Lists, Memory Diagrams & Mutable vs. Immutable Sequences

CS111 Computer Programming
Department of Computer Science
Wellesley College
Why Lists (and other sequences)?

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

- Lists of all Wellesley courses (714)
- List of all registered US voters
- List of all public tweets ever posted on Twitter, in time order
- 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 **homogeneous**. Most of the lists we’ll use will be homogeneous.

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

Lists can also contain other lists as elements!

```python
# 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.)

```python
[17, True, 'Wendy', None, [42, False, 'computer']]
```
Lists: glue for many values

# Lists returned from builtin functions and methods
oddlies  = 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

- **Primes**: Numbers, booleans, and `None` are “small enough” to fit directly in variables and list slots.
- **Bools**: All other values are drawn outside the variable/list slot, with an arrow pointing to them.
- **Houses**: 'Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin'

List Diagrams/Mutability
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: list index out of range

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

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

In[7]: houses[2:]     # Slicing
Out[7]: ['Ravenclaw', 'Slytherin']

In[8]: houses[:2]     # Slicing
Out[8]: ['Gryffindor', 'Hufflepuff']

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.
How to represent list values: Memory Diagrams [1]

animalLists = [['fox', 'raccoon'],

['duck', 'raven', 'gosling'],

[],

['turkey']]
Nested list indexing (is not special!)

In[1]: animalLists = [['fox', 'raccoon'], ['duck', 'raven', 'gosling'], [], ['turkey']]

In[2]: animalLists[0][1]
Out[2]: 'raccoon'

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>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’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 example

\[ \text{shoesizes} = [8, 8.5, 12.5, 10] \]

\[ \text{shoesizes}[3] = 11.5 \]
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.

\[
\begin{align*}
\text{myList}[1] &= \text{myList}[0] + 6 \\
\text{myList}[3] &= \text{myList}[0] > \text{myList}[1] \\
\text{myList}[4][1] &= 'was'
\end{align*}
\]
**append**: add a new slot to the end of a list

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

List Diagrams/Mutability 10-14
List Mutability

Assigning to a list index:

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:

In [ ]: numStrings.append('five')
In [ ]: numStrings
Out[ ]: ['zero', 'one', 'two', 'THREE', 'four', 'five']
More list mutability

**pop**
(removes 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

```python
myList.pop(3)
```
**pop**: remove slot at an index and return its value

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

List Diagrams/Mutability 10-18
**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)
```

List Diagrams/Mutability 10-19
**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
```
**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()
```
**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() ➞ 42  # When no index, last one is assumed
```

List Diagrams/Mutability 10-22
**insert**: add a slot, add an index

```python
myList.insert(0, 98.6)
```

List Diagrams/Mutability 10-23
**insert**: add a slot, add an index

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

List Diagrams/Mutability
**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')
```

List Diagrams/Mutability 10-25
**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')  # Index of previous slot 2 is incremented
```

List Diagrams/Mutability 10-26
Aliasing: the very same object can be stored in different variables & slots

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

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

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

![List Diagrams/Mutability](List Diagrams/Mutability 10-29)
Aliasing: the very same object can be stored in different variables & slots

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

```python
list2 = myList

circ = list2[5]

```

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

```
list2 = myList

circ = list2[5]

```

List Diagrams/Mutability

List Diagrams/Mutability
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 10-33
Aliasing: the very same object can be stored in different variables & slots

\[ \text{list2} = \text{myList} \]
\[ \text{circ} = \text{list2[5]} \]
\[ \text{myList[1]} = \text{myList[4]} \]
\[ \text{myList[1][3]} = \text{circ} \]
Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList

circ = list2[5]


myList[1][3] = circ
```
Aliasing: the very same object can be stored in different variables & slots

\[
\text{list2} = \text{myList} \\
\text{circ} = \text{list2}[5] \\
\text{myList}[1] = \text{myList}[4] \\
\text{myList}[1][3] = \text{circ}
\]
Aliasing: the very same object can be stored in different variables & slots

```
list2 = myList

circ = list2[5]


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

List Diagrams/Mutability 10-37
Aliasing: the very same object can be stored in different variables & slots

\[
\begin{align*}
\text{list2} & = \text{myList} \\
\text{circ} & = \text{list2}[5] \\
\text{myList}[1] & = \text{myList}[4] \\
\text{myList}[1][3] & = \text{circ}
\end{align*}
\]

\[
\begin{align*}
\text{myList}[1][2] & = \text{'}a\text{'} \\
\text{list2}[4][2] & \rightarrow \text{'}a\text{'}
\end{align*}
\]
Aliasing: the very same object can be stored in different variables & slots

\[
\begin{align*}
\text{list2} &= \text{myList} \\
\text{circ} &= \text{list2}[5] \\
\text{myList}[1] &= \text{myList}[4] \\
\text{myList}[1][3] &= \text{circ}
\end{align*}
\]

\[
\begin{align*}
\text{myList}[1][2] &= \text{'}a\text{'} \\
\text{list2}[4][2] &= \text{'}a\text{'} \\
\text{myList}[5].\text{setRadius}(75)
\end{align*}
\]
List Diagrams/Mutability

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

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

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

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

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

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

List Diagrams/Mutability 10-41
Aliasing: the very same object can be stored in different variables & slots

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

```
myList[1][2] = 'a'
list2[4][2] = 'a'
myList[5].setRadius(75)
circ.getRadius() = 75
```
Aliasing: the very same object can be stored in different variables & slots

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

```
myList[1][2] = 'a'
list2[4][2] = 'a'
myList[5].setRadius(75)
circ.getRadius() = 75
list2[1][3].getRadius()
```

List Diagrams/Mutability 10-43
Aliasing: the very same object can be stored in different variables & slots

```python
list2 = myList
circ = list2[5]
myList[1][3] = circ
myList[1][2] = 'a'
list2[4][2] = 'a'
myList[5].setRadius(75)
circ.getRadius() = 75
list2[1][3].getRadius() = 75
```

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

\[
a = [15, 20] \\
b = [15, 20] \\
c = [10, a, b]
\]

\[
b[1] = 2*a[0] \\
c[1][0] = c[0] \\
c[0] = a[0] + c[1][1] + b[0] + c[2][1]
\]

Draw a memory diagram!

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

Does the answer change if we change the 2\textsuperscript{nd} line from \(b = [15, 20]\) to \(b = a\)?
Lists are mutable. What about strings?

Strings are sequences:

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

Mutation operations do not work on strings:

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

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

```
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'

\[
\begin{array}{c}
\text{college} \rightarrow 'WELLESLEY' \\
\end{array}
\]

Immutble, not changed.

In[14]: college.lower()

Out[14]: 'wellesley'

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

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

\[
\begin{array}{c}
\text{myCollege} \rightarrow 'wellesley' \\
\end{array}
\]
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.

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

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