loop invariants
Figuring out invariants: loops

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

• 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
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 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 $j$, $0 \leq j < i$ : arglist[$j$] == $j$
Example 1

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

for each element $i$ of arglist, arglist[$i$] == $i$

for each $j$, $0 \leq j < i$ : arglist[$j$] == $j$
def foo(arglist):
    i = 0
    while i < len(arglist):
        arglist[i] = i
        i = i + 1
    return arglist

assert foo_invariant(arglist, i):
    j = 0
    while j < i:
        if arglist[j] != j:
            return False
    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
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: $x \geq \text{list}[i]$
Example 2

def foo(arglist):
    x = arglist[0]
    for i in range(len(arglist)):
        if x < arglist[i]:
            x = arglist[i]
    return x
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
Example 3

def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?
    return x
Example 3

def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?
    return x
Example 3

```python
def foo(x):
    # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?
    return x
```

x[i] = smaller of x[i], x[j]
x[j] = bigger of x[i], x[j]
Example 3

def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

# what can we say about x here?
return x
def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

# what can we say about x here?
return x
def foo(x):  
    # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?
    return x

after iteration 1:
    x[i] = min(x[i], x[i+1])

after iteration 2:
    x[i] = min(x[i], x[i+1], x[i+2])
Example 3

def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

# what can we say about x here?
return x

after iteration 1:
    x[i] = min(x[i], x[i+1])

after iteration 2:
    x[i] = min(x[i], x[i+1], x[i+2])

after iteration 3:
    x[i] = min(x[i], x[i+1], x[i+2], x[i+3])
Example 3

```python
def foo(x):
    # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?
    return x
```

after iteration 1:

\[ x[i] = \text{min}(x[i], x[i+1]) \]

after iteration 2:

\[ x[i] = \text{min}(x[i], x[i+1], x[i+2]) \]

after iteration 3:

\[ x[i] = \text{min}(x[i], x[i+1], x[i+2], x[i+3]) \]

... after iteration \( k \) (\( j = i+k \)):

\[ x[i] = \text{min}(x[i] \ldots x[i+k]) \]
Example 3

def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?
    return x

Loop invariant (inner loop):
after iteration $k$:
$\text{x}[i] = \text{min}(\text{x}[i] \ldots \text{x}[i+k])$
Example 3

def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?
    return x

A more "Pythonic" version:
    x[i] = min(x[i] ... x[i+k])

Loop invariant (inner loop):
    after iteration k:
        x[i] = min(x[i] ... x[i+k])
Example 3

def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

# what can we say about x here?
return x

after the inner loop finishes:
x[i] = min(x[i] ... x[len(x)-1])
Example 3

```python
def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

# what can we say about x here?
return x
```

after the inner loop finishes:

```plaintext
x[i] = min(x[i] ... x[len(x)-1])
```

Better:

```plaintext
x[i] = min(x[i:])
```
def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?
    return x

after iteration 1:
x[0] = min(x[0:])
    = smallest value in x
Example 3

def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?
    return x

after iteration 1:
x[0] = min(x[0:])
    = smallest value in x

after iteration 2 (i = 1):
x[0] = smallest value in x
x[1] = second-smallest value in x
Example 3

def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?
return x

after iteration 1:
x[0] = min(x[0:])
    = smallest value in x

after iteration 2 (i = 1):
x[0] = smallest value in x
x[1] = second-smallest value in x

after iteration 3 (i = 2):
x[0] = smallest value in x
x[1] = second-smallest value in x
x[2] = third-smallest value in x
Example 3

def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?

return x

Loop invariant (outer loop):
    after \( k \) iterations, the \( k \) smallest elements of \( x \) are in sorted (ascending) order
Example 3

def foo(x):    # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]

    # what can we say about x here?
    return x

Loop invariant (outer loop):
    after $k$ iterations, the first $k$ elements of $x$ are in sorted (ascending) order

the outer loop iterates $\text{len}(x)$ times
⇒ the smallest $\text{len}(x)$ elements of $x$ are in sorted (ascending) order
Example 3

def foo(x):  # x a list of numbers
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if x[i] > x[j]:
                x[i], x[j] = x[j], x[i]
    # what can we say about x here?
    return x

Loop invariant (outer loop):
    after $k$ iterations, the first $k$ elements of $x$ are in sorted (ascending) order

the outer loop iterates $\text{len}(x)$ times
$\Rightarrow$ the smallest $\text{len}(x)$ elements of $x$ are in sorted (ascending) order

$\Rightarrow$ $x$ is sorted in ascending order
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 = ???

what can we say about y here?
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
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.

>>> 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
>>> 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
>>> average([])
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    average([])
  File "<pyshell#6>", line 5, in average
    avg = sum/len(x)
ZeroDivisionError: division by zero
```
Preconditions

In order to work correctly, `average(x)` **requires** \(\text{len}(x) > 0\)

- this requirement is called a *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 the list x
# 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: $\text{abs}(x) \geq 0$
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
    ○ 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.