CSC148, Lab #4
week of October 7th, 2013

This document contains the instructions for lab number 4 in CSC148H1. To earn your lab mark, you must actively participate in the lab. We mark you in order to ensure a serious attempt at learning, NOT to make careful critical judgments on the results of your work.

General rules

We will use the same general rules as for the first lab (including pair programming). See the instructions at the beginning of Lab 1 to refresh your memory.

Overview

In this lab, you will practice writing recursive functions.

Recursive functions model the solution to recursive problems — problems that can (mostly) be broken down into instances of a similar problem. Most of the work is done up front — thinking about the solution before you begin writing a lot of code.

- Think about how the function you are writing sould be called, and write down a few examples. In fact, do yourself a favour and write the function header and docstring with some examples:

  ```
  def f(n: int) ->:
     """
     Single-sentence describing f

     >>> f(3)
     expected return of f(3)
     ...
     """
  ```

  This is extremely important: it helps to ensure you have a very clear sense of the exact use of the function.

- Consider what your function does in the **general** case — when it combines one or more calls to itself, and try to write this down.

- Consider what your function does in those **special** cases where the problem isn't broken down into smaller instances of the same problem.

- Figure out how to combine the general and special cases.

If you get stuck after you've come up with some test cases and thinking about the general and special (AKA base cases), ask for help.

Binary notation

In the base 2 number system there are only two possible digits: 0 and 1. These are often called bits (for **bi**nary digi**ts**). Each bit is an increasing power of 2 (that is, each bit counts for twice as much as the bit to the right), so the binary number 101 represents the value:

$$(1 \times 4) + (0 \times 2) + (1 \times 1) = 5$$

Here is a table of the first 8 non-negative numbers in both binary and decimal.

Decimal	Binary string
0	"0"
1	"1"
2	"10"
3	"11"
4	"100"
5	"101"
6	"110"
7	"111"
8	"1000"

Your task

(Student s1 drives, student s2 navigates)

Start Wing in a new directory called `lab04` and open a new file called `recursive_examples.py`. Before you start solving the problem below, type:

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

. . . at the end of of your file, so that it will later automatically test all the examples you put in your docstrings. This is lightweight unit testing.

To convert a non-negative integer to its binary representation, first think about its right-most binary digit (bit). This digit is always 1 if the number is odd, and 0 if the number is even. So part of converting a non-negative integer to binary is simply finding the remainder after dividing the number by 2 — use the `%` operator.

The next thing in converting a non-negative integer to base-2 is to think about how the rest of the number — the part that will make up the bits to the left of the right-most bit — gets converted. Well, removing the right-most bit of a binary number is equivalent to integer dividing it by 2 (divide and round-down). So all the bits to the left of the rightmost digit represent our original number integer divided by 2.

And that's almost all there is to it. If n is a non-negative integer, you represent it as a binary string by combining the binary representation of $n//2$ with the binary representation of $n\%2$.[1]

[1] In Python 3 // is the integer division operator

The special cases are numbers 0 and 1, which have binary representations "0" and "1", respectively.

Use these ideas to complete the implementation of function `bin_rep` below. You can (should) add examples to the docstring.

```
def bin_rep(n: int) -> str:
    """
    Produce the binary string representing non-negative integer n in binary.

    >>> bin_rep(0)
    '0'
    >>> bin_rep(1)
    '1'
    >>> bin_rep(5)
    '101'
    """
```

If you get stuck, call your TA or a classmate over. If you're not stuck, trace through some small examples until you have intuition for how the code works. At each step, treat the calls to `bin_rep` on smaller numbers as black boxes or solved problems — something you've already seen works and don't need to trace any further.

Be sure to show your TA your work when you're finished.

Structured lists

(Student s2 drives, student s1 navigates)

You have likely used the Python operator "`in`" to determine whether there is an element in a list with the same value as an element you're seeking, for example

```
>>> 3 in [1, 2, 3, 5]
True
>>> 3 in [1, 2, 5]
False
>>>
```

You may have enough intuition now to search a nested list to determine whether it has an equivalent element. However, such a search would likely take work that is proportional to the size of the list — you would never really know you were done until you had examined each list element.

Consider the advantage of particularly structured nested lists, which I'll call `SearchLists`. These are defined:

SearchList: A python list of 3 elements

(0) Element 0 is an integer

(1) Element 1 is either a `SearchList` containing integers smaller than Element 0, or `None`

(2) Element 2 is either a `SearchList` containing integers larger than Element 0, or `None`

Here's a small example of a `SearchList`

```
[5, [2, [1, None, None], [3, None, None]], [6, None, None]]
```

Use the properties of a SearchList to figure out how to determine whether a given integer is somewhere in the SearchList or not. Notice that if a given integer is less than element 0, it is either in element 1 somewhere, or not in the SearchList all. Similarly, if a given integer is more than element 0, it is either in element 2 somewhere, or not in the SearchList at all.

Use the ideas to complete the implementation of find_num below. Don't forget to start by writing some examples in the docstring!

```
def find_num(SL: 'SearchList', n: int) -> bool:
    """
    Return True if n is in SL, False otherwise

    >>> ... some examples, please!
    ...
    ...
    """
```

If you get stuck, call your TA over for some hints.

If you're not stuck, trace through simple examples — un-nested lists — to convince yourself your code works. Then trace through the next most-complicated example — a list containing at least one un-nested list — to convince yourself that your code works. Remember to treat the simple cases you've already traced as black boxes, or already solved problems. Then show your TA your work.

Freezing list copies

(Student s1 drives, student s2 navigates)

Usually, when you copy a Python list you store a reference to it. This means that if the original list changes, your reference leads you to the changed list, and often you want this behaviour.

Sometimes, however, you'd like a copy of all the values of a list where all its elements (and elements of elements, if the list contains sub-lists) remain as they were at the moment you copied them, not subject to changes that might be invoked in some other part of your code, or even somebody else's code. Python provides a deep copy function, copy.copy

Here's an idea. It's not as complete, or difficult, as copy.copy (for example, it won't deal with lists that contain references to themself). But it's a first cut. Produce a new list (think list comprehension!) by assigning new elements as follows:

1. If the corresponding old element is a non-list, simply make a reference to it (this is what normally happens when we assign expressions in Python)
2. If the corresponding old element is a list, treat it just as you do its enclosing list — produce a new list by assigning new elements as elements as in these two steps (think recursion!)

Use your ideas to complete the implementation of the function freeze below. Here's a small example of how it should work:

```
>>> L1 = [1, [2, 3], 4]
>>> L2 = freeze(L1)
>>> L1 is L2
False
>>> L1[1] is L2[1]
False
```

```
    >>> L1 == L2
    True
    >>> L1[1] == L2[1]
    True

def freeze(X: object) -> object:
    """
    If X is a list, return a new list with equivalent contents,
    and recursively treat the contents of X as you treated X itself...
    If X is not a list, return X itself

    >>> ... don't forget examples!
    ...
    ...
    """
```

If you're stuck, call your TA over to show your work.

If you're not stuck, trace through simple examples — un-nested lists — to convince yourself your code works. Then trace through the next most-complicated example — a list containing at least one un-nested list — to convince yourself that your code works. Remember to treat the simple cases you've already traced as black boxes, or already solved problems.

Be sure to show your TA your work.

1 Extra, optional problems

If you finish with the problems above, here are some other ones to try. These are certainly not required for completing the lab.

- Write a recursive function `rev_string(s)` that produces the reversal of string `s`.
- Extend `bin_rep(n)` to create `base_rep(n,k)` to give a string representing non-negative integer `n` in base `k`, where $2 \leq k < 10$. In base `k`, the only digits allowed are $\{0, \ldots, k-1\}$.
- Extend `freeze(L)` so that it also handles tuples and dictionaries.
- For a list of distinct integers, `L`, define `switches(L)` as the number of pairs in `L` that are not in increasing order. For example, `switches([3,1,2])` returns 2, since (3, 1) and (3,2) are out of order. Implement the function `switches(L)` recursively.