University of Washington
CSE 140 Data Programming
Winter 2013

Final exam

March 11, 2013

Name: ***Solutions***

Section: _____

UW Net ID (username): _____

This exam is closed book, closed notes. You have **50 minutes** to complete it. It contains 8 questions and 7 pages (including this one), totaling 100 points. Before you start, please check your copy to make sure it is complete. Turn in all pages, together, when you are finished. **Write your initials on the top of *ALL* pages** (in case a page gets separated during test-taking or grading).

Please write neatly; we cannot give credit for what we cannot read.

Good luck!

Page	Max	Score
2	20	
3	12	
4	24	
5	12	
6	16	
7	16	
Total	100	

1 Short answer

1. (6 points) State the steps you should follow when writing a function. Write just one phrase for each step. (These steps come after you have defined the problem and decomposed it into parts, and before you run tests.) Depending on how you break up the tasks, you might write 3, 4, or 5 steps; you can get full credit for any of these answers as long as your steps include all the necessary parts and no extraneous parts.

   (a) ***choose a name***

   (b) ***choose input and output types***

   (c) ***write the specification (documentation string)***

   (d) ***write tests***

   (e) ***write the implementation and code comments***

   It is acceptable to combine some or all of the first three steps above. It is acceptable to state that more tests should be written after the implementation.

2. (6 points) Name the two types of documentation, and state for whom each one is intended.

   (a) ***Specification***
   Intended for: ***programmers who are creating clients/users of your module***

   (b) ***Code comments***
   Intended for: ***programmers who are debugging/enhancing the implementation***

3. (8 points) State three advantages of splitting up your code into functions. Give reasons that are as different as possible. Write no more than one sentence each (a phrase each is enough).

   (a) ***Documentation: Gives a comprehensible name to a computation.***

   (b) ***Abstraction: Clients can depend on a short specification, and ignore details of the implementation.***

   (c) ***Re-use: In this program or programs you might write in the future.***

   (d) Testing and debugging: enables divide and conquer.

2 Environment diagram

Here are the powers of 5, for your information:

$5^0 = 1$
$5^1 = 5$
$5^2 = 25$
$5^3 = 125$
$5^4 = 625$
$5^5 = 3125$
$5^6 = 15625$

4. (12 points) Consider the following code:

```
def exp(base, exponent):
    """Return base to the exponent power.
       Exponent is a non-negative integer."""
    if exponent == 0:
        return 1
    if exponent % 2 == 0:
        return exp(base*base, exponent/2)
    return base [*] exp(base, exponent - 1)
```

Given the expression `exp(5, 6)`, show the stack frames when the multiplication on the last line of code (the multiplication with the box around it) is about to be executed for the first time.

global:
exp -> fn

exp:
base: 5
exponent: 6

exp:
base: 25
exponent: 3

exp:
base: 25
exponent: 2

exp:
base: 625
exponent: 1

Note that the first time the last line of code is encountered (in the second exp stack frame) is not the first time the multiplication on that line is executed.

What are the two arguments to the multiplication referenced in the last question?

625 ***1***

3 Testing

5. (24 points) Consider the following function specification:

```
def indices_of(lst, elt):
    """Return a sorted list of the indices at which the element appears in lst."""
```

Here is an example test:

```
assert indices_of([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3], 9) == [5, 12, 14]
```

A good test suite contains tests that are different from one another.

Apply the "0–1–many" principle to the `indices_of` function. Write three different tests, and state what is varying between them.

What is varying: ***Length of `lst` argument***

Test 1: *`assert indices_of([], 1) == []`*

Test 2: *`assert indices_of([1], 0) == []`*

Test 3: *`assert indices_of([1, 2, 3], 0) == []`*

Now, use the "0–1–many" principle again, but apply it to something different. Again, write three different tests, and state what is varying between them.

What is varying: ***Length of result (equivalently, number of times `elt` appears in `lst`)***

Test 1: *`assert indices_of([2, 1, 2], 0) == []`*

Test 2: *`assert indices_of([2, 1, 2], 1) == [1]`*

Test 3: *`assert indices_of([2, 1, 2], 2) == [0, 2]`*

Another thing to apply the "0–1–many" principle to is position of `elt` in `lst`: beginning, middle, or end.

When varying one thing, it's best to avoid varying other things, to reduce the number of things to check if a test does not succeed. That's why in the first set of tests above, the output value does not change, and for the second set of tests above, the input list does not change.

Partial credit for varying the type of the list elements and `elt`: this does create diversity, but not diversity that is likely to expose errors and not diversity

No credit for "different values": of course the values will differ, but not all choices fit the 0–1–many pattern or are likely to detect errors. The 0, 1, and many cases must be chosen with care: they should induce different program behavior, but `indices_of` doesn't depend on (for example) whether the list elements are negative or positive.

Write an implementation of `indices_of`, in a single line of code.

Hint: use a list comprehension.

`return [idx for idx in range(len(lst)) if lst[idx] == elt]`

A common incorrect answer was:
`return [list.index(num) for num in lst if num == elt]`
With this body,
`indices_of([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3], 9)`
evaluates to `[5, 5, 5]`. Do you see why?

4 Write code

6. (12 points) Write the body of the following function.

```
def word_frequency(list_of_tuples):
    """Return a dictionary mapping words to numbers.
       The input is a list of (word, number) tuples.
       In the output dictionary, each word maps to the sum of all numbers it
       co-occurs with in the input list."""

    result = {}
    for (word, num) in list_of_tuples:
        if not (word in result):
            result[word] = 0
        result[word] += num
    return result
```

Here is a test case, to help you understand the specification:

```
assert word_frequency([("to", 2), ("be", 2), ("or", 2), ("not", 3), ("to", 2),
  ("be", 2)]) == {'be': 4, 'not': 3, 'or': 2, 'to': 4}
```

7. (16 points) Write a class `RatNum` that represents rational numbers. A rational number is the ratio of two integers; examples are $\frac{1}{2}$ and $\frac{22}{7}$. You do not have to write any documentation strings.

   Once your class is defined, then running the following code:

```
one_half = RatNum(1,2)
one_third = RatNum(1,3)
five_sixths = one_half.add(one_third)
print one_half.to_float()
print one_third.to_float()
print five_sixths.to_float()
```

   should print

```
0.5
0.333333333333
0.833333333333
```

   Hint: Do not define any more methods than necessary.

   Hint: Recall from middle school that $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$.

   Hint: You do not have to deal with client errors, such as using zero as the denominator.

   Hint: Do not think about reducing fractions to lowest terms. That is not necessary for this problem. (If you don't understand this hint, then don't worry about it.)

```
class RatNum:
    """Represents a rational number such as 1/2 or 22/7."""
    # Representation:  two integers numerator and denominator, not necessarily
    # in reduced form.

    def __init__(self, num, denom):
        """Construct a RatNum with the given numerator and denominator."""
        self.numerator = num
        self.denominator = denom

    def add(self, other):
        """Return a new RatNum that is the sum of this RatNum and RatNum other."""
        return RatNum(self.numerator * other.denominator
                      + other.numerator * self.denominator,
                     self.denominator * other.denominator)

    def to_float(self):
        """Return a float that approximately equals the value of this RatNum."""
        return float(self.numerator) / self.denominator
```

8. (16 points) Write the body of the following function. Write the function twice: iteratively (using a loop) and recursively (using a recursive call). You may solve the two parts of this problem in either order. You may use the function `max(x, y)` that takes two numbers and returns the larger one.

```
## Iterative version
def list_max(numbers):
    """Returns the largest value that appears in the argument list.
       The argument is a non-empty list of numbers."""

    result = numbers[0]
    for num in numbers:
        result = max(result, num)
    return result


## Recursive version
def list_max(numbers):
    """Returns the largest value that appears in the argument list.
       The argument is a non-empty list of numbers."""

    if len(numbers) == 1:
        return numbers[0]
    return max(numbers[0], list_max(numbers[1:]))
```