CS 1110, LAB 10: ASSERTIONS AND WHILE-LOOPS

http://www.cs.cornell.edu/courses/cs1110/2014sp/labs/lab10.pdf

1. Preliminaries

This lab gives you practice with writing loops using *invariant*-based reasoning. Invariants document the meaning of variables, typically in terms of how they relate to one another, and so help you reason about the correctness of code.

1.1. **Reference material.** The slides on ranges on the handout for lecture 19 (while-loops); Section 7.3 of the textbook, "The `while` statement". If you like to read ahead, it may also be useful to consult http://www.cs.cornell.edu/courses/cs1110/2013sp/materials/loop_invariants.pdf, "Worked examples regarding loops and invariants".

1.2. **Getting Credit.** Show your lab instructor this handout and your changes to `lab10.py`.

What you don't finish during the lab session is homework that you'll need to get checked off in TA office hours or consulting hours by, at the latest, **Monday April 14th**. Remember that labs are "graded" on effort, not correctness.

Check-off checklist:

- ☐ You have out this handout with the answer boxes (except for those in the optional appendix) filled in to the best of your ability.
- ☐ You have your completed lab10.py open in Komodo Edit.
- ☐ You have a command shell open in the directory your lab 10 code is in, so you can demonstrate the running of your code.

2. Written Exercises

2.1. **Motivation: an idea for counting counting occurrences of a character.** Your first coding task will be to complete function `count_letter(c,s)`, which returns the number of times lowercase character `c` occurs in lowercase string `s`.[1]

You will do so via code based on the following *invariant* on variables `n` and `i`:

> Invariant: Character `c` occurs `n` times in `s[0..i-1]` (meaning the Python slice `s[0:i]`).

The main idea is that if we make sure the invariant is true at the outset, and then we perform actions so that the invariant holds true for larger and larger `i`, then eventually we'll know the following is true:

Lab authors: D. Gries, L. Lee, S. Marschner, W. White

[1] This is just for practice; it's not a particularly useful function to write because Python already has the string method `count` built in.

Postcondition (special case of the invariant): Character `c` occurs `n` times in `s[0..len(s)-1]` (which is `s`).

If that's true, we just have to return `n`, and we are done.

What's all this business about invariants, postconditions, and the like? The written exercises below will help you get comfortable with quickly reasoning these things.

2.2. **Questions on Ranges.** From the fact that the range h..k=h, h+1, h+2, ..., k contains $k + 1 - h$ values[2], how many values are in the following ranges?

Given range	Number of values in it	Given range	Number of values in it
5..7		h..h+1	
5..6		h..h	
5..5		h..h-1	
5..4		0..i-1	

2.3. **Assigning to Range Variables.** Each line below asks you to write an assignment statement. We have done the first one for you to give you an idea of what we are looking for.

Range	Want	Assignment Statement
`h..k`	Assign to `k` so that the range has 1 element	`k = h`
`h..k`	Assign to `h` so that the range has 1 element	
`h..k`	Assign to `k` so that the range has 0 elements	
`h..k`	Assign to `h` so that the range has 0 elements	
`0..n`−1	Assign to `n` so that the range has 0 elements	

2.4. **Completing Assertions.** Each line below contains an assertion P that is guaranteed to be true. Each line also contains an assertion R, which we would like to be true. In the righthand column, write down a claim that, when true, allows us to conclude that R is true. We have filled in the first one for you.

We use s[i..j] to denote the portions of s running from index i to index j inclusive. For example, if s were 'abcdef', then s[3..5] is 'def'. Remember that this is *different* than Python's slice notation; s[i..j] is equivalent to Python's `s[i:j+1]`!

[2]Recall that in the notation `h..k`, we require `k` $\geq$ `h`−1. For example, 5..4 is OK but 5..3 is not allowed.

Know P	Want R	Additional Info Needed
x is the sum of `1..n`	x is the sum of `1..100`	n is 100
x is smallest element of the segment `s[0..k-1]`	x is smallest element of the segment `s[0..len(s)-1]`	
x is the product of `k..n`	x is the product of `1..n`	
b is True if nothing in `h..k` divides `x`; False otherwise	b is True if nothing in `m..k` divides `x`; False otherwise	
`x` is the sum of `a[h..i]` and `y` is the sum of `a[j..k]`	`x + y` is the sum of `a[h..k]`	

2.5. **Preserving Invariants.** Below is a *precondition* P, an assignment to a variable, and the same assertion P as a *postcondition*. Where indicated, place a statement so that if P is true initially, it will be true afterward (as indicated by the postcondition comment). The statement can be in English, if you are not sure how to write it in Python, but make it a command to do something. In the exercises below, `v` is a list of `int`s. We've done the first one for you.

```
(a) # P: x is the sum of 1..n
    # Put a statement here:
```

Answer `x = x + (n+1)`

```
    n = n + 1
    # P: x is the sum of 1..n
```

```
(b) # P: x is the sum of h..100
    # Put a statement here:

    h = h - 1
    # P: x is the sum of h..100
```

```
(c) # P: x is the minimum of v[0..k-1]
    # Put a statement here:

    k = k + 1
    # P: x is the minimum of v[0..k-1]
```

```
(d) # P: x is the minimum of v[h..100]
    # Put a statement here:

    h = h - 1
    # P: x is the minimum of v[h..100]
```

2.6. **A while-loop for counting occurrences of a character.** Let's get back to function `count_letter(c,s)`, which returns the number of times lowercase character `c` occurs in lowercase string `s`. Consider the following *invariant* on variables `n` and `i`:

> Invariant: Character `c` occurs `n` times in `s[0..i-1]` (meaning the Python slice `s[0:i]`).

This means that `i` marks the "index" in `s` of where we stop necessarily knowing anything about the number of `c`s that are contained there — it marks the beginning of the "unknowns". ("Index" is in quotes because we might allow `i` to be `len(s)`.)

Again, the main idea is that if we make sure the invariant is true at the outset, and then we perform actions so that the invariant holds true for larger and larger `i`, then eventually we'll know the following is true:

> Postcondition (special case of the invariant): Character `c` occurs `n` times in `s[0..len(s)-1]` (which is `s`).

If that's true, we just have to return `n`, and we are done.

Consider the following body:

```
0    i = 0
1    n = 0
2    while i < len(s):
3         if s[i] == c:
4             n += 1
5         i += 1
6    return n
```

Why does the invariant hold just after the execution of line 1, when `i` is 0? Your explanation should make mention of what `s[0..i-1]` is when `i` is 0, and what the initial value of `n` is.

If we know that character `c` occurs `n` times in `s[0..i-1]`, and we know that `s[i] == c`, then how many times does `c` occur in `s[0..i]`? And why should we increment `i` in that case?

If we know that character `c` occurs `n` times in `s[0..i-1]`, and we know that `s[i]` is some character other than `c`, then how many times does `c` occur in `s[0..i]`? Does this explain why there's no `else` clause in the code above? And why do we increment `i` in this case too?

When you reach line 6, what is the value of `i`? And why does that mean we can return n?

3. Coding Exercises

Start a new folder and download `lab10.py` into this folder. Complete the body of the two functions there according to their specifications, the implementation comments, and, especially, the invariants we give you.

3.1. **isprime.** This function tests where its input is a prime number. The idea is to keep checking for possible divisors of the input until either a nontrivial divisor is found — in which case the input isn't prime — or the set of possible divisors has been exhausted without any being found — in which case the input is prime.

We chose the loop condition for simplicity, not efficiency.

We gave you a few test cases in lab10.py, but they aren't exhaustive.

3.2. **exp_pair.** Continuing with this semester's theme of card tricks, we ask the question, "How many draws from a shuffled deck does it take, on average, until you get two of a kind?" It seems easier to answer this question through simulation rather than analytically.

We've given you a new module, card2, which is the same as the `card` module from the previous lab except that function full_deck returns a *shuffled* deck. We've also given you quite a bit of code to start with, but you need to figure out what needs to be filled in.

You can check or debug your code by typing the following into a command shell:
`python lab10.py 5 True`
Here's what output from a sample run of our solution looks like:

```
10
1
13
6
8
7
1
number of draws:  7
1
11
7
9
4
1
number of draws:  6
8
7
1
10
12
9
9
number of draws:  7
10
```

```
12
9
4
11
1
13
4
number of draws:  8
7
2
13
4
12
10
12
number of draws:  7
In 5 trials, the avg number of draws to get a pair was 7.0
```

And once everything is working, you can try 10,000 trials by typing the following in a command shell:
`python lab10`

4. Optional Extra Practice: Reasoning about a for-loop version of `count_letters`

Consider the following code:

```
0    i=-1  # need a value of i for the invariant to hold initially
1    n = 0
2    for i in range(len(s)): # last value will be len(s)-1
3        if s[i] == c:
4           n = n+1
5    return n
```

We claim this code can be analyzed in terms of the invariant `i`:

> Invariant: Character `c` occurs `n` times in `s[0..i]` (meaning the Python slice `s[0:i+1]`).

and corresponding postcondition

> Postcondition (special case of the invariant): Character `c` occurs `n` times in `s[0..len(s)-1]` (which is `s`).

If that's true, we just have to return `n`, and we are done.

Note that in contrast to the while-loop example we did above, here, `i` marks the *end* of the *known* part of the string.

Why does the invariant hold just after the execution of line 1, when `i` is -1? Your explanation should make mention of what `s[0..i]` is when `i` is -1, and what the initial value of `n` is.

The body of the loop makes the invariant hold for the newly assigned value of `i`. If we know that character `c` occurs `n` times in `s[0..i-1]`, and we know that `s[i] == c`, then how many times does `c` occur in `s[0..i]`?

If we know that character `c` occurs `n` times in `s[0..i-1]`, and we know that `s[i]` is some character other than `c`, then how many times does `c` occur in `s[0..i]`? Does this explain why there's no else clause in the code above?

When you reach line 5, what is the value of `i`? And why does that mean we can return n?