Announcements

- PS 4 is due on Monday, 10/26
- Try to get help
- Reading assignment for this slide set:
  - Chapter 10 of Downey
- Break around 10:15am
Lists

- A list is a sequence of values

- String vs. list
  - String is a sequence of characters
  - List is a sequence of values of any type

  - String is immutable
  - List is mutable
Creating a list

• Simply enclose the elements of a list in [ and ]

```python
>>> []   # empty list
>>> [1, 2, 4, 23, 47]
>>> ['apple', 'pear', 'kiwi']
>>> ['John Doe', 23, 'History', 3.56, '909-345-1232']
```

• Call `range(n)` like this:

```python
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(2, 10)
[2, 3, 4, 5, 6, 7, 8, 9]
>>> range(2, 10, 3)
[2, 5, 8]
```
Lists are **mutable**

- The syntax of accessing elements of a list is the same as for accessing the characters in a string
  ```python
  >>> fruits = ['apple', 'pear', 'kiwi']
  >>> print fruits[0]
  'apple'
  >>> print fruits[2]
  'kiwi'
  ```

- Can **update** elements in a list in place, thus **mutable**!
  ```python
  >>> fruits[2] = 'orange'
  >>> fruits
  ['apple', 'pear', 'orange']
  ```
Lists are indexed (0-based)

- A list is a relationship between indices and elements
- This relationship is a *mapping* from an index to an element, e.g., 1 is mapped to ‘pear’

<table>
<thead>
<tr>
<th>'apple'</th>
<th>'pear'</th>
<th>'orange'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- List indices work the same way as string indices
  - Any *integer* expression can be used as an index
  - Trying to read or write an element with a non-existing index gives an error
  - If an index has a negative value, it counts backward from the end of the list (−1 is the index of the *last* element)
in with a list

• Works as expected
  >>> fruits = ['apple', 'pear', 'orange']
  >>> 'apple' in fruits
  True

• Traversing a list
  >>> for f in fruits:
      print f

  >>> for n in []
      print 'this will never be printed'
List operations

- The + operator concatenates lists
  
  ```python
  >>> a = [1, 2, 3]
  >>> b = [3, 4, 5]
  >>> c = a + b
  >>> c
  [1, 2, 3, 3, 4, 5]
  ```

- The * operator repeats a list a given number of times
  
  ```python
  >>> ['a'] * 5
  ['a', 'a', 'a', 'a', 'a']
  >>> [1, 2, 3] * 3
  [1, 2, 3, 1, 2, 3, 1, 2, 3]
  ```
List slices with ‘::’

- Slicing works as expected on lists as well
  ```python
  >>> u = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
  >>> u[1:3]
  ['b', 'c']
  >>> u[:3]
  ['a', 'b', 'c']
  >>> u[5:]
  ['f', 'g']
  >>> u[:]
  ['a', 'b', 'c', 'd', 'e', 'f', 'g']
  ```

- Slice operator is mutable
  ```python
  >>> u[1:4] = ['B', 'C', 'D']
  >>> u
  ['a', 'B', 'C', 'D', 'e', 'f', 'g']
  ```
List methods

• The **append** method:

```python
>>> u = ['a', 'b', 'c']
>>> u.append('d')
>>> print u
['a', 'b', 'c', 'd']
>>> u.append(['x', 'y'])
>>> print u
['a', 'b', 'c', 'd', ['x', 'y']]
```

• The **extend** method:

```python
>>> u1 = ['x', 'y', 'z']
>>> u2 = ['a', 'b']
>>> u1.extend(u2)
>>> print u1
['x', 'y', 'z', 'a', 'b']
```

• The **sort** method:

```python
>>> u1.sort()
>>> print u1
['a', 'b', 'x', 'y', 'z']
```
Deleting elements

- Use `pop` if you know the **index** of the element to be deleted
- `pop` modifies the list and returns the element that was removed

```python
>>> u = ['a', 'b', 'c']
>>> x = u.pop(1)
>>> print u
['a', 'c']
>>> print x
b
```

- `del` is like `pop` again using index except it does not return the element deleted

```python
>>> u = ['a', 'b', 'c']
>>> del u[1]
>>> print u
['a', 'c']
```
Deleting elements (cont.)

- Use `remove` if you know the element to be deleted, but not the index
  ```python
  >>> u = ['a', 'b', 'c']
  >>> u.remove('b')
  >>> print u
  ['a', 'c']
  ```
- The return value from `remove` is `None`
- Use `del` with a slice index if you want to remove *multiple* elements
  ```python
  >>> u = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
  >>> del u[2:4]
  >>> print u
  ['a', 'b', 'e', 'f', 'g']
  ```
- Note that `pop`, `del` and `remove` use different syntax
Lists and strings

- A string is a sequence of characters
- A list is a sequence of values of any type
- But a list of characters is not the same as a string, e.g.,
  - A list of characters: ['r', 'a', 'y']
  - A string: 'ray'

To convert a string into a list of characters, use `list`:

```python
>>> s = 'apple'
>>> u = list(s)
>>> print(u)
['a', 'p', 'p', 'l', 'e']
```
list and split

• The **list** function breaks a string into individual characters and put them in a list (previous slide)

• The **split** function breaks a multi-word string into a list of words

  ```python
  >>> s = 'this is a multi-word string.'
  >>> u = s.split()
  >>> print u
  ['this', 'is', 'a', 'multi-word', 'string']
  ```

• An optional argument called a **delimiter** specifies which characters to use as word boundaries. Example: using `'-'` as a delimiter:

  ```python
  >>> s = 'multi-word string like-this'
  >>> s.split('-')
  ['multi', 'word string like', 'this']
  ```
split and join

- **join** is the inverse of **split**

- It takes a list of strings and concatenates the elements

- **join** is a string method, so you have to invoke it on the delimiter and pass the list as an argument

```python
>>> u = ['multi', 'word string like', 'this']
>>> delimiter = '-
>>> delimiter.join(u)
'multi-word string like-this'
```
Sameness: identical vs. equivalent

- The two assignment statements below creates two names with one copy of ‘apple’
  \[
  a = \text{‘apple’} \\
  b = \text{‘apple’} \\
  a == b \text{ is True if equivalent, i.e., same value although stored in diff locations} \\
  a \text{ is } b \text{ is True if same value in the same memory location}
  \]

- The two assignment statements below create two names with two different copies of \([1, 2, 3, 4]\)
  \[
  u1 = [1, 2, 3, 4] \\
  u2 = [1, 2, 3, 4] \\
  u1 == u2 \text{ is True (because equivalent, same value)} \\
  u1 \text{ is } u2 \text{ is False (because not identical, not in the same memory location)}
  \]

- See the figures on the next slide
Identical vs. equivalent (cont.)

- **a** and **b** below are not only equivalent but also identical
  - **a** and **b** are string literals
- **u1** and **u2** below are equivalent but not identical
  - **u1** and **u2** are list objects

\[
\begin{align*}
\text{a} & \rightarrow \text{‘apple’} \\
\text{b} & \\
\text{u1} & \rightarrow [1, 2, 3, 4] \\
\text{u2} & \rightarrow [1, 2, 3, 4]
\end{align*}
\]
Aliasing (next)

- If \( a \) refers to an object (like a list) and you assign \( b = a \), then both variables refer to the same object, i.e., they are identical.
  
  \[
  \begin{align*}
  a &= [1, 2, 3, 4] \\
  b &= a \\
  \text{b is a is True}
  \end{align*}
  \]

- The association of a variable with an object is called a *reference*.
  - Both \( a \) and \( b \) are two references to the same object \([1, 2, 3, 4]\).

- An object with more than one reference has more than one name, and we say that the object is *aliased*.

- If the aliased object is mutable, changes made with one alias affect the other.
  
  >>> b[1] = 22
  >>> print a
  [1, 22, 3, 4]
Passing a list as an argument

- When a list is passed as an argument to a function, the function gets a reference to the list.
- If the function modifies the list parameter, the caller sees the change.

```python
def delete_head(u):
    del u[0]

>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

- A stack diagram:

```
delete_head: u           ['a', 'b', 'c']
    __main__: letters
```
Mutation vs. new creation

- The `+` operator creates a new list
- The `append` method modifies an existing list

```python
>>> u1 = [1, 2, 3]
>>> u2 = u1.append(4)
>>> print u1
[1, 2, 3, 4]
>>> print u2
None
>>> u3 = u1 + [4, 5]
>>> print u3
[1, 2, 3, 4, 4, 5]
```

- The distinction is important when you . . . (next slide)
Writing functions that modify lists

- This function does not delete the head of a list:
  ```python
def bad_delete_head(u):
    u = u[1:]  # Wrong! (try stack diagram to see why!)
  ```
- The slice operator creates a new list and the assignment makes `u` to refer to it, but none of that has any effect on the list that was passed as an argument. It only affects the local copy in the function.
- Alternative? Write a function that creates and returns a new list and have the caller use the returned value. Here, `tail` returns all but the first of a list:
  ```python
def tail(u):
    return u[1:]

>>> letters = ['a', 'b', 'c']
>>> letters = tail(letters)
>>> print letters
['b', 'c']
```
Exercise 1

- Write a function called `length` that takes a list as a parameter and returns the length of the list. Python already gives us a function named `len` that does this, but I want you to write one without using `len`. You may use `for ... in ...` though.

- For example,

  ```python
  length([]) = 0
  length(['a']) = 1
  length(['a', [1, 2, 3], 'c']) = 3
  ```
Exercise 2

• Write a function called `is_member` that takes an element and a list as two parameters and returns `True` if the element is contained in the list; and returns `False` otherwise. Python already gives us an operator called `in` that does this, but I want you to write one without using `in`. You may use `for ... in ...` though. Note that `in` has two different meanings depending on the context, e.g.,
  • `for x in u` : to iterate elements in `u`
  • `x in u` : to see if `x` is contained in `u`

• For example,
  ```python
  is_member(1, []) = False
  is_member('c', ['a', 'b', 'c']) = True
  is_member([1, 2, 3], ['a', [1, 2, 3], 'c']) = True
  ```
Exercise 3

• Write a function called `is_sorted` that takes a list as a parameter and returns `True` if the list is sorted in ascending order and `False` otherwise. You can assume (as a precondition) that the elements of the list can be compared with the relational operators: `<`, `>`, etc. You may use `length` that you defined earlier or the `len` function from Python.

• For example, `is_sorted(['a', 'c', 'd', 'd', 'e'])` should return `True` and `is_sorted([1, 2, 4, 3])` should return `False`. 
Exercise 4

- Write a function called `has_duplicates` that takes a list and returns `True` if there is any element that appears more than once. It should not modify the original list. For example,
  
  ```python
  has_duplicates([1]) => False
  has_duplicates([1, 2, 3, 2, 4, 5]) => True
  has_duplicates([1, 3, 2, 4, 5]) => False
  ```

- This time write a function called `remove_duplicates` that takes a list and returns a new list with only the unique elements from the original. Hint: they don’t have to be in the same order. For example,
  
  ```python
  remove_duplicates([1, 2, 3, 2, 4, 5]) => [1, 3, 2, 4, 5]
  remove_duplicates([1, 3, 2, 4, 5]) => [1, 3, 2, 4, 5]
  ```

- There are more interesting exercises in the text
A map function

- An operation like `square_all` below is called a map because it "maps" a function (`square` in this case) onto each element of a sequence.

```python
def square(n):
    return n*n

def square_all(u):
    res = []
    for n in u:
        res.append(square(n))
    return res

>>> print square_all([1, 2, 3, 4])
[1, 4, 9, 16]
```
The **map** function

- `map(func, seq)` transforms a list by applying the function `func` to each element of `seq`.
- Its return value is the transformed list.

```python
def square(n):
    return n*n

# square_all call can be expressed like this using map:
>>> map(square, [1, 2, 3, 4])
[1, 4, 9, 16]

# Alternatively, using a nameless function for square:
>>> map(lambda n: n*n, [1, 2, 3, 4])
[1, 4, 9, 16]

lambda n: n*n is a way of creating the square function without giving an explicit name.
A filter function

An operation like `only_even` below is called a *filter* because it “selects” some of the elements and filters out the others.

```python
def is_even(n):
    return n % 2 == 0

def only_even(u):
    res = []
    for s in u:
        if is_even(s):
            res.append(s)
    return res

>>> print only_even([1, 2, 3, 4])
[2, 4]
```
The **filter** function

- **filter(func, list)** collects the elements from the list list which satisfy the function func.
- Its return value is the collected list.

```python
def is_even(n):
    return n%2 == 0

# only_even call can be expressed like this using filter:
>>> filter(is_even, [1, 2, 3, 4])
[2, 4]

# Alternatively using a nameless function for is_even
>>> filter(lambda n: n%2 == 0, [1, 2, 3, 4])
[2, 4]
```
**sum as a reduce function**

- **Example:**
  ```python
  >>> u = [1, 2, 3]
  >>> sum(u)
  6
  ```

- An operation like this that combines a sequence of elements into a single value is called *reduce*

- **sum** may be implemented as follows but is provided by Python:
  ```python
def mysum(u):
    total = 0
    for x in u:
        total = total + x
    return total
  ```
The **reduce** function

- `reduce(func, seq)` continually applies the function `func` to the sequence `seq`.
- It returns a single value.
  ```python
def plus(x, y):
    return x + y
  ```

  ```
>>> reduce(plus, [1, 2, 3, 4])
10
  ```
  
  # Alternatively, using a nameless function for `plus`:
  ```python
>>> reduce(lambda x,y: x+y, [1, 2, 3, 4])
10
  ```

- A function like `map`, `filter`, and `reduce` that takes a function as a parameter is called a **higher order function** and quite powerful.
Do these before next class

- Finish reading Chapter 10

- Next
  - Dictionaries (chapter 11)