CSc 120
Introduction to Computer Programming II

05: Intro to 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.
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.
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.
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 $\iff$ the code has a bug.
Invariants (informal definition)

An *invariant* is something you might put in a comment, which explains the current state of the program.

(preview):
An *assert* is an invariant written as code – which is checked by the program at runtime.
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, list) -- returns the # position where the given string # occurs in the given list.

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

**Use of lookup()**

```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, list):
    for i in range(len(list)):
        if string == list[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?
Example

```python
def lookup(string, list):
    for i in range(len(list)):
        if string == list[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?

- **True**
  - this is an invariant, but not an interesting one since it does not tell us anything about the program

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

- **x[pos] == y**
  - this is not an invariant (why?)

- ???
Example

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

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

Ideally, we want something like:

```python
if y in x then x[pos] == y
else pos == some_special_value
```

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

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

Q: What invariant(s) hold here?
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, list):
    for i in range(len(list)):
        if string == list[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'
lookup(y, x) returns None
the invariant says it should return 1

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

⇒ leads us to examine the code for returning with None
## Example

<table>
<thead>
<tr>
<th>Buggy code</th>
<th>Fixed code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>def lookup(string, list):</strong></td>
<td><strong>def lookup(string, list):</strong></td>
</tr>
<tr>
<td>for i in range(len(list)):</td>
<td>for i in range(len(list)):</td>
</tr>
<tr>
<td>if string == list[i]:</td>
<td>if string == list[i]:</td>
</tr>
<tr>
<td>return i</td>
<td>return i</td>
</tr>
<tr>
<td>return None</td>
<td>return None</td>
</tr>
</tbody>
</table>
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 this 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 the each function/method and work our way down its statements
  — if nothing is known, the invariant is True
Figuring out invariants: assignments

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

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

if $exp_1$
\>
\>
$P \land \neg exp_1$
\>
\>
$stmt_1$

elif $exp_2$
\>
\>
$P \land \neg exp_1$
\>
\>
\>
\>
$and \ exp_2$
\>
\>
$stmt_2$

elif $exp_3$
\>
\>
$P \land \neg exp_1$
\>
\>
\>
\>
$and \ \neg exp_2$
\>
\>
\>
\>
$and \ exp_3$
\>
\>
$stmt_3$

....

invariants shown in green
Figuring out invariants: conditionals

**Special case:**

```python
if exp:
    stmt_1
else:
    stmt_2
```

**Invariants shown in green:**

- $P$ and $exp_1$
- $P$ and $not exp_1$ and $exp_2$
- $P$ and $not exp_1$ and $not exp_2$ and $exp_3$

```python
if exp_1:
    stmt_1
elif exp_2:
    stmt_2
elif exp_3:
    stmt_3
....
```
Figuring out invariants: conditionals

\[
\text{if } exp : \\
\begin{align*}
\text{exp} &= \text{True} \\
\text{exp} &= \text{False}
\end{align*}
\]

\[
\begin{align*}
\text{stmt}_1 &\quad \text{and} \quad \text{exp} \\
\text{stmt}_2 &\quad \text{and} \quad \text{not exp} \\
\text{stmt}_3 &\quad \text{P and whatever is common in both } P_1 \text{ and } P_2
\end{align*}
\]

\text{(NOTE: This is not the same as “} P_1 \text{ and } P_2 \text{”)}

invariants shown in green
Example 1

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

25
Example 1

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

Example 1 diagram:

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

- **x** is an integer value
- **x** is even
- **y** = **x** + 2
- **y** is even
- **y** > **x** and **y** is even
- **invariants shown in green**
EXERCISE

```
x = int(input())
if (x % 2 == 0):  # x is even
    y = x + 2
else:
    y = x - 1
#print(y)
```
x = int(input())
if (x < 0):
    x = -x
#
print(x)
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
# 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 keys(employee_db) \
        and name in keys(employee_db[dept]) \
        and amount > 0
    employee_db[dept][name][salary] += amount
def give_raise(name, dept, amount, employee_db):

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

    employee_db[dept][name][salary] += amount