Lists, Memory Diagrams & Mutable vs. Immutable Sequences

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']]

Why Lists (and other sequences)?

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

10-2

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.

10-3

Lists: glue for many values

# Lists returned from builtin functions and methods
ododies = 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 = []
How to represent list values: Memory Diagrams [0]

- **Primes**
  - list slot indices
  - list slots
  - primes
    - 2
    - 3
    - 5
    - 7
    - 11
    - 13
    - 17
    - 19

- **Bools**
  - list slot indices
  - list slots
  - booleans
    - True
    - False

- **Houses**
  - list slot indices
  - list slots
  - houses
    - index 0: 'Gryffindor'
    - index 1: 'Hufflepuff'
    - index 2: 'Ravenclaw'
    - index 3: 'Slytherin'

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**
  - list slot indices
  - list slots
  - animalLists
    - index 0: ['fox', 'raccoon']
    - index 1: ['duck', 'raven', 'gosling']
    - index 2: []
    - index 3: ['turkey']

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.

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]`
- **Out[5]**: 'Gryffindor'

Negative indexing: negative indices index from the end of the list.

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

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

- **In[7]**: `houses[2:]`
- **Out[7]**: ['Ravenclaw', 'Slytherin']

- **In[8]**: `houses[:2]`
- **Out[8]**: ['Gryffindor', 'Hufflepuff']

List Diagrams/Mutability

Nested list indexing (is not special!)

- **In[1]**: `animalLists = [['fox', 'raccoon'],
   ['duck', 'raven', 'gosling'],
   [],
   ['turkey']]`
- **In[2]**: `animalLists[0][1]`
- **Out[2]**: 'raccoon'

List of lists.

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

- **In[3]**: `mammals = animalLists[0]`
- **In[4]**: `mammals`
- **Out[4]**: ['fox', 'raccoon']

- **In[5]**: `mammals[1]`
- **Out[5]**: 'raccoon'

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><em>x in seq</em></td>
<td>True if an item of seq is equal to x</td>
</tr>
<tr>
<td><em>x not in seq</em></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><em>n</em> <em>seq</em></td>
<td><em>n</em> copies of seq concatenated</td>
</tr>
<tr>
<td>seq[i]</td>
<td><em>i</em>th item of seq, where origin is 0</td>
</tr>
<tr>
<td>seq[i:j]</td>
<td>slice of seq from <em>i</em> to <em>j</em></td>
</tr>
<tr>
<td>seq[i:j:k]</td>
<td>slice of seq from <em>i</em> to <em>j</em> with step <em>k</em></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 example**

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

```
shoesizes[3] = 11.5
```

**List slot mutability larger example [0]**

```
myList = [17, 3.141, True, None, ['I', 'am', 'Sam'], Circle(50, Point(200, 100))]
```

```
myList[4] = ['I', 'am', 'Sam']
```

```
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.

\[
\text{myList}[1] = \text{myList}[0] + 6 \\
\text{myList}[3] = \text{myList}[0] > \text{myList}[1] \\
\text{myList}[4][1] = 'was'
\]

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

\[
\text{myList}.append(42) \\
\text{myList}[4].append('Adams')
\]

More list mutability

pop

(removing 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)`

```
0 1 2 3 4 5 6
17 23 True False False False
```

List Diagrams/Mutability 10-17

```
myList[3].pop(2)
```

List Diagrams/Mutability 10-19

```python
myList = [17, 23, True, False, False, False]
myList.pop(3)
```

List Diagrams/Mutability 10-17

```
myList = [17, 23, True, False, False, False]
myList[3].pop(2)
```

List Diagrams/Mutability 10-19
**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) → 'Sam'  # Index of previous slot 3 is decremented
myList.pop()

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

myList.insert(0, 98.6)

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() → 42  # When no index, last one is assumed
**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
# Indices of previous slots 0 and above
# are incremented

myList[4].insert(2, 'not')  # Index of previous slot 2 is incremented

list2 = myList
```
Aliasing: the very same object can be stored in different variables & slots

\[
\text{list2} = \text{myList}
\]
\[
\text{circ} = \text{list2}[5]
\]

\[
\text{myList} = [98.6, 17, 23, \text{True}, \text{None}, \text{"I" was "not" "Adams"}]
\]
\[
\text{list2} = \text{myList}
\]
\[
\text{circ} = \text{list2}[5]
\]

\[
\text{myList}[1] = \text{myList}[4]
\]
\[
\text{list2} = \text{myList}
\]
\[
\text{circ} = \text{list2}[5]
\]
Aliasing: the very same object can be stored in different variables & slots

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

List Diagrams/Mutability 10-33

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

List Diagrams/Mutability 10-34

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

List Diagrams/Mutability 10-35

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

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
```

```
list2 = myList
myList[1][2] = 'a'
circ = list2[5]
list2[4][2] = 'a'
myList[1][3] = circ
```

List Diagrams/Mutability 10-37

List Diagrams/Mutability 10-38

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

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

```
list2 = myList
myList[1][2] = 'a'
circ = list2[5]
list2[4][2] = 'a'
myList[1][3] = circ
```

List Diagrams/Mutability 10-39

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
What is the final value of \( c[0] \)?

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

Draw a memory diagram!

Does the answer change if we change the 2nd line from 
\( b = [15, 20] \) to \( b = a[:] \)?

Does the answer change if we change the 2nd line from 
\( b = [15, 20] \) to \( b = a \)?

---

**Lists are mutable. What about strings?**

Strings are sequences:

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

Mutation operations **do not work** on strings:

\[
\begin{align*}
\text{In}[10]: \quad & \text{name}[4] = 't' \quad \# what happens? \\
\text{TypeError} \quad & \text{name[0]} = 't' \\
\text{TypeError} \quad & \text{'str' object does not support item assignment}
\end{align*}
\]

\[
\begin{align*}
\text{In}[11]: \quad & \text{name.append('s')} \quad \# what happens? \\
\text{AttributeError} \quad & \text{name.append('s')} \\
\text{AttributeError} \quad & \text{'str' object has no attribute 'append'}
\end{align*}
\]

---

**Strings are immutable sequences.**

Once you create a string, it cannot be changed.

\[
\begin{align*}
\text{In}[13]: \quad & \text{college} = 'WELLESLEY' \\
\text{college} & \quad \rightarrow 'WELLESLEY' \\
\text{Immutable, not changed.}
\end{align*}
\]

\[
\begin{align*}
\text{In}[14]: \quad & \text{college.lower( )} \\
\text{Out}[14]: \quad & 'wellesley' \quad \# Returns a new string 'wellesley'; \\
& \quad \# old one is unchanged!
\end{align*}
\]

\[
\begin{align*}
\text{In}[15]: \quad & \text{myCollege} = \text{college.lower( )} \\
\text{myCollege} & \quad \rightarrow 'wellesley'
\end{align*}
\]

---

**Tuples**

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

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

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

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

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

Mutation operations do not work on tuples.

```python
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.

```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]'
```

Tuple Assignment

Suppose `harryInfo` is a tuple of three values:

```python
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:

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

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

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

We can now use these names like any other variables:

```python
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:

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

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

Note that `for (index, char) in enumerate('boston'):` is a use of tuple assignment notation in a `for` loop.