr>
<td>seq1 + seq2</td>
<td>The concatenation of seq1 and seq2</td>
</tr>
<tr>
<td>seq<em>n, n</em>seq</td>
<td>n copies of seq concatenated</td>
</tr>
<tr>
<td>seq[i]</td>
<td>i\text{th} item of seq, where origin is 0</td>
</tr>
<tr>
<td>seq[i:j]</td>
<td>slice of seq from i to j</td>
</tr>
<tr>
<td>seq[i:j:k]</td>
<td>slice of seq from i to j with step k</td>
</tr>
<tr>
<td>len(seq)</td>
<td>length of seq</td>
</tr>
<tr>
<td>min(seq)</td>
<td>smallest item of seq</td>
</tr>
<tr>
<td>max(seq)</td>
<td>largest item of seq</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** 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.

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

```
myList: [17, True, False, 'I', 'am', 'Sam', 'was']
```

---

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

```python
myList.append(42)
```

```python
myList[4].append('Adams')
```

```
myList: [17, 23, True, False, 42, 'Adams']
```

---

**List Mutability**

Assigning to a list index:

```python
In []: numStrings = ['zero', 'one', 'two', 'three', 'four']
In []: numStrings[3] = 'THREE'
In []: numStrings
Out[ ]: ['zero', 'one', 'two', 'THREE', 'four']
```

Adding an element to the end of a list with **append**:

```python
In []: numStrings.append('five')
In []: numStrings
Out[ ]: ['zero', 'one', 'two', 'THREE', 'four', 'five']
```
**pop**: remove slot at an index and return its value

```python
myList.pop(3)
```

**List Diagrams/Mutability 10-17**

```python
myList.pop(3)  # Indices of slots after 3 are decremented
```

**List Diagrams/Mutability 10-18**

```python
myList[3].pop(2)  # Index of previous slot 3 is decremented
```

**List Diagrams/Mutability 10-19**

**List Diagrams/Mutability 10-20**
**pop**: remove slot at an index and return its value

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

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

```python
myList.insert(0, 98.6)
myList.insert(0, 98.6)  # Indices of previous slots 0 and above are incremented
```
**insert**: add a slot, add an index

```python
myList.insert(0, 98.6)  # Indices of previous slots 0 and above
# are incremented

myList[4].insert(2, 'not')
```

**Aliasing**: the very same object can be stored in different variables & slots

```python
list2 = myList
```

```python
myList = [98.6, 17, 23, True]  # Indices of previous slots 0 and above
# are incremented

myList[4].insert(2, 'not')  # Index of previous slot 2 is incremented
```
**Aliasing: the very same object can be stored in different variables & slots**

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

```
myList
list2
0 1 2 3 4 5
98.6 17 23 True False

Circle
radius 50
direction

Point
x 200
y 100

'I' 'was' 'not' 'Adams'
```

List Diagrams/Mutability 10-29

**Aliasing: the very same object can be stored in different variables & slots**

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

```
myList
list2
0 1 2 3 4 5
circ
98.6 17 23 True False

Circle
radius 50
direction

Point
x 200
y 100

'I' 'was' 'not' 'Adams'
```

List Diagrams/Mutability 10-30

**Aliasing: the very same object can be stored in different variables & slots**

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

```
myList
list2
0 1 2 3 4 5
98.6 17 23 True False

Circle
radius 50
direction

Point
x 200
y 100

'I' 'was' 'not' 'Adams'
```

List Diagrams/Mutability 10-31

**Aliasing: the very same object can be stored in different variables & slots**

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

```
myList
list2
0 1 2 3 4 5
circ
98.6 17 23 True False

Circle
radius 50
direction

Point
x 200
y 100

'I' 'was' 'not' 'Adams'
```

List Diagrams/Mutability 10-32
Aliasing: the very same object can be stored in different variables & slots

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

```
myList

<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>23</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

List Diagrams/Mutability 10-33
```

Aliasing: the very same object can be stored in different variables & slots

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

```
myList

<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>23</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

List Diagrams/Mutability 10-34
```

Aliasing: the very same object can be stored in different variables & slots

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

```
myList

<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>23</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

List Diagrams/Mutability 10-35
```

Aliasing: the very same object can be stored in different variables & slots

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

```
myList

<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>23</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

List Diagrams/Mutability 10-36
```
Aliasing: the very same object can be stored in different variables & slots

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

List Diagrams/Mutability 10-37

```
myList
<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>'I'</td>
<td>'was'</td>
<td>'a'</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
list2
<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>23</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
circ
<table>
<thead>
<tr>
<th>radius</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>
```

List Diagrams/Mutability 10-39

## Aliasing: the very same object can be stored in different variables & slots

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

List Diagrams/Mutability 10-38

```
myList
<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>23</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
circ
<table>
<thead>
<tr>
<th>radius</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>
```

List Diagrams/Mutability 10-40
Aliasing: the very same object can be stored in different variables & slots

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

```
myList[1][2] = 'a'
list2[4][2] = 'a'
myList[5].setRadius(75)
circ.getRadius()
```

List Diagrams/Mutability 10-41

```
list2 = myList
circ = list2[5]
myList[1][3] = circ
circ.getRadius() = 75
```

List Diagrams/Mutability 10-42

```
list2 = myList
circ = list2[5]
myList[1][3] = circ
myList[5].setRadius(75)
circ.getRadius() = 75
```

List Diagrams/Mutability 10-43

```
list2 = myList
circ = list2[5]
myList[1][3] = circ
circ.getRadius() = 75
list2[1][3].getRadius() = 75
```

List Diagrams/Mutability 10-44
What is the final value of \( c[0] \)?

\[
\begin{align*}
  a &= [15, 20] \\
  b &= [15, 20] \\
  c &= [10, a, b] \\
  b[1] &= 2*a[0] \\
  c[1][0] &= c[0] \\
  c[0] &= a[0] + c[1][1] + b[0] + c[2][1]
\end{align*}
\]

Draw a memory diagram!

Does the answer change if we change the 2nd line from
\[
b = [15, 20] \quad \text{to} \quad b = a[:]
\]

Does the answer change if we change the 2nd line from
\[
b = [15, 20] \quad \text{to} \quad b = a
\]

Strings are immutable. What about strings?

Strings are sequences:

\[
\begin{align*}
  \text{In [6]: } & \quad \text{name} = 'Gryffindor' \\
  \text{In [7]: } & \quad \text{name[2]} \quad \# \quad \text{y} \\
  \text{In [8]: } & \quad \text{name[4:8]} \quad \# \quad \text{find} \\
  \text{In [9]: } & \quad \text{'do' in name} \quad \# \quad \text{True}
\end{align*}
\]

Mutation operations do not work on strings:

\[
\begin{align*}
  \text{In [10]: } & \quad \text{name[4]} = 't' \quad \# \quad \text{what happens?} \\
  \text{In [11]: } & \quad \text{name.append('s')} \quad \# \quad \text{what happens?}
\end{align*}
\]

Tuples

Lists are mutable sequences of values.
Tuples are immutable sequences of values.

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

\[
\begin{align*}
  \# \quad \text{A homogeneous tuple of five integers} \quad \text{(a 4-tuple)} \\
  & \quad (5, 8, 7, 1, 3) \\
  \# \quad \text{A homogeneous tuple of four strings} \quad \text{(Gryffindor, 'Hufflepuff', 'Ravenclaw', 'Slytherin')} \\
  \# \quad \text{A heterogeneous tuple of three elements} \quad \text{(a 3-tuple)} \\
  & \quad (42, 'Hello', False) \\
  \# \quad \text{A pair is a tuple with two elements} \quad \text{(a 2-tuple)} \\
  & \quad (7, 3) \\
  \# \quad \text{A tuple with one element must use a comma to avoid} \\
  \quad \text{being confused with a parenthesized expression} \\
  & \quad () \quad \# \quad \text{A tuple with 0 values}
\end{align*}
\]
**Tuples are immutable sequences.**

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

```python
In[32]: houseTuple = ('Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin')
Out[32]: 'Gryffindor'
In[33]: houseTuple[2]
Out[33]: 'Ravenclaw'
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')
```

**Mutation operations do not work on tuples.**

```python
In [38]: houseTuple[0] = '4 Privet Drive'
----------------------------------------------------------
TypeError
In [39]: houseTuple.append('The Shrieking Shack')
-----------------------------------------------------------
AttributeError
In [40]: houseTuple.pop(1)
-----------------------------------------------------------
AttributeError
```

**Conversion between sequence types**

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

```python
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]'
```

**Enumerations**

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

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

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 n
3 e
4 s
5 t