Recall: 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

• **invariant**: assertion supposed to "always" be true
  * If temporarily invalidated, must make it true again
  * Example: class invariants and class methods

• **loop invariant**: assertion supposed to be true before and after each iteration of the loop

• **iteration of a loop**: one execution of its body

---

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 a (type of) assertion
  * One you are enforcing
  * Cannot always convert a comment to an assert

---

Preconditions & Postconditions

<table>
<thead>
<tr>
<th>precondition</th>
<th>postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code># x = sum of 1..n-1</code></td>
<td><code>x contains the sum of these (6)</code></td>
</tr>
<tr>
<td><code>x = x + n</code></td>
<td><code>n = n + 1</code></td>
</tr>
<tr>
<td><code># x = sum of 1..n-1</code></td>
<td><code>x contains the sum of these (10)</code></td>
</tr>
</tbody>
</table>

**Relationship Between Two**

If precondition is true, then postcondition will be true

---

Invariants: Assertions That Do Not Change

• **Loop Invariant**: an assertion that is true before and after each iteration (execution of repeatend)
  
  \[
  x = 0; \ i = 2 \\
  \text{while } i <= 5: \\
  \quad x = x + i^2 \\
  \quad i = i + 1 \\
  \# x = sum of squares of 2..i-1
  \]

  **Invariant:**
  \[
  x = \text{sum of squares of } 2..i-1
  \]
  
  in terms of the range of integers that have been processed so far

---

Invariants: Assertions That Do Not Change

\[
\begin{align*}
\text{Integers that have been processed: } & 2, 3, 4, 5 \\
\text{Range } 2..i-1: & 2..5
\end{align*}
\]

**Invariant**

\[
\text{The loop processes the range 2..5}
\]
Designing Integer while-loops

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

Finding an Invariant

# Make b True if n is prime, 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

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

Finding an Invariant

# set x to # adjacent equal pairs in s
x = 0
# inv x = # adjacent equal pairs in s[0..k-1]
while k < len(s):
    # Process k
    k = k + 1
# x = # adjacent equal pairs in s[0..len(s)-1]
k: next integer to process.
What is initialization for k?
A: k = 0
B: k = 1
C: k = -1
D: I don’t know

Finding an Invariant

# set x to # adjacent equal pairs in s
x = 0
# inv x = # adjacent equal pairs in s[0..k-1]
while k < len(s):
    # Process k
    k = k + 1
# x = # adjacent equal pairs in s[0..len(s)-1]
k: next integer to process.
What is initialization for k?
A: k = 0
B: k = 1
C: k = -1
D: I don’t know

Reason carefully about initialization

1. What is the invariant?
2. How do we initialize c and k?
A: c = 0; k = s[0];
B: c = k = 1;
C: c = 1; k = s[0];
D: c = k = 0; c = s[1];
E: None of the above