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.

Lists: glue for many values
# Lists returned from built-in 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', '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]
animals = [['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, 'Wandy', None, [42, False, 'computer']]
How to represent list values: Memory Diagrams [0]

- a variable starts at 0, not 1
- list indices
- list slot indices
- list slots
- negative indices work from end
- 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 is a list of lists.
- nested list indexing is not special.
- it is just repeated list indexing.

List indexing and slicing (review)

- indexing: get one element from the given position (index) in the list.
- negative indexing: negative indices index from the end of the list.
- slicing: get a new list of all list elements at indices in the given range.

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>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[i]</td>
<td>i_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**, 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]

```
  slots
shoesizes 0 1 2 3
  8   8.5 12.5 10

shoesizes[3] = 11.5
```

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

```python
circles = Circle(50, Point(200, 100))

myList = [17, 3.141, True, None, ['I', 'am', 'Sam'], circles]
```

```
  slots
myList 0 1 2 3 4 5
  17 3.141 True None 1
```

**instance variables**

- `radius`
- `location`

- `x` 200
- `y` 100

- `an object`
**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'
```

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

```python
myList.append(42)
myList[4].append('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']
```

**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:**

```
myList
0 1 2 3 4 5
| 17 23 True False 42 |
```

```
Circle
x 200
y 100
'1' 'was' 'Sam' 'Adams'
```

List Diagrams/Mutability 8-17

**Final memory diagram:**

```
myList
0 1 2 3 4
| 17 23 True |
```

```
Circle
x 200
y 100
'I' 'was' 'Adams'
```

List Diagrams/Mutability 8-18

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

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

**Draw updates:**

```
myList
0 1 2 3 4
| 17 23 True |
```

```
Circle
x 200
y 100
'I' 'was' 'Adams'
```

List Diagrams/Mutability 8-19

**Final memory diagram:**

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

```
Circle
x 200
y 100
'I' 'was' 'not' 'Adams'
```

List Diagrams/Mutability 8-20
**Aliasing:** the very same object can be stored in different variables & slots

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

---

**Draw updates:**

---

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

```python
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]
```

**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:

```python
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:**

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

```
TypeError ... name[0]= 't'
TypeError: 'str' object does not support item assignment
```

```python
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'

```
college ➡ 'WELLESLEY'
```

Immutable, not changed.

In[14]: college.lower( )

Out[14]: 'wellesley'  # Returns a new string 'wellesley';  # old one is unchanged!

In[15]: myCollege = college.lower( )

```
myCollege ➡ 'wellesley'
```

Tuples

Lists are mutable sequences of values.

Tuples are immutable sequences of values.

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)

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

# A tuple with 0 values

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')

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.

```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[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
   ...:     print index, char
0 b
1 o
2 s
3 t
4 o
5 n
```