CSc 120
Introduction to Computer Programming II

Adapted from slides by
Dr. Saumya Debray

03: 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

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

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

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

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, list) -- returns the
# position where the given string
# occurs in the given list.

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

```
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

```
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?

- 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, 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?

Ideally, we want something like:

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

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

Buggy code

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

        return None
```

Fixed code

```
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 ⇔ 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 the each function/method and work our way down its statements
  - if nothing is known, the invariant is True

Figuring out invariants: assignments

invariants shown in green

$x_1, \ldots, x_n = e_1, \ldots, e_n$

- $x_1 == e_1$ and ... and $x_n == e_n$
- anything else: unchanged from before the assignment

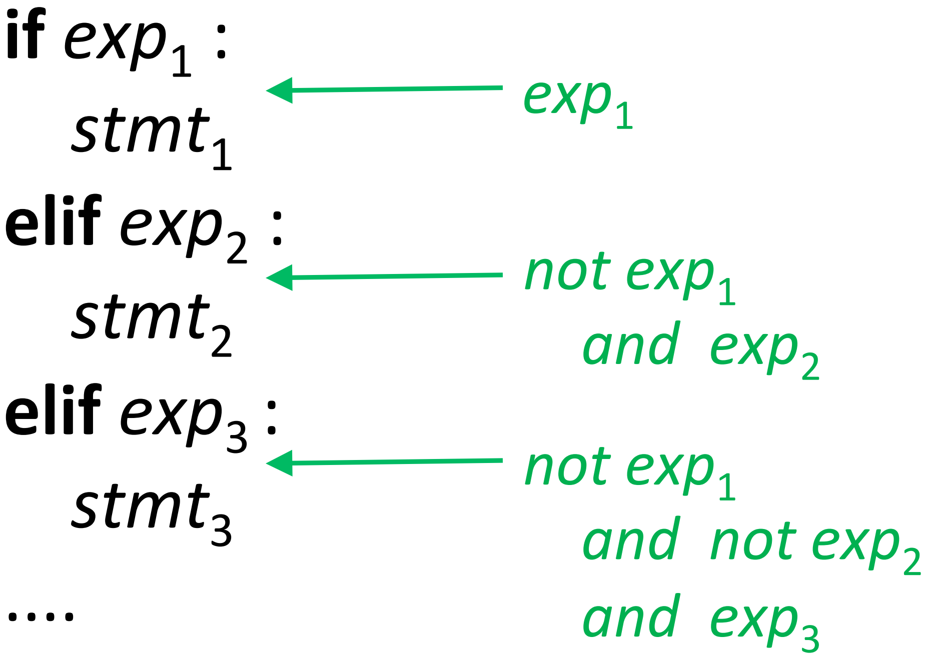
Figuring out invariants: conditionals

invariants shown in green

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

Figuring out invariants: conditionals

invariants shown in green

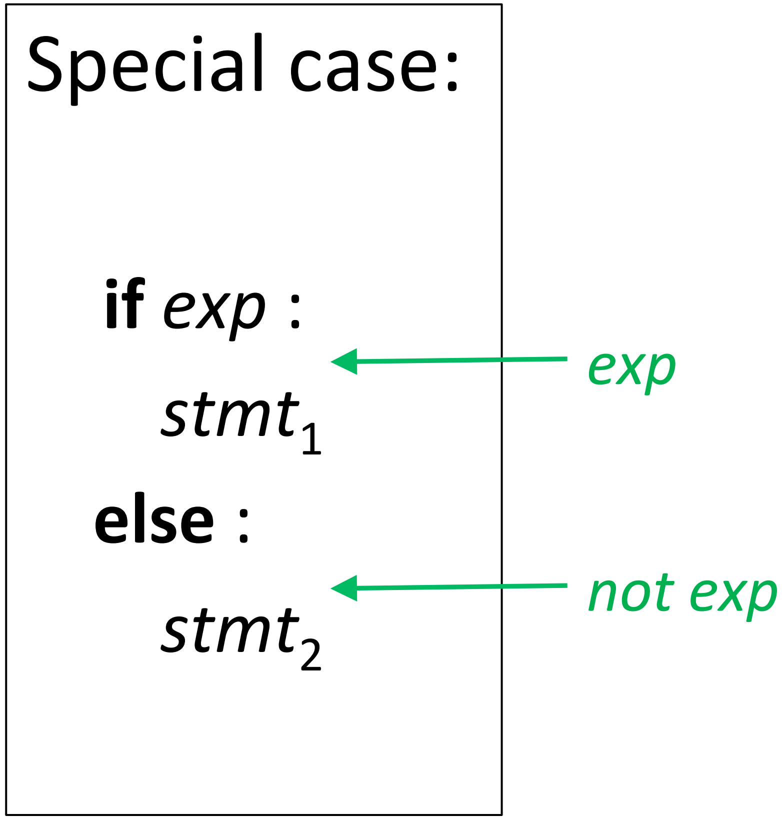


```
if $exp_1$ :
    $stmt_1$
elif $exp_2$ :
    $stmt_2$
elif $exp_3$ :
    $stmt_3$
....
```

exp_1

not exp_1
and exp_2

not exp_1
and not exp_2
and exp_3



Special case:

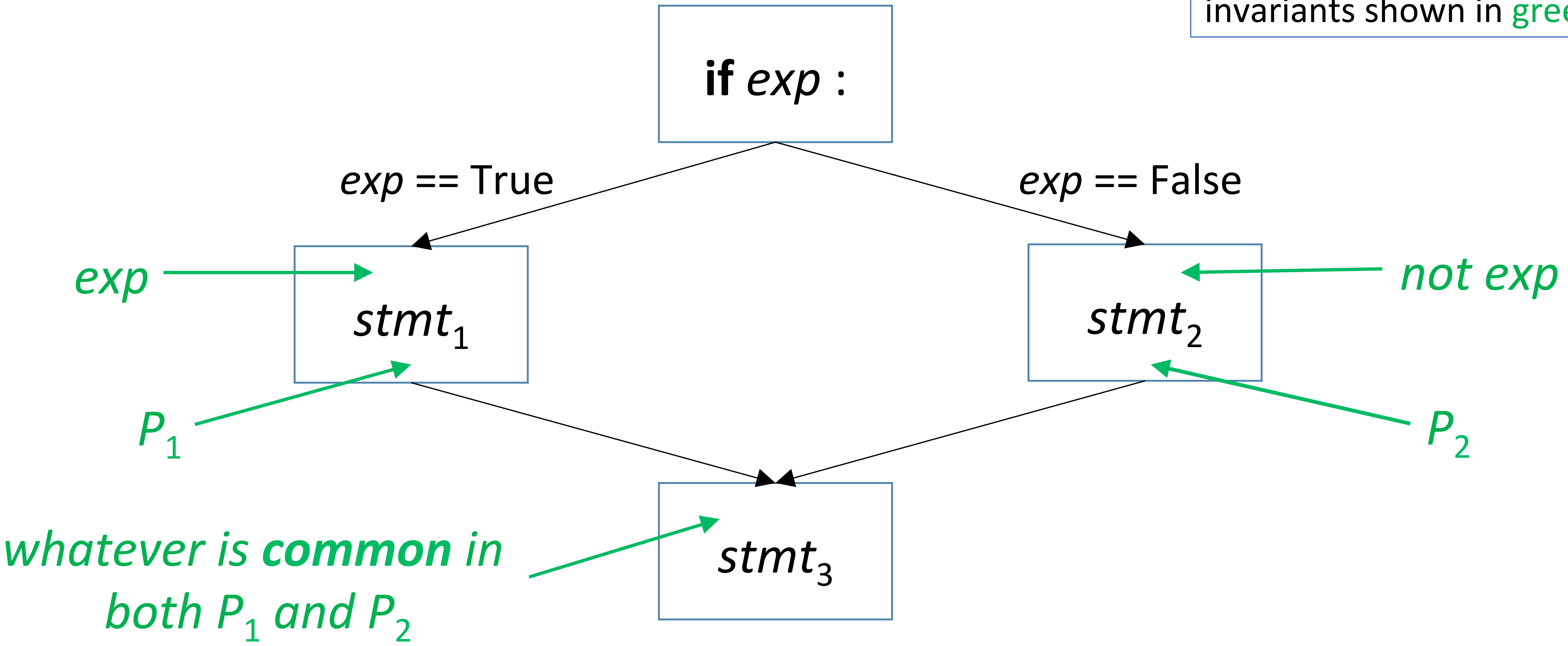
```
if $exp$ :
    $stmt_1$
else :
    $stmt_2$
```

exp

not exp

Figuring out invariants: conditionals

invariants shown in green



if *exp* :

exp == True

exp == False

exp

not exp

stmt$_1$

stmt$_2$

P_1

P_2

stmt$_3$

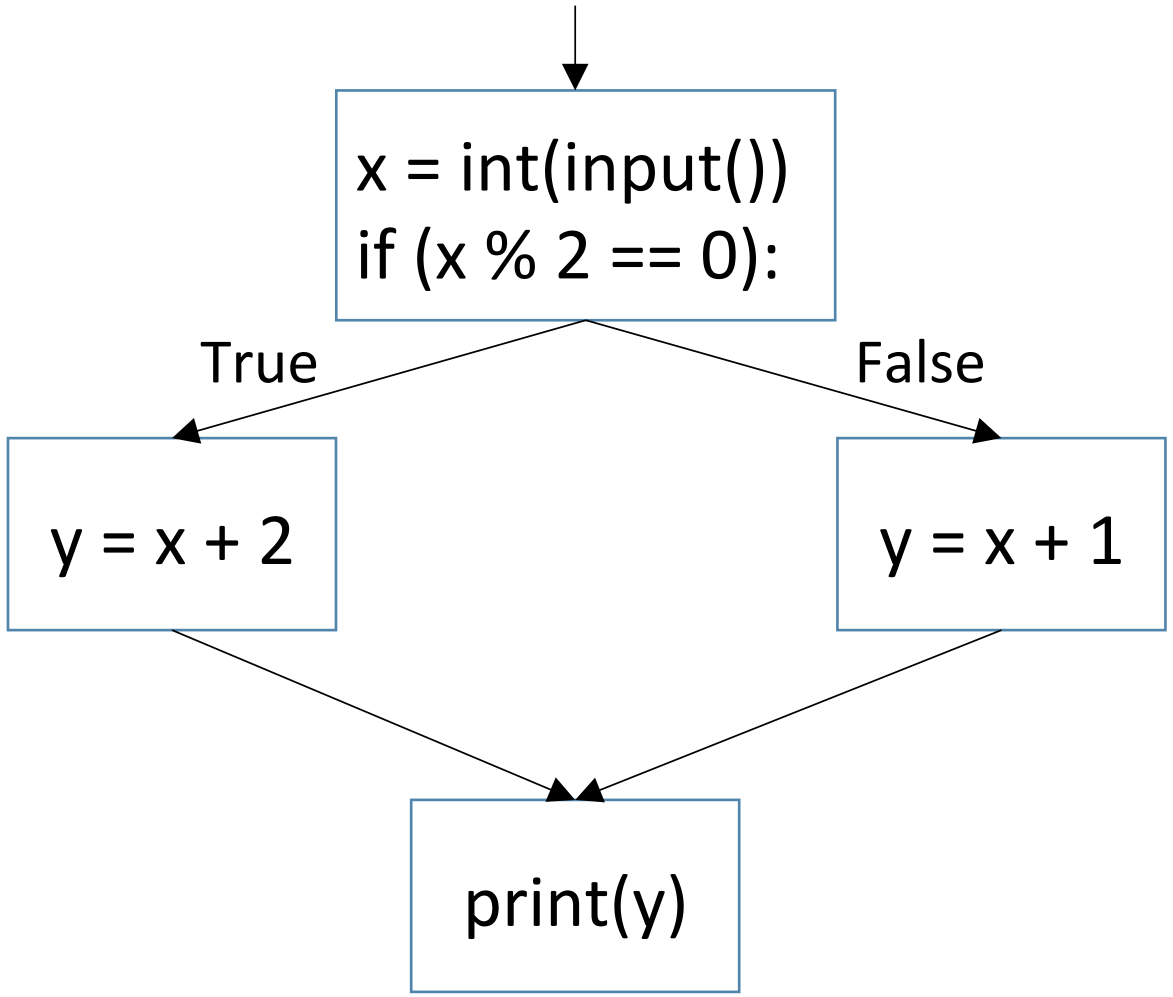
whatever is ***common*** *in both* P_1 *and* P_2

(NOTE: This is not the same as “P_1 *and* P_2”)

Example 1

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

Example 1



x = int(input())
if (x % 2 == 0):

True

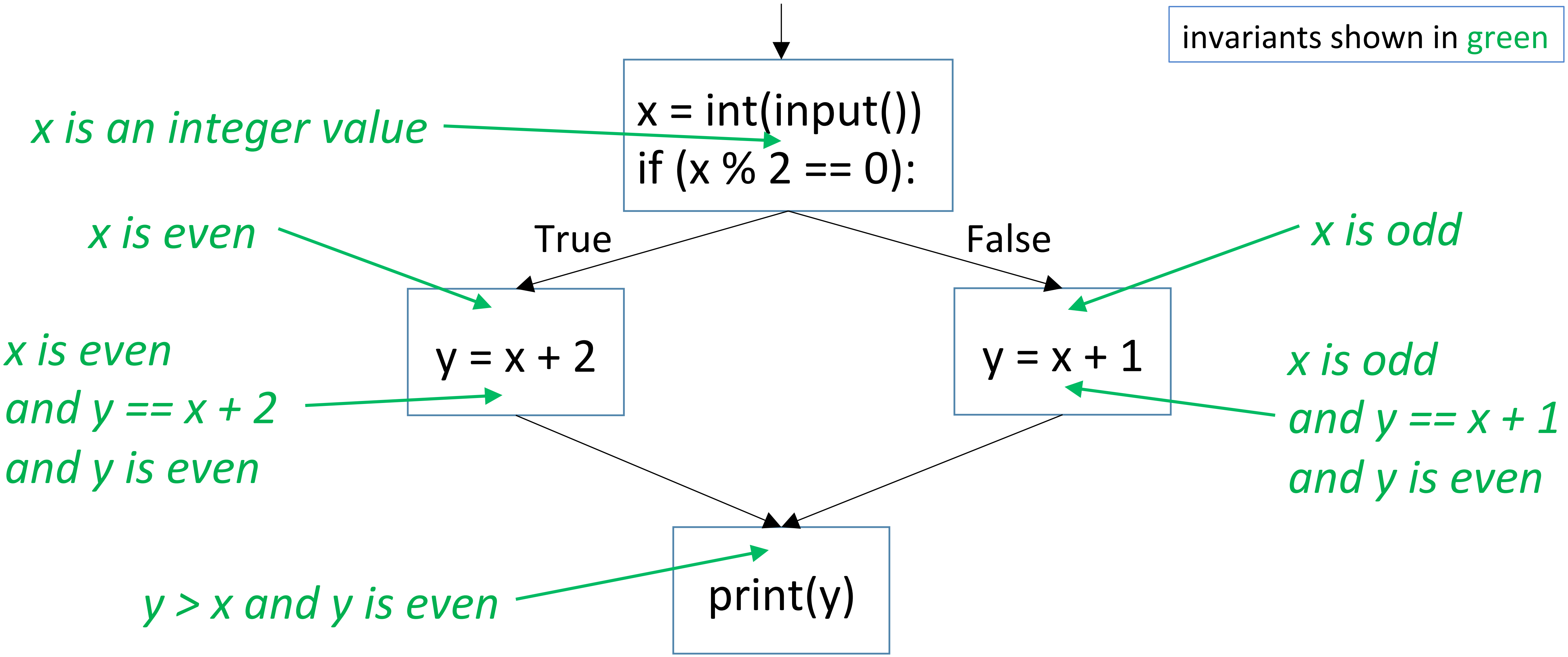
False

y = x + 2

y = x + 1

print(y)

Example 1



invariants shown in green

x = int(input())
if (x % 2 == 0):

x is an integer value

True

False

x is even

y = x + 2

x is even
and y == x + 2
and y is even

x is odd

y = x + 1

x is odd
and y == x + 1
and y is even

print(y)

y > x and y is even

EXERCISE

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

print(y)
```

???

EXERCISE

```
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

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 keys(employee_db) \
            and name in keys(employee_db[dept]) \
            and amount > 0
    employee_db[dept][name][salary] += amount
```

Example

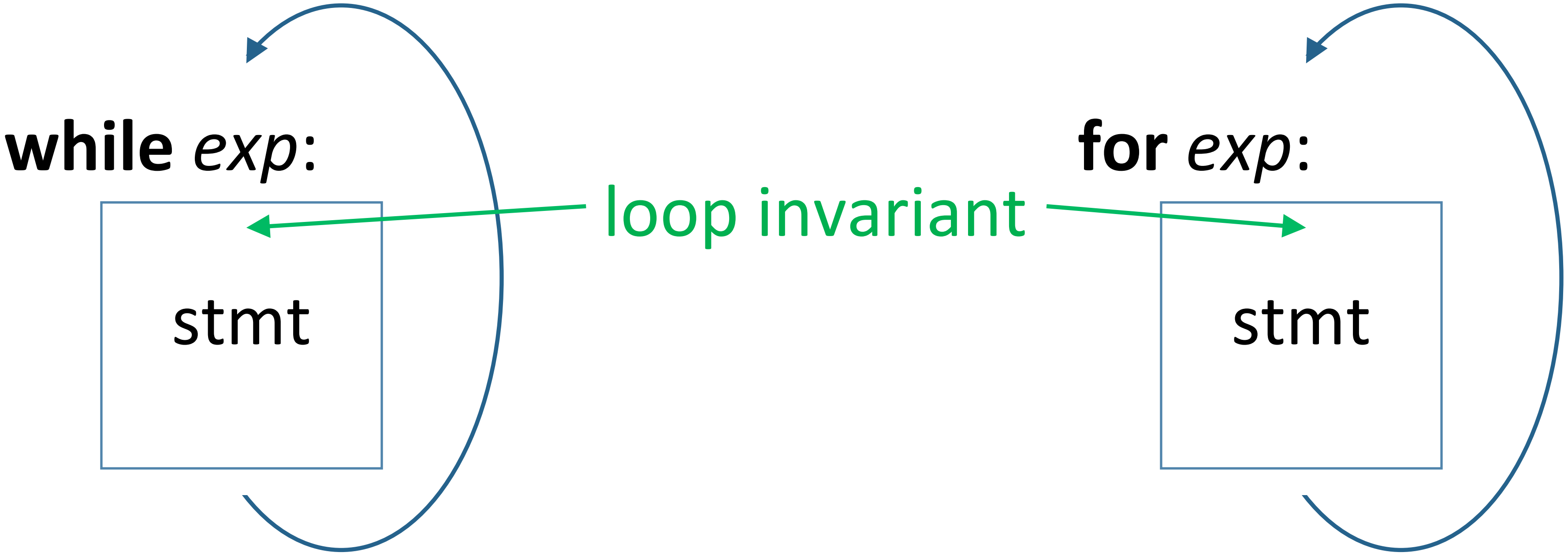
```
# give_raise(name, dept, amount, employee_db): update the database
# employee_db to give the employee specified, from the department specified,
# a raise of the amount specified

def give_raise(name, dept, amount, employee_db):
    assert  dept in 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
```

More informative runtime error messages

loop invariants

Figuring out invariants: loops

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



while *exp*:

stmt

loop invariant

for *exp*:

stmt

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
    - ⟹ 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 j, $0 \leq j < i$: arglist[j] == j

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

Asserting invariants

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

    return arglist
```

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

    return True
```

assert foo_invariant(arglist, i)

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: $x \geq arglist[i]$

Example 2

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

    return x
```

loop invariant:

x is the max of the list elements from arglist[0] up to arglist[i]

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

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 (will look at this later)

EXERCISE

```
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

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

Preconditions

```
>>> 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

```
>>> 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, average(x) ***requires*** 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:

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

precondition: x is a number

postcondition: abs(x) ≥ 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 **assert**s 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