Class 11 - Notes

Schedule

Project 3 is due on Friday, 26 February (moved back because of the snow day). Read the collaboration policy carefully — you may work either alone or with a partner on Project 3 if you want, but see project page for details.

If you don't want to risk being scooped by an 8th-grader, you should definitiely read Chapter 6: *Machines* of the course book before Friday's class.

Computer Science Distinguished Alumni Speaker: **Jason Mars**, *Let's Get Sirius*. Friday at 3:30pm in Rice 130 (followed by reception). For some background, see www.jasonmars.org (or embedded video on on-line notes).

Higher-Order Functions

A **higher-order function** is a function that manipulates functions. A function can take functions as inputs (you've seen this already in Project 1!), and can return functions as outputs. As you will see in Project 3, being able to pass around and return functions like this is very powerful.

Making Pairs without Lists

```
def make_pair(a, b):
    def selector(which):
        if which:
            return a
        else:
            return b
    return selector

def pair_first(pair):
    # Finish this code:



def pair_last(pair):
    # Finish this code:
```

Programming with Lists and Functions

```
def list_increment(lst):
    """
    Returns a new list that contains all the elements in the input list incremented by 1.
    For example, list_increment([1, 2, 3]) should return [2, 3, 4].
    """
```

```
def list_double(lst):
    """
    Returns a new list that contains all the elements in the input list doubled.
    For example, list_increment([1, 2, 3]) should return [2, 4, 6].
    """
```

```
def increment(n): return n + 1

def list_map(fn, lst):
    """
    Returns a new list that contains the result of applying fn to all the
    elements in the input list.

    For example, list_map(increment, [1, 2, 3]) should return [2, 3, 4].
    """
```

Show how to define `list_double` using `list_map`:

Yellow Belts: List Procedure Practice

These problems are for "yellow belts" (not yet confident you can define recursive procedures that operate on lists).

```
def list_sum(lst):
    # Returns the sum of the elements in lst.
```

```
def list_max(lst):
    # Returns the maximum (according to >) element in the lst.
```

Define a function, `is_list(obj)` that returns `True` if and only if the input object is a list.

```
def is_list(obj):
```

```
def list_flip(lst):
    # Returns a new list, where each element
    # is the negation of the elements in the input lst.
```

Green Belts Only: Fun with List Comprehensions

These problems are for "green belts" (confident about list procedures and able to complete Project 2, even if you haven't been promoted yet). Please don't get confused by this until you are very comfortable writing recursive procedures on lists (and can solve the "Yellow Belt" problems).

Unlike other Python constructs that we will define precisely, for list comprehensions it is best to build an intuition for what they do with experiments.

Here are some examples using list comprehensions.

Try to guess what they do, and then try evaluating them in your Python interpreter. Assume the variables `lst`, `lst1`, and `lst2` have been initialized to lists, e.g., `lst = [1, 2, 3]`, but you should figure out what these would do on any list inputs.

```
[m for m in lst]

[-m for m in lst]

[abs(m) for m in lst]

sum([1 for e in lst])

[m1 + m2 for m1, m2 in zip(lst1, lst2)]  # figure out what zip does separately

[m1 == m2 for m1, m2 in zip(lst1, lst2)]

sum([not m1 == m2 for m1, m2 in zip(lst1, lst2)])
```

Comprehending List Comprehensions. Rewrite your implementations of `sequence_complement`, `count_matches`, and `hamming_distance` as one-line list comprehensions.

Triple Gold Star Challenge. Write `edit_distance` using a list comprehension (and no loops or recursive calls).

