CS 1110

Lecture 19: **Loop invariants**

Announcements

Prelim 2 conflicts

Today (April 2) is two weeks before the prelim, and the deadline for submitting prelim conflicts.

Instructor travel

This week and the next two weeks, Profs. Lee and Marschner will be traveling on and off. Instructor office hours are unaffected, though there will sometimes be just one of us available.

Slides by D. Gries, L. Lee, S. Marschner, W. White

Some Important Terminology

- **assertion**: true-false statement placed in a program to *assert* that it is true at that point
  - Can either be a **comment**, or an **assert** command
- **precondition**: assertion placed before a statement
  - Same idea as **function precondition**, but more general
- **postcondition**: assertion placed after a statement
- **loop invariant**: assertion supposed to be true before and after each iteration of the loop
  - Distinct from **attribute invariant**
- **iteration of a loop**: one execution of its repetend

Assertions versus Asserts

- Assertions **prevent bugs**
  - Help *you* keep track of what you are doing
- Also **track down bugs**
  - Make it easier to check belief–code mismatches
- The assert statement is also an assertion
  - an assertion you are asking Python to enforce
  - Cannot always convert a comment to an assert

 # x is the sum of 1..n

Comment form of the assertion.

The root of all bugs!

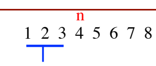
x	?	n	1
x	?	n	3
x	?	n	0

Preconditions & Postconditions

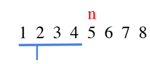
precondition

```
# x  = sum of 1..n-1
x = x + n
n = n + 1
# x =  sum of 1..n-1
```

postcondition



x contains the sum of these (6)



x contains the sum of these (10)

- **Precondition:** assertion placed before a segment
- **Postcondition:** assertion placed after a segment

Relationship Between Two

If precondition is true, then postcondition will be true

Solving a Problem

precondition

```
# x  = sum of 1..n

n = n + 1
# x =  sum of 1..n
```

postcondition

What statement do you put here to make the postcondition true?

A: x = x + 1
B: x = x + n
C: x = x + n+1
D: None of the above
E: I don't know

Invariants: Assertions That Do Not Change

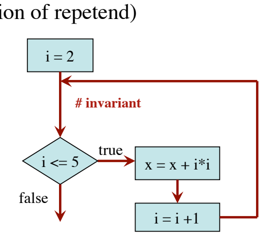
- **Loop Invariant**: an assertion that is true before and after each iteration (execution of repetend)

```
x = 0; i = 2
while i <= 5:
    x = x + i*i
    i = i +1
# x = sum of squares of 2..5
```

Invariant:

x = sum of squares of 2..i-1

in terms of the range of integers that have been processed so far



The loop processes the range 2..5

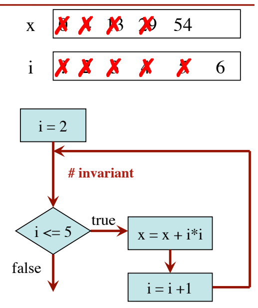
Invariants: Assertions That Do Not Change

```
x = 0; i = 2
# Inv: x = sum of squares of 2..i-1
while i <= 5:
    x = x + i*i
    i = i +1
# Post: x = sum of squares of 2..5
```

Integers that have been processed: 2, 3, 4, 5

Range 2..i-1: 2..5

Invariant was always true just before test of loop condition. So it's true when loop terminates



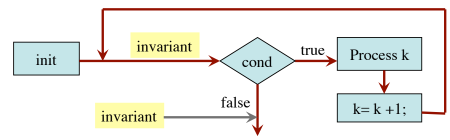
The loop processes the range 2..5

Designing Integer while-loops

```
# Process integers in a..b
# inv: integers in a..k-1 have been processed
k = a
while k <= b:
    process integer k
    k = k + 1
# post: integers in a..b have been processed
```

Command to do something

Equivalent postcondition



Designing Integer while-loops

1. Recognize that a range of integers b..c has to be processed
2. Write the command and equivalent postcondition
3. Write the basic part of the for-loop
4. Write loop invariant
5. Figure out any initialization
6. Implement the repetend (process k)

```
# Process b..c
Initialize variables (if necessary) to make invariant true
# Invariant: range b..k-1 has been processed
while k <= c:
    # Process k
    k = k + 1
# Postcondition: range b..c has been processed
```

Finding an Invariant

Command to do something

```
# Make b True if no int in 2..n-1 divides n, False otherwise
b = True
k = 2
# invariant: b is True if no int in 2..k-1 divides n, False otherwise
while k < n:
    # Process k;
    if n % k == 0:
        b = False
    k = k +1
# b is True if no int in 2..n-1 divides n, False otherwise
```

Equivalent postcondition

What is the invariant? 1 2 3 … k-1 k k+1 … n

Finding an Invariant

```
# set x to # adjacent equal pairs in s[0..len(s)-1]
# invariant: ???
k = 0
while k < len(s):
    # Process k;
    k = k +1
# x = # adjacent equal pairs in s[0..len(s)-1]
```

Command to do something

for s = 'ebeee', x = 2

Equivalent postcondition

k: next integer to process.
Which have been processed?

- A: 0..k
- B: 1..k
- C: 0..k–1
- D: 1..k–1
- E: I don't know

What is the invariant?

- A: x = no. adj. equal pairs in s[1..k]
- B: x = no. adj. equal pairs in s[0..k]
- C: x = no. adj. equal pairs in s[1..k–1]
- D: x = no. adj. equal pairs in s[0..k–1]
- E: I don't know

Reason carefully about initialization

```
# s is a string; len(s) >= 1
# Set c to largest element in s
c = ??
k = ??
# inv: c is largest element in s[0..k–1]
while k < len(s):
    # Process k
    k = k+1
# c = largest char in s[0..len(s)–1]
```

Command to do something

Equivalent postcondition

1. What is the invariant?
2. How do we initialize c and k?

- A: k = 0; c = s[0]
- B: k = 1; c = s[0]
- C: k = 1; c = s[1]
- D: k = 0; c = s[1]
- E: None of the above

An empty set of characters or integers has no maximum. Therefore, be sure that 0..k–1 is not empty. You must start with k = 1.