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 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']]
```

Why Lists (and other sequences)?

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

```
# 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.
```

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

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

Lists: glue for many values

```
# Lists returned from builtin functions and methods
oddsies = 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]

- a variable
  - list indices start at 0, not 1
  - 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
  - list of lists
  - nested list indexing is not special.
  - it is just repeated list indexing.

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]  # List indexing
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 all list elements at indices in the given range.
Out[6]: ['fox', 'raccoon']
In[7]: mammals[1]  # Slicing: get a new list of all list elements at indices in the given range.
Out[7]: 'raccoon'
In[8]: mammals[:2]  # List slicing
Out[8]: ['fox', 'raccoon']

List Diagrams/Mutability

10-5
10-6
10-7
10-8
**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 seq is equal to x</td>
</tr>
<tr>
<td><code>x not in seq</code></td>
<td>False if an item of seq is equal to x</td>
</tr>
<tr>
<td><code>seq1 + seq2</code></td>
<td>The concatenation of seq1 and seq2</td>
</tr>
<tr>
<td><code>seq*n, n*seq</code></td>
<td>n copies of seq concatenated</td>
</tr>
<tr>
<td><code>seq[i]</code></td>
<td>i'th item of seq, where origin is 0</td>
</tr>
<tr>
<td><code>seq[i:j]</code></td>
<td>slice of seq from i to j</td>
</tr>
<tr>
<td><code>seq[i:j:k]</code></td>
<td>slice of seq from i to j with step k</td>
</tr>
<tr>
<td><code>len(seq)</code></td>
<td>length of seq</td>
</tr>
<tr>
<td><code>min(seq)</code></td>
<td>smallest item of seq</td>
</tr>
<tr>
<td><code>max(seq)</code></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))]
```

---

**List Diagrams/Mutability**

Page 9

---

**List Diagrams/Mutability**

Page 10

---

**List Diagrams/Mutability**

Page 11

---

**List Diagrams/Mutability**

Page 12
**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:**

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

**Final memory diagram:**

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>42</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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:**

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Final memory diagram:**

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>42</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Aliasing: the very same object can be stored in different variables & slots

```python
def draw_updates():
    list2 = myList
    circ = list2[5]
    myList[1][3] = circ
```

Draw updates:

Final memory diagram:

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

List Diagrams/Mutability 10-22

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]
```

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

Draw a memory diagram!

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: 'str' object does not support item assignment

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

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

Once you create a string, it cannot be changed.

In[13]: college = 'WELLESLEY'

Out[13]:

Immutability, not changed.

In[14]: college.lower()

Out[14]: 'wellesley'

# Returns a new string 'wellesley';
# old one is unchanged!

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

myCollege

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

Mutation operations do not work on tuples.

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

----------------------------------------------------------
TypeError
...

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

-----------------------------------------------------------
AttributeError
...

In [40]: houseTuple.pop(1)

-----------------------------------------------------------
AttributeError
...

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')
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]'

Enumerations

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

In [46]: list(enumerate('boston'))
Out[46]: [(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