Wednesday, Oct. 28 Nicole White

Insertion sort:

- Helper function: given a sorted list and an element, return the list that results from inserting the element where it ought to go

```
TEMPLATE
def insert(e,L):
        # if L == Empty:
                # base case
        # else:
                # ...e...L.hd...insert(e,L.tl)...

def insert(e,L):
        if L == Empty:
                return List (e,Empty) # inserting e into an empty list gives the list 'e'
        elif e < L.hd:
                return List(e, L)
        else:
                return List(L.hd, insert(e, L.tl))
# Inserts element 'e' into a sorted list so that the list remains sorted
```

Now: we need a function that repeatedly inserts elements into a sorted result:

```
def insertion_sort(L):
        if L == Empty:
                return Empty
        else:
                return insert(L.hd, insertion_sort(L.tl))

>>> insertion_sort(List(5,List(3,List(9,List(4,List(1,Empty))))))
List(1, List( 3, List(4, List(5, List(9, Empty))))))
```

<u>What kinds of lists will this work for?</u>

- numeric lists
- strings?

```
        >>> 'abc' < 'def'  # This works!
        True
```

- Booleans?

```
        >>> True > False  # This works too!
        True
```

It will *not* work for lists containing objects we have built ourselves (e.g. Point), because Python does not know by default how to compare these values

How to do this: add an extra parameter to the function (the parameter will itself be a function that compares e1 and e2, and answers True if e1 should be considered less than e2, and False otherwise)

```
def insertion_sort(L, lessthan):
    def insert(e,L):
        if L == Empty:
            return List(e,Empty)
        elif lessthan(e, L.hd):
            return List(e,L)
        else:
            return List(L.hd, insert(e,L.tl))
    if L == Empty:
        return Empty
    else:
        return insert(L.hd, insertion_sort(L.tl))
```

Now:
>>> insertion_sort(L, lambda x,y: x<y)
gives a list sorted in ascending order

>>> insertion_sort(L, lambda x,y: x>y)
gives a list sorted in descending order

What if we want to sort cards, descending by rank?
>>> insertion_sort(hand, lambda c1,c2: c1.rank > c2.rank)

Now we can also find the highest rank in a hand of cards:

```
def highest_rank(hand):
    return insertion_sort(hand, lambda c1,c2: c1.rank > c2.rank).hd.rank
```

NEW TOPIC: Built-in Data Structures

Suppose you want to write a function that returns 2 things

e.g. Find the max and the min values in a non-Empty list
- we could write 2 functions, but this requires that a list be scanned twice
- the more efficient way is to find both values in a single scan through the list

```
def max_min_list(L):
        if L.tl == Empty:
                # ...L.hd...
        else:
                # ...L.hd...max_min_list(L.tl)...
```

- in order for this to work, the recursive call max_min_list(L.tl) would need to return 2 values (the max and the min)

How to do this:
(1) declare a class (e.g., Point)
(2) NEW: return a *pair*
        If you write (x,y) in Pyhon, it simply denotes the pair of numbers x and y

```
def max_min_list(L):
        if L.tl == Empty:
                return (L.hd, L.hd)
        else:
                (mx,mn) = max_min_list(L.tl)
                newmax = max(L.hd, mx)
                newmin = min(L.hd, mn)
                return (newmax, newmin)
```

What can we do with pairs?
- operation on pairs is called INDEXING

```
>>> p = (3,5)
>>> p[0] # retrieve the 0th (first) element of the pair p
3
>>> p[1]
5
>>> p[2]
crash ... IndexError (another type of exception)
```

So, we have an alternative to classes: when should we use one, and when should we use the other?
- use a class when the data being grouped together has some meaning when considered as a collective (e.g., Point, Card, Contact)
- use a pair when the grouping is more arbitrary - when the data do not belong together and are not likely to be treated together (this saves overhead on writing useless classes)