As Apple works diligently on improving Siri before the launch of iOS 6, a new study points out how difficult it is to get Siri to provide accurate results to common questions.

http://www.digitaltrends.com/mobile/study-apples-siri-is-wrong-over-one-third-of-the-time/
Today

- Map, Filter, and Reduce
- Immutable Dictionaries
- Extras: Generator Expressions
Review: Immutable Recursive Lists

An *immutable recursive list* (or an *IRList*) is a *pair* such that:

- The first element of the pair is the *first* element of the list.
- The second element of the pair is the *rest* of the list – another immutable recursive list. The rest of the list could be empty.
IMMUTABLE RECURSIVE LISTS

empty_irlist = ()

def make_irlist(first, rest=empty_irlist):
    return (first, rest)

def irlist_first(irlist):
    return irlist[0]

def irlist_rest(irlist):
    return irlist[1]
**Review: Immutable Recursive Lists**

An IRList is a pair.

The first element of the pair is the first element of the list.

The empty tuple represents the empty list, and the end of the list.

The second element of the pair is the rest of the list.

Also an IRList!
EXAMPLE: APPENDING IRLISTS

Let's write the function `irlist_append` which takes two IRLists and returns a new IRList with the elements of the first IRList followed by the elements of the second IRList.

```python
>>> x = irlist_populate(1, 2, 3)
>>> y = irlist_populate(4, 5, 6)
>>> irlist_str(irlist_append(x, y))
"<1, 2, 3, 4, 5, 6>"
```
Example: Appending IRLists

```python
>>> x = irlist_populate(1, 2, 3)
>>> y = irlist_populate(4, 5, 6)
>>> irlist_str(irlist_append(x, y))
"<1, 2, 3, 4, 5, 6>"
```
APPENDING IRLISTS

Like most IRList questions, we can solve this using recursion.

def irlist_append(irl1, irl2):
    if irl1 == empty_irlist:
        return irl2
    first = irlist_first(irl1)
    rest = irlist_append(irlist_rest(irl1), irl2)
    return make_irlist(first, rest)

If the first list is empty, we return the second list.

First item of the first list

followed by...

The rest of the first list appended to the second list.
PRACTICE: USING IRLISTS

Write the function `sorted_insert`, which takes a number and a sorted IRList of numbers and returns a new sorted IRList with the number inserted into the sorted sequence at the proper place.

```python
>>> x = irlist_populate(1, 3, 6, 9)
>>> irlist_str(sorted_insert(5, x))
"<1, 3, 5, 6, 9>"
```
**Practice: Using IRLists**

Write the function `sorted_insert`, which takes a number and a sorted IRList of numbers and returns a new sorted IRList with the number inserted into the sequence.

```python
def sorted_insert(num, sorted_irl):
    if num < irlist_first(sorted_irl):
        return make_irlist(num, sorted_irl)
    first = irlist_first(sorted_irl)
    rest = sorted_insert(num, irlist_rest(sorted_irl))
    return make_irlist(first, rest)
```
ANNOUNCEMENTS

• Homework 4 is due **July 3**.
• Homework 5 is due **July 6**.
• Project 2 is due **July 13**.
• No class tomorrow, **July 4**.
• Project 1 contest is on!
  – *How to submit*: Submit a file pig.py with your `final_strategy` to proj1-contest.
  – *Deadline*: Friday, **July 6** at **11:59pm**.
  – *Prize*: One of 3 copies of *Feynman* and 1 extra credit point.
  – *Metric*: We will simulate your strategy against everyone else’s, and tally your win rate. Draws count as losses.
ANNOUNCEMENTS: MIDTERM 1

• Midterm 1 is on **July 9**.
  – *Where*? 2050 VLSB.
  – *When*? 7PM to 9PM.
• Closed book and closed electronic devices.
• One 8.5” x 11” ‘cheat sheet’ allowed.
• Group portion is 15 minutes long.
• Post-midterm potluck on Wednesday, **July 11**.
COMMON HIGHER ORDER FUNCTIONS FOR SEQUENCES

There are a few very common styles of functions for interacting with sequences.

map

```python
>>> nums = (1, 2, 3, 4, 5)
>>> tuple(map(lambda x: x * x, nums))
(1, 4, 9, 16, 25)
>>> tuple(map(lambda x: x + 1, nums))
(2, 3, 4, 5, 6)
```

Note: The output of `map` is not a tuple, but instead a “map object.” Python does this for efficiency reasons and it is an example of a stream, which we will see towards the end of the course.
COMMON HIGHER ORDER FUNCTIONS FOR SEQUENCES

There are a few very common styles of functions for interacting with sequences.

**filter**

```python
>>> nums = (1, 2, 3, 4, 5)
>>> tuple(filter(lambda x: x % 2 == 0, nums))
(2, 4)
>>> tuple(filter(lambda x: x <= 3, nums))
(1, 2, 3)
```

*Note:* Like map, the output of filter is not a tuple, but instead a “filter object.” Python does this for efficiency reasons and it is an example of a *stream*, which we will see towards the end of the course.
COMMON HIGHER ORDER FUNCTIONS
FOR SEQUENCES

There are a few very common styles of functions for interacting with sequences.

reduce

```python
>>> from functools import reduce
>>> nums = (1, 2, 3, 4, 5)
>>> reduce(lambda x, y: x * y, nums, 1)
120
>>> reduce(lambda x, y: x + y, nums, 0)
15
```
BREAK

Recursion!
Often we want to associate pieces of data with other pieces of data.
IMMUTABLE DICTIONARIES

```python
>>> phone_bk = make_idict((“Ozzy”, “555-5555”),
... (“Tony”, “123-4567”),
... (“Geezer”, “722-2284”))
>>> idict_select(phone_bk, “Ozzy”)
“555-5555”
>>> idict_select(phone_bk, “Geezer”)“722-2284”
>>> idict_keys(phone_bk)
(“Ozzy”, “Tony”, “Geezer”)```
def make_idict(*mappings):
    return mappings

def idict_select(idict, key):
    for mapping in idict:
        if key == mapping[0]:
            return mapping[1]

def idict_keys(idict):
    return tuple(map(lambda mapping: mapping[0], idict))
EXAMPLE: IMMUTABLE DICTIONARIES

Say I wanted to remove an entry from my dictionary. Let’s write the function `idict_remove`, which takes an IDict and a key and returns a new IDict with that key removed.

```python
>>> d = make_idict((“A”, 1), ...
                  (“B”, 2), ...
                  (“C”, 3))
>>> idict_select(d, “B”)
2
>>> d = idict_remove(d, “B”)
>>> idict_select(d, “B”)  # Returns None
```
**Example: Immutable Dictionaries**

We can solve this by focusing on what *is* going in the new IDict, rather than thinking about removing an item from the group.

```python
def idict_remove(id, rm_key):
    kv_pairs = ()
    for key in idict_keys(id):
        val = idict_select(id, key)
        if key != rm_key:
            kv_pairs += ((key, val),)
    return make_idict(*kv_pairs)
```
PRACTICE: IMMUTABLE DICTIONARIES

Say instead I wanted to have a function to add a new item to the dictionary. Write `idict_insert`, which takes an IDict, a key, and a value and returns a new IDict with this update.

```python
>>> d = make_idict(("A", 1), ("B", 2), ("C", 3))
>>> idict_select(d, "Z")  # Returns None
>>> d = idict_insert(d, "Z", 55)
>>> idict_select(d, "Z")
55
>>> d = idict_insert(d, "B", 42)
>>> idict_select(d, "B")
42
```
**Practice: Immutable Dictionaries**

Say instead I wanted to have a function to add a new item to the dictionary. Write `idict_insert`, which takes an IDict, a key, and a value and returns a new IDict with this update.

```python
def idict_insert(id, new_key, new_val):
    kv_pairs = ()
    for key in idict_keys(id):
        val = idict_select(id, key)
        if key != new_key:
            kv_pairs += ((key, val),)
    kv_pairs += ((new_key, new_val),)
    return make_idict(*kv_pairs)
```
CONCLUSION

• Map, Filter, and Reduce are very common higher order functions for working with sequences.

• Dictionaries are a useful way of mapping one set of data to another.

• **Preview**: Hierarchical Data!
EXTRAS: GENERATOR EXPRESSIONS

As you might imagine, the idea of mapping and filtering through a sequence to produce a new sequence is extremely useful.

Python’s got an app a syntax for that!

<expr> for <var> in <sequence> [if <boolean expr>]
EXTRAS: GENERATOR EXPRESSIONS

>>> (x for x in (1, 2, 3))
<generator object <genexpr> at 0x01771968>
>>> tuple(x for x in (1, 2, 3))
(1, 2, 3)
>>> tuple(x * x for x in range(10))
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
>>> tuple(x for x in range(10) if x % 2 == 0)
(0, 2, 4, 6, 8)
>>> tuple(x * x for x in range(10) if x % 2 == 0)
(0, 4, 16, 36, 64)