CSc 120
Introduction to Computer Programming II

Adapted from slides by
Dr. Saumya Debray

04: Invariants
basic concepts
Invariants

An invariant is a predicate about the state of a program at some point in the code that should always be true if the program is running correctly.
An invariant is a predicate about the state of a program at some point in the code that should always be true if the program is running correctly.
Invariants

An *invariant* is a predicate about the *state of a program* at some point in the code that should always be true if the program is running correctly.

A *predicate* is a Boolean, i.e., is True or False.

The *state* of a program refers to:
- values of variables; and
- relationships between values of variables.
Invariants

An invariant is a predicate about the state of a program at some point in the code that should always be true if the program is running correctly.

a predicate is a Boolean, i.e., is True or False
the state of a program refers to:
• values of variables; and
• relationships between values of variables

the invariant refers to the program state when execution reaches this point in the code
An invariant is a predicate about the state of a program at some point in the code that should always be true if the program is running correctly.

An invariant is False ⇔ the code has a bug.
Invariants

An invariant is a predicate about the state of a program at some point in the code that should always be true if the program is running correctly.

A predicate is a Boolean, i.e., is True or False. The state of a program refers to:
- values of variables; and
- relationships between values of variables

The invariant refers to the program state when execution reaches this point in the code.

An invariant is False ⇔ the code has a bug.
Invariants: Why do we care?

• They help with programming
  – thinking of the invariants that need to hold can help us figure out what code we need to write

• They help with debugging
  – debugging involves identifying invariants that should hold but don't

• Useful for documentation
  – invariants (either in the code or in comments) can make it easier to understand someone else's code
Example

Definition of lookup()

```
# lookup(string, alist) - returns the position where the given string occurs in the given list.
def lookup(string, alist):
    for i in range(len(alist)):
        if string == alist[i]:
            return i
```

Use of lookup()

```
x = input().split()  # a list of strings
y = input()         # a string
z = 23

pos = lookup(y, x)
```

Q: What invariant(s) hold here?
Example

```python
def lookup(string, alist):
    for i in range(len(alist)):
        if string == alist[i]:
            return i
```

```python
x = input().split()  # a list of strings
y = input()  # a string
z = 23
pos = lookup(y, x)
```

Q: What invariant(s) hold here?
Example

def lookup(string, alist):
    for i in range(len(alist)):
        if string == alist[i]:
            return i

x = input().split()  # a list of strings
y = input()  # a string
z = 23

pos = lookup(y, x)

Q: What invariant(s) hold here?

• z == 23
  – this is an invariant, but (maybe) not relevant to lookup()

• x[pos] == y
  – this is not an invariant (why?)

• ???
Example

def lookup(string, alist):
    for i in range(len(alist)):
        if string == alist[i]:
            return i

x = input().split()  # a list of strings
y = input()  # a string
z = 23
pos = lookup(y, x)

Q: What invariant(s) hold here?

Ideally, we want something like:

```python
def lookup(string, alist):
    for i in range(len(alist)):
        if string == alist[i]:
            return i
```

This leads to a bug fix in `lookup()`:

- return some special value (e.g., None) if the string is not found in the list.
Summary 1

• There can be many different invariants at a point in a program
  – the one(s) we focus on depend on which aspects of the code we care about

• Thinking about invariants can help us figure out what code we should write
Invariants and debugging

• If a program has a bug, then by definition some invariant $I$ somewhere is broken
  – i.e., the invariant $I$ should hold but does not

• Debugging is the process of:
  – looking at the state of the program to identify where this is happening; and
  – changing the program so that the invariant $I$ holds

We usually don't think of debugging explicitly in terms of invariants, but implicitly that is what is going on.
Example

def lookup(string, alist):
    for i in range(len(alist)):
        if string == alist[i]:
            return i
    return None

Desired invariant after lookup(y,x):
    if y in x then x[pos] == y
    else pos == None

For the arguments
    x = ['ab', 'bc', 'cd']
    y = 'bc'
the invariant says it should return 1
What does lookup(y, x) return?
Example

```python
def lookup(string, alist):
    for i in range(len(alist)):
        if string == alist[i]:
            return i
    return None
```

Desired invariant after `lookup(y, x)`:
- if `y` in `x` then `x[pos] == y`
- else `pos == None`

For the arguments
- `x = ['ab', 'bc', 'cd']`
- `y = 'bc'`

the invariant says it should return 1
`lookup(y, x)` returns None

⇒ `lookup(y, x)` is returning too early with the wrong return value

⇒ leads us to examine the code for returning with None
<table>
<thead>
<tr>
<th>Buggy code</th>
<th>Fixed code</th>
</tr>
</thead>
</table>
| def lookup(string, list):
  for i in range(len(list)):
    if string == list[i]:
      return i
  return None |
| def lookup(string, list):
  for i in range(len(list)):
    if string == list[i]:
      return i
  return None |
Summary 2

Invariants are useful for debugging
• a bug $\iff$ an invariant that should hold somewhere, but in fact does not
• thinking about invariants can help us localize the problem and identify the bug

(We will discuss debugging in more detail later in the course)
figuring out invariants
Figuring out invariants

• An invariant at a program point is an expression that must be true whenever execution reaches that point
  – we want to focus on invariants that are relevant to the code
    ○ It’s OK to state only some of the things that must be true

• We start at the beginning of each function/method and work our way down its statements
Figuring out invariants: assignments

\[ x_1, ..., x_n = e_1, ..., e_n \]

\[
\begin{align*}
\bullet & \ x_1 == e_1 \text{ and } ... \text{ and } x_n == e_n \\
\bullet & \text{anything else: unchanged from before the assignment}
\end{align*}
\]
Figuring out invariants: conditionals

```python
if exp_1:
    stmt_1
elif exp_2:
    stmt_2
    not exp_1
    and exp_2
elif exp_3:
    stmt_3
    not exp_1
    and not exp_2
    and exp_3
....
```

Invariants shown in green
Figuring out invariants: conditionals

\[\text{if } \text{exp}_1 : \]
\[\quad \text{stmt}_1 \]
\[\text{elif } \text{exp}_2 : \]
\[\quad \text{stmt}_2 \]
\[\text{elif } \text{exp}_3 : \]
\[\quad \text{stmt}_3 \]
\[\ldots\]

Special case:

\[\text{if } \text{exp} :\]
\[\quad \text{stmt}_1\]
\[\text{else} : \]
\[\quad \text{stmt}_2\]

Invariants shown in green.
x = int(input())
if (x < 0):
    x = -x
#
print(x)

What is true about x for either branch taken?
Figuring out invariants: conditionals

if \( exp \):

\[ exp == \text{True} \]

\[ \text{stmt}_1 \]

\[ exp == \text{False} \]

\[ \text{stmt}_2 \]

\[ \text{stmt}_3 \]

\[ \text{not exp} \]

\[ \text{P}_1 \]

\[ \text{P}_2 \]

\text{whatever is common in both } \text{P}_1 \text{ and } \text{P}_2 \]

(NOTE: This is not the same as “\( \text{P}_1 \text{ and } \text{P}_2 \)”)

invariants shown in green
EXERCISE

```python
x = int(input())
if (x % 2 == 0):  # x is even
    y = x + 2
else:
    y = x + 1
#
print(y)
```

What are two invariants here about y?
Answer

```python
x = int(input())
if (x % 2 == 0):  # x is even
    y = x + 2
else:
    y = x + 1
    #
print(y)
```
Answer

```python
x = int(input())
if (x % 2 == 0):
    y = x + 1
else:
    y = x + 2
print(y)
```
Answer

\[ x = \text{int(input())} \]
\[ \text{if } (x \mod 2 == 0): \]
\[ y = x + 1 \]
\[ \text{print(y)} \]
\[ \text{if } (x \mod 2 != 0): \]
\[ y = x + 2 \]
\[ \text{print(y)} \]

Invariants shown in green:

- \( x \) is an integer value
- \( x \) is even
- \( x \) is even and \( y == x + 2 \) and \( y \) is even
- \( y > x \) and \( y \) is even

- \( x \) is odd
- \( x \) is odd and \( y == x + 1 \) and \( y \) is even
EXERCISE

x = int(input())
if (x % 2 == 0):  # x is even
    y = x + 2
else:
    y = x + 1

#Invariant: y > x and y is even
print(y)
EXERCISE

x = int(input())
if (x % 2 == 0):
    # x is even
    y = x + 2
else:
    y = x - 1
#
print(y)

What are two invariants here about y?
• state the invariants
• write the assert statement
EXERCISE

x = int(input())
y = int(input())
if y > x:
    z = y
else:
    z = x
print(z)

What is an invariant for z?
Write the assert statement
Asserting invariants

• Adding the statement `assert E` at a point in the code indicates that we expect an invariant $E$ to hold there.

• If $E$ is ever False at that point, we find out right away:
  – catches bugs early
  – makes it easier to locate the problem
Example

# give_raise(name, dept, amount, employee_db): update the database
# employee_db to give the employee specified, from the department specified,
# a raise of the amount specified

def give_raise(name, dept, amount, employee_db):
    assert dept in employee_db.keys()\
    and name in employee_db[dept].keys() \
    and amount > 0
    employee_db[dept][name][salary] += amount
Example

# give_raise(name, dept, amount, employee_db): update the database
# employee_db to give the employee specified, from the department specified,
# a raise of the amount specified

def give_raise(name, dept, amount, employee_db):

  assert dept in employee_db.keys(), "Bad department name: " + dept
  assert name in employee_db[dept].keys(), "Bad employee name: " + name
  assert amount > 0, "Bad raise amount: " + str(amount)

  employee_db[dept][name][salary] += amount
loop invariants
Figuring out invariants: loops

- A *loop invariant* is an invariant that is true at the beginning of each iteration of the loop.

```plaintext
while exp:
  stmt

for exp:
  stmt
```

loop invariant
Loop invariants

• A loop repeatedly executes a piece of code in order to achieve some goal
  – at the very beginning, none of that goal has been achieved
  – each iteration of the loop represents one step of progress towards that goal
  – at the end of the loop, the entirety of the goal has been achieved

• A loop invariant is a precise statement of how much progress has been made up to the beginning of the $i^{th}$ iteration
Example 1

def foo(arglist):
    i = 0
    while i < len(arglist):
        arglist[i] = i
        i = i + 1

    return arglist
Example 1

def foo(arglist):
    i = 0
    while i < len(arglist):
        arglist[i] = i
        i = i + 1
    return arglist

• First, understand the code
  – what will the contents of arglist be when the loop is complete?
  – write out the elements of arglist
    arglist[0] == 0
    arglist[1] == 1
    ...
    arglist[i] == i
Example 1

def foo(arglist):
    i = 0
    while i < len(arglist):
        arglist[i] = i
        i = i + 1
    return arglist

• Consider what happens on iteration $i$ ($i$ is arbitrary):

  - the $i^{th}$ element of arglist is set to the value $i$
  - $i$ is incremented
    $\Rightarrow$ index of the next element of arglist
def foo(arglist):
    i = 0
    while i < len(arglist):
        arglist[i] = i
        i = i + 1
    return arglist

• Consider what happens on iteration $i$ ($i$ is arbitrary)

the loop body computes one step of progress in the loop's computation
Example 1

def foo(arglist):
    i = 0
    while i < len(arglist):
        arglist[i] = i
        i = i + 1
    return arglist

Loop invariant

= what must be true at the beginning of each iteration

= what must be true at the beginning of iteration $i$

= what must be true of the accumulated effect of the first $i-1$ iterations
Example 1

def foo(arglist):
    i = 0
    while i < len(arglist):
        arglist[i] = i
        i = i + 1
    return arglist

Loop invariant
= what must be true of the accumulated effect of the first $i-1$ iterations
= for each iteration $j$ before iteration $i$, arglist[$j$] is set to $j$

= for each $j$, $0 \leq j < i : \text{arglist}[j] == j$
Example 1

def foo(arglist):
    i = 0
    while i < len(arglist)
        arglist[i] = i
        i = i + 1
    return arglist

for each $j$, $0 \leq j < i$ : $\text{arglist}[j] == j$

for each element $i$ of arglist, $\text{arglist}[i] == i$
Asserting invariants

def foo(arglist):
    i = 0
    while i < len(arglist):
        arglist[i] = i
        i = i + 1
    return arglist

assert fooInvariant(arglist, i)
assert fooInvariant(arglist, len(arglist))
Asserting invariants

def foo(arglist):
    i = 0
    while i < len(arglist):
        arglist[i] = i
        i = i + 1
    return arglist

assert foo_invariant(arglist, i)

def foo_invariant(arglist, i):
    j = 0
    while j < i:
        if arglist[j] != j:
            return False
        j += 1
    return True

assert foo_invariant(arglist, len(arglist))
Example 2

def foo(arglist):
    x = arglist[0]
    for i in range(len(arglist)):
        if x < arglist[i]:
            x = arglist[i]
    return x

• First, understand the code
  – what should x be after the loop completes?
• x is the max of the list
Example 2

def foo(arglist):
    x = arglist[0]
    for i in range(len(arglist)):
        if x < arglist[i]:
            x = arglist[i]
    return x

the loop body computes one step of progress in the loop's computation

invariant for iteration i: $x \geq \text{arglist}[i]$
def foo(arglist):
    x = arglist[0]
    for i in range(len(arglist)):
        if x < arglist[i]:
            x = arglist[i]
    return x

loop invariant:
  i == 0 and x == arglist[0]

Or
  i > 0 and x is the max of the list elements from arglist[0] up to arglist[i-1]
Example 2

def foo(arglist):
    x = arglist[0]
    for i in range(len(arglist)):
        if x < arglist[i]:
            x = arglist[i]
    return x

invariant:
x is the max of all the elements of arglist
Exercise-1

def foo(arglist):
    x = arglist[0]
    for i in range(len(arglist)):
        if x < arglist[i]:
            x = arglist[i]
    return x

def foo_invariant(arglist, i, x):
    # Write the code to satisfy the loop invariant

loop invariant:
    $i = 0$ and $x = \text{arglist}[0]$

or

$(i > 0 \text{ and } x \text{ is the max of the list elements from arglist}[0] \text{ up to arglist}[i-1])$
Figuring out loop invariants: summary

• Figure out the effect of an (arbitrary) iteration of the loop body
• From this, figure out what must be true after \( k \) iterations of the loop
  – the accumulated effect of iterations 0, ..., \( k-1 \)
• If there are nested loops: work from the innermost loop(s) outward
def foo(x):  # x is a list
    y = []
    i = len(x) - 1
    while i >= 0:
        y.append(x[i])  # attach x[i] to the end of y
        i -= 1
    return y

Loop invariant = ???
def foo(x):  # x is a list
    y = []
    i = len(x) - 1
    while i >= 0:
        y.append(x[i])
        i -= 1
    return y

k == 0 and y == []
or
k > 0 and y[0] == x[-1] and
    y[1] == x[-2] and
    y[2] == x[-3] and
    y[3] == x[-4] and
    ...
    y[j] == x[-(j+1)]
for 0 <= j < k
def foo(x):  # x is a list
    y = []
    i = len(x) - 1
    while i >= 0:
        y.append(x[i])
        i -= 1
    return y

def foo_invariant(x, y, k):
    # Write the code to satisfy the
    # loop invariant
    # do we really need k?
    # will something else give us that information?

    loop invariant:
    k == 0 and y == []
    or
    (k > 0 and y[j] == x[-(j+1)] for all j < k)
def foo(x):  # x is a list
    y = []
    i = len(x) - 1
    while i >= 0:
        assert foo_invariant(x, y)
        y.append(x[i])
        i -= 1
    return y

def foo_invariant(x, y):
    j = 0
    while j < len(y):
        if y[j] != x[-(j+1)]:
            return False
        j += 1
    return True
pre- and post-conditions
Preconditions

```python
def average(x):
    sum = 0
    for i in range(len(x)):
        sum += x[i]
    avg = sum/len(x)
    return avg
```
 Preconditions

```python
>>> def average(x):
    sum = 0
    for i in range(len(x)):
        sum += x[i]
    avg = sum/len(x)
    return avg

>>> average([1,2,3,4])
2.5
```
Preconditions

```python
>>> average([])
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    average([])
  File "<pyshell#19>", line 5, in average
    avg = sum/len(x)
ZeroDivisionError: division by zero
```
Preconditions

>>> average([ ])
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    average([])
  File "<pyshell#19>", line 5, in average
    avg = sum/len(x)
ZeroDivisionError: division by zero

In order to work correctly, \texttt{average(x)} \textit{requires} \texttt{len(x) > 0}

- this requirement is called a \textit{precondition} for this function
  - preconditions should be documented in comments
  - they can be asserted in the code
Documenting preconditions: Example

“"" average(x) : returns the average of the numbers in a list
parameters: x is a list of numbers
precondition: x must be non-empty"""

def average(x):
    assert len(x) > 0
    sum = 0
    for i in range(len(x)):
        sum += x[i]
    avg = sum/len(x)
    return avg
Postconditions

• A *postcondition* for a piece of code $C$ is a condition that must be true immediately after the execution of $C$
  – assumes $C$'s precondition has been met

Example:

```python
def abs(x):
    if x < 0:
        x = -x
    return x
```

precondition: $x$ is a number
postcondition: $|x| \geq 0$
Figuring out invariants: function calls

\[ y = \text{somefunc}(\text{arg}_1, \ldots, \text{arg}_n) \]

- figure out the invariant just before the call to somefunc()

- the value of \( y \), and the invariant after somefunc() returns, is obtained using somefunc()'s postcondition
Using invariants

• Given a piece of code:
  – examine it to figure out the invariants
  – compare it with what we think it's supposed to do

• Given a program specification:
  – figure out the invariant(s) that should hold
  – check the code to see whether these invariants are met
    o insert asserts at appropriate points
Invariants: Summary

• An invariant at a program point states what must be true about the program's state when control reaches that point

• Particular kinds of invariants: loop invariants, preconditions, postconditions

• Uses:
  – check whether a piece of code does what it's supposed to do
  – early detection of problems (via `assert` statements)
  – documentation