Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Specifications and Implementations

We describe the specification of various kinds of programming-language constructs, and how their implementations contribute to a program that meets its requirements:

- *Statements*, which define effects.
- *Declarations*, which create program variables.
- *Methods*, which group *statements* and *declarations* into meaningful operations.
- *Classes*, which aggregate *methods* and *declarations* into coherent modules.

Programs serve a purpose. They satisfy a requirement.

Statement Specifications

Programs serve a purpose. They satisfy a requirement.

Some requirements are small: Square a number.

Some requirements are large: Control a rocket to the moon.

Regardless, our goal is to write a program that satisfies a requirement.

A specification, written as comment, is a precise articulation of a requirement.

```
#.Specification.
```

A specification, written as comment, is a precise articulation of a requirement.

```
#.Specification.
```

The dot (`.`) after the hash mark (#) signifies that the specification has not yet been implemented.

An implementation, aligned below it, says how to meet the requirement.

```
# Specification.
Implementation
```

Statement Specifications

Implementations

Write specifications as imperatives that say what must be accomplished.

```
# Specification.
Implementation
```

Write specifications as imperatives that say what must be accomplished.

```
# Specification.
Implementation
```

Sample program-segment derivations starting with the *specification-implementation pattern*.

Statement Specifications

Implementations

Write specifications as imperatives that say what must be accomplished.

```
# Specification.
Implementation
```

```
# Specification.
Implementation
```

Write specifications as imperatives that say what must be accomplished.

```
# Specification.
Implementation
```

```
# Output the square of an integer that is provided as input.
Implementation
```

Write implementations that say how to do so.

```
# Specification.
Implementation
```

```
# Output the square of an integer that is provided as input.
n = int(input())
print(n * n)
```

A given specification can be implemented in multiple ways.

```
# Specification.
Implementation
```

WAY 1

```
# Output the square of an integer that is provided as input.
n = int(input())
print(n * n)
```

A given specification can be implemented in multiple ways.

```
# Specification.
Implementation
```

WAY 2

```
# Output the square of an integer that is provided as input.
# ---------------------------------------------------------
n = int(input())
#.Let s be the square of n.
print(s)
```

Statement Specifications

Implementations

When the implementation of one specification contains another specification, the first should be followed by a line of dashes to indicate that the first specification is a level above its implementation.

A given specification can be implemented in multiple ways.

```
# Specification.
Implementation
```

WAY 2

```
# Output the square of an integer that is provided as input.
# ------------------------------------------------------------
n = int(input())
#.Let s be the square of n.
print(s)
```

When the specification is not yet implemented, it can stand as a single line within other lines, with no vertical whitespace around it.

A given specification can be implemented in multiple ways.

```
# Specification.
Implementation
```

WAY 2

```
# Output the square of an integer that is provided as input.
# ------------------------------------------------------------
n = int(input())
#.Let s be the square of n.
print(s)
```

Implementations

Statement Specifications

A given specification can be implemented in multiple ways.

```
# Specification.
Implementation
```

WAY 2a

```
# Output the square of an integer that is provided as input.
# ------------------------------------------------------------
n = int(input())

# Let s be the square of n.
s = n * n

print(s)
```

Statement Specifications

Implementations

When the specification is not yet implemented, it can stand as a single line within other lines, with no vertical whitespace around it. But once it has been implemented, it (together with its implementation) should be set off from the rest with blank lines.

A given specification can be implemented in multiple ways.

```
# Specification.
Implementation
```

WAY 2a

```
# Output the square of an integer that is provided as input.
# ---------------------------------------------------------
n = int(input())

# Let s be the square of n.
s = n * n

print(s)
```

Statement Specifications

Implementations

A given specification can be implemented in multiple ways.

```
# Specification.
Implementation
```

WAY 2b

```
# Output the square of an integer that is provided as input.
# ----------------------------------------------------------
n = int(input())

# Let s be the square of n.
m = abs(n)
s = 0
for k in range(0, m):
    s = s + m

print(s)
```

Write specifications as imperatives.

```
# Output the square of an integer that is provided as input.
n = int(input()); print(n * n)
```

Avoid meandering descriptions.

Be succinct. Eliminate needless words.

```
# Output the square of an integer that is provided as input.
n = int(input()); print(n * n)
```

☞ **Repeatedly improve comments by relentless copy editing.**

Statement Specifications

Specification Style

By convention, state input before output.

```
# Input an integer, and output the square of that integer.
n = int(input()); print(n * n)
```

Use pronouns.

```
# Input an integer, and output its square.
n = int(input()); print(n * n)
```

Use letters as pronouns.

```
# Input integer k, and output k squared.
n = int(input()); print(n * n)
```

Statement Specifications

Specification Style

Use letters as pronouns.

```
# Input integer j, and output j squared.
n = int(input()); print(n * n)
```

The scope of such a pronoun is local to the specification.

Statement Specifications

Specification Style

Use letters as pronouns, or as the names of variables.

```
# Input integer n, and output n squared.
n = int(input()); print(n * n)
```

Use letters as pronouns, or as the names of variables.

```
# Input integer n, and output n squared.
print(math.pow(int(input()), 2))
```

But in this implementation there is no variable n, so n must be a pronoun.

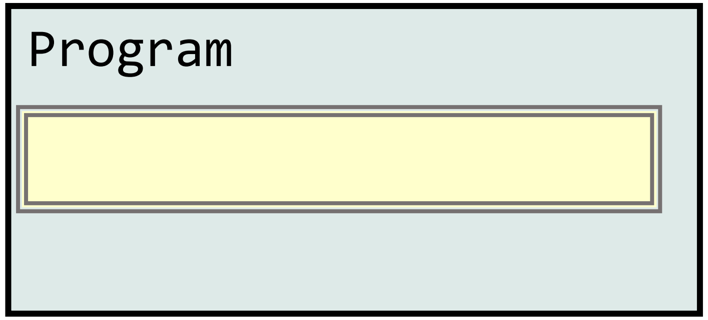
Use programming-language *expressions* in specifications, if you wish.

```
# Input integer n, and output n*n.
n = int(input()); print(n * n)
```

But an *expression* in a specification isn't necessarily computed.

```
# Input integer x, and output the number n such that n*n=x.
print(math.sqrt(int(input()))
```

Suppose, in a program, you need to exchange the values of variables x and y.



Write the specification as if in a higher-level programming language.

```
#.Swap x and y.
```

☞ **Write comments as an integral part of the coding process, not as afterthoughts.**

Defer implementation so you don't get distracted. Move on to other matters.

```
#.Swap x and y.
```

other matters

☞ **Write comments as an integral part of the coding process, not as afterthoughts.**

Or implement it now, if simple enough to not get distracted.

```
# Swap x and y.
temp = x
x = y
y = temp
```

other matters

☞ **Write comments as an integral part of the coding process, not as afterthoughts.**

Then ignore it in considering the specification's relationship to other matters.

```
# Swap x and y.
temp = x
x = y
y = temp
```

other matters

Let your eye skip over the implementation.

☞ **Write comments as an integral part of the coding process, not as afterthoughts.**

Then ignore it in considering the specification's relationship to other matters.

```
# Swap x and y.
...
```

other matters

Let your eye skip over the implementation as if it were elided.

☞ **Write comments as an integral part of the coding process, not as afterthoughts.**

An implementation can include another specification

```
# Swap x and y.
# -------------
#.Copy x to temp.
x = y
y = temp
```

other matters

which is then implemented.

```
# Swap x and y.
# -------------
# Copy x to temp.
temp = x

x = y

y = temp
```

other matters

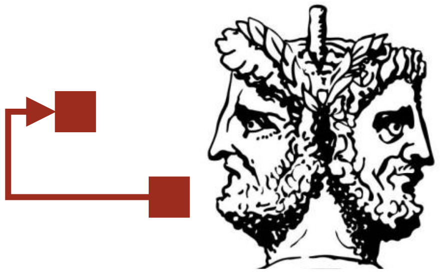
A specification faces two directions, like the Roman god Janus.

```
# Swap x and y.
# -------------
# Copy x to temp.
temp = x

x = y

y = temp
```

other matters



Outward, it is part of the implementation of an encompassing specification.

```
# Swap x and y.
# -------------

# Copy x to temp.
temp = x

x = y

y = temp
```

other matters

Inward, it is a specification that is being implemented.

```
# Swap x and y.
# -------------
# Copy x to temp.
temp = x

x = y

y = temp
```

other matters

Avoid redundant specifications that say the obvious.

```
# Copy x to temp.
temp = x
```

other matters

☞ **Omit specifications whose implementations are at least as brief and clear as the specification itself.**

Avoid redundant specifications that say the obvious.

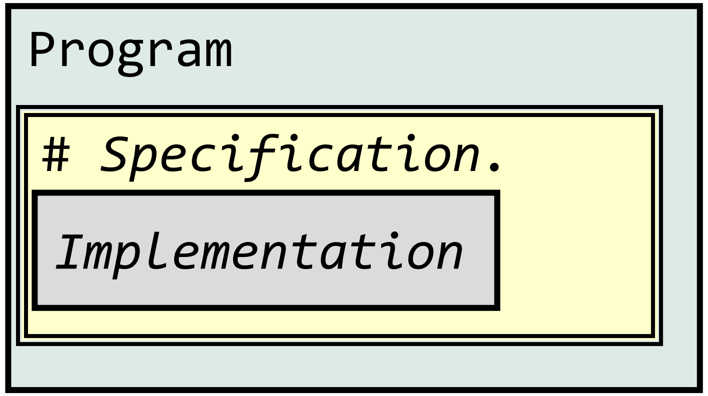
```
temp = x
```

other matters

☞ **Omit specifications whose implementations are at least as brief and clear as the specification itself.**

Statement Specifications
What? Not How?

A specification is a contract with the rest of the program that says *what* must be accomplished, not *how* to do so.

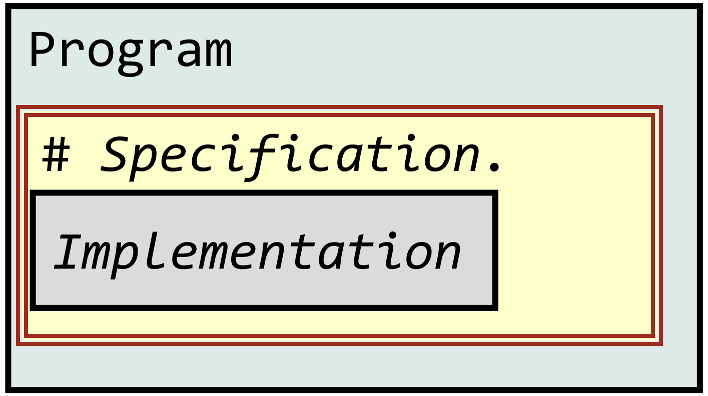


Proviso: As long as the program does this and that.
Promise: The specification (and its implementation) will do thus and such.

Statement Specifications

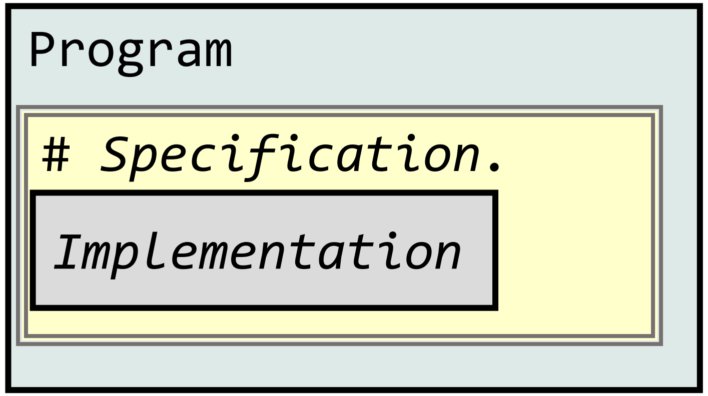
What? Not How?

A specification helps to control complexity.



The contract (double line) partitions code into the specification and its implementation (on the one hand), and the rest of the program (on the other).

A specification is both constraining and liberating.

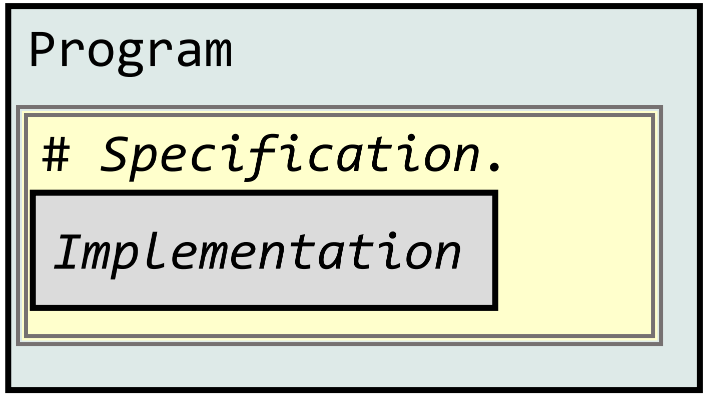


Constraining: (If the proviso is met) then it must do what is required.
Liberating: But its implementation is free to do so in any way it wants.

Statement Specifications

What? Not How?

A specification promotes pliability and comprehensibility.



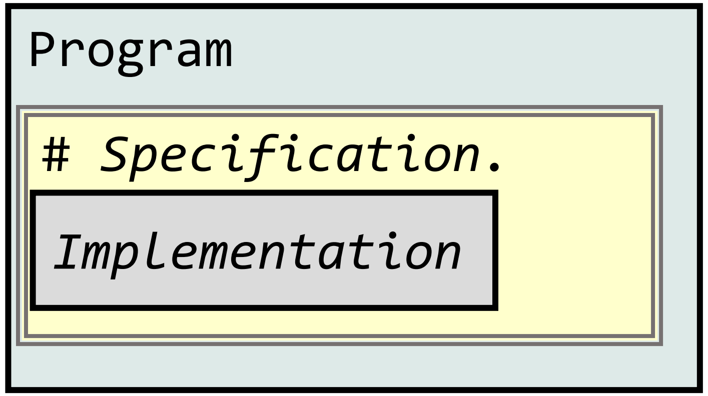
Pliability: The implementation can be changed without affecting the rest of the program.

Comprehensibility: The program can ignore implementation details not mentioned by the specification.

Statement Specifications

What? Not How?

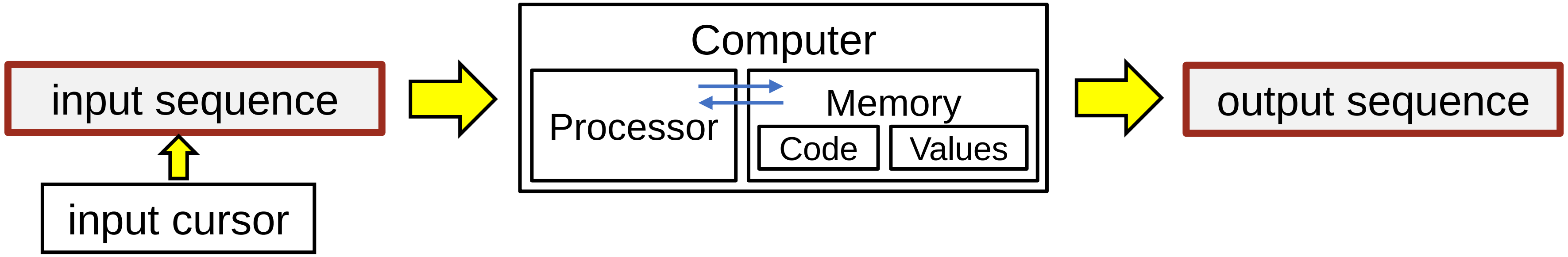
Specifications encapsulate details and hide information behind abstraction barriers.



These notions are central to object-oriented programming (discussed later, but already relevant at the level of statement specifications).

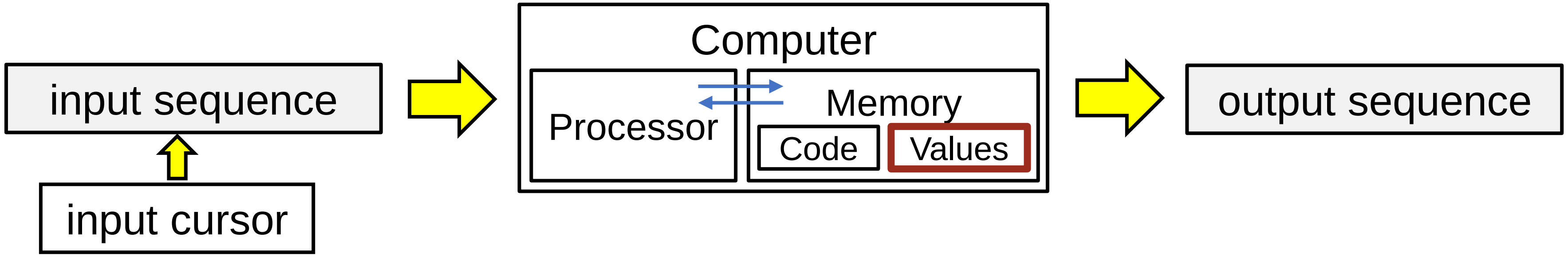
An Input/Output specification ("I/O spec") reads and writes external data

```
#.Input integer n, and output n squared.
```



Alternatively, an I/O spec sets values of some variables from values of other variables, leaving the external data unchanged.

```
#.Given integer variable n, let variable s be n squared.
```



Alternatively, an I/O spec sets values of some variables from values of other variables, leaving the external data unchanged.

```
#.Given integer variable n, let variable s be n squared.
```

	Before		After	
n	4	input variable	4	
s	9		16	output variable

In general, an I/O spec requires changing a before state into an after state.

```
#.Given before state, establish after state.
```

In general, an I/O spec requires changing a before state into an after state.

```
#.Given precondition, establish postcondition.
```

Before

described by precondition

After

described by postcondition

Use pronouns to distinguish the before and after values of a variable that is both input and output

```
#.Swap x and y.
```

Before			After		
x	4	input variable	x	9	output variable
y	9	input variable	y	4	output variable

Use pronouns to distinguish the before and after values of a variable that is both input and output

```
#.Given x=X and y=Y, establish x=Y and y=X.
```

Before

x 4 input variable

y 9 input variable

After

x 9 output variable

y 4 output variable

A specification says what must happen when the precondition holds

```
#.Given x≥0, let y be the square root of x.
```

Before			After		
x	4	input variable	x	4	
y	9		y	2	output variable

But says nothing about what may happen otherwise.

```
#.Given x≥0, let y be the square root of x.
```

	Before			After	
x	-4	input variable	x	-4	
y	9		y	9	output variable

But says nothing about what may happen otherwise.

```
#.Given x≥0, let y be the square root of x.
```

Before

x | -4 | input variable

y | 9 |

After

Reaching a specification whose precondition doesn't hold is indicative of an error, e.g., we expect x to be nonnegative, so it was incorrectly computed.

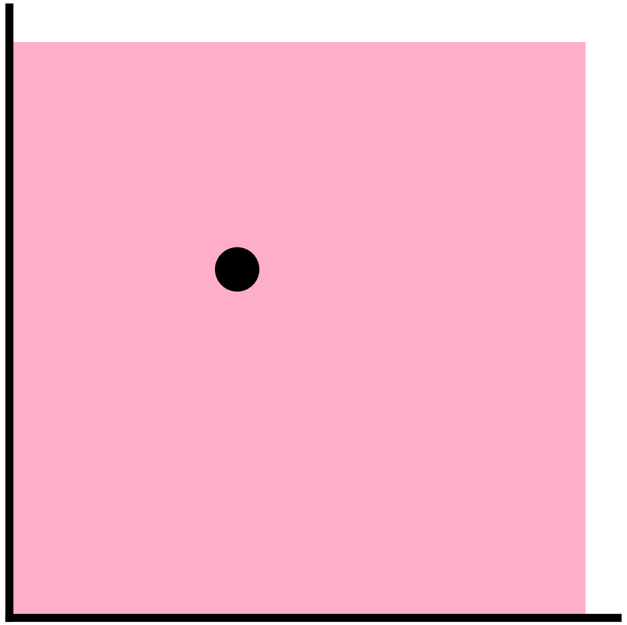
```
#.Given x≥0, let y be the square root of x.
```

Before			After
x	-4	input variable	arbitrary
y	9		

Statement Specifications

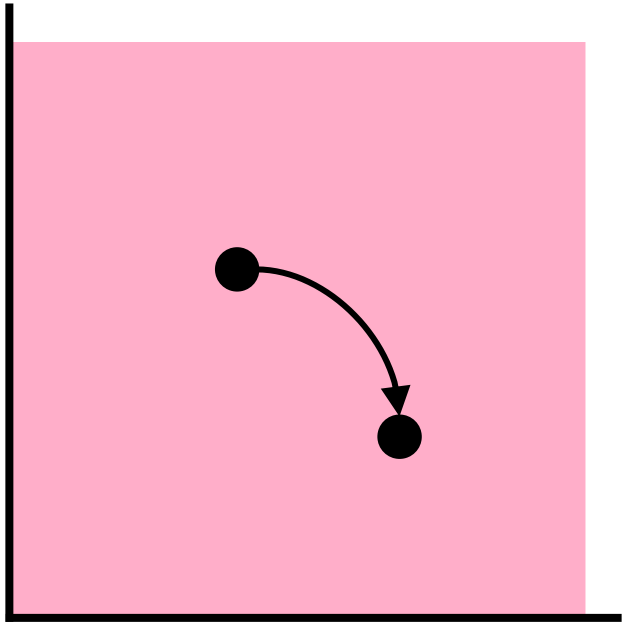
Conditions and Sets of States

Interrupt a program's execution. Before powering the computer down, save all that you will need to resume later. This is the state.



state and state space

The effect of executing code is to transition from one state to another.

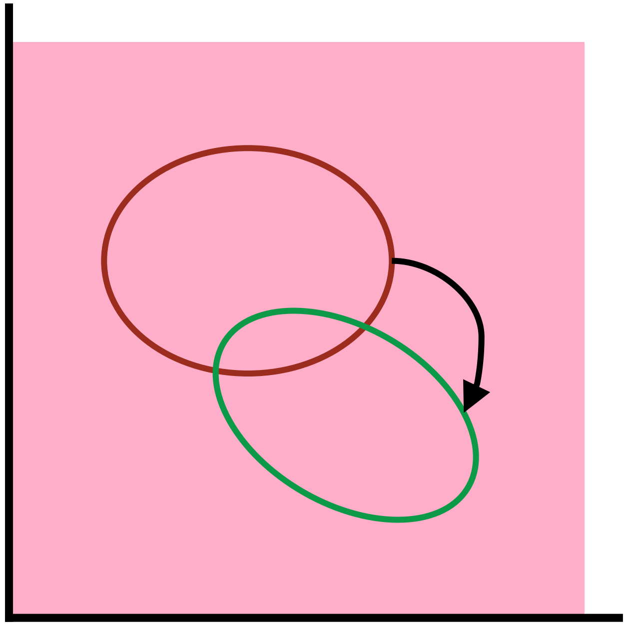


state transition

Statement Specifications

Conditions and Sets of States

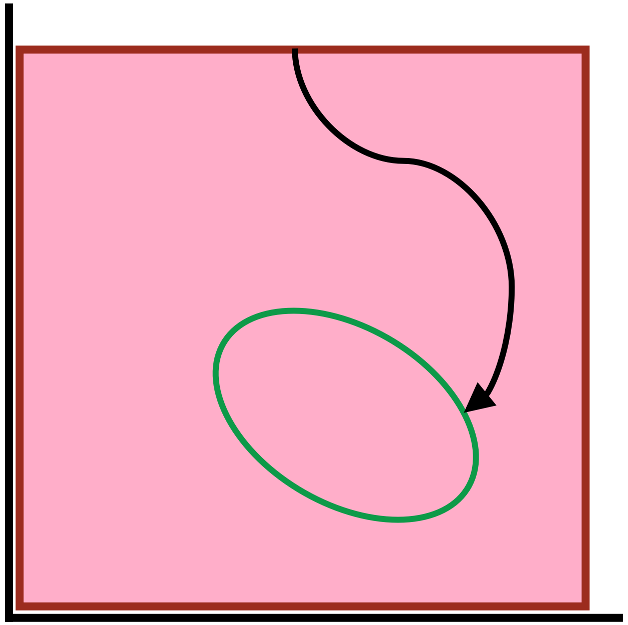
```
#.Given precondition, establish postcondition .
```



precondition to postcondition

The specification requires transition from any state satisfying the precondition to some state satisfying the postcondition.

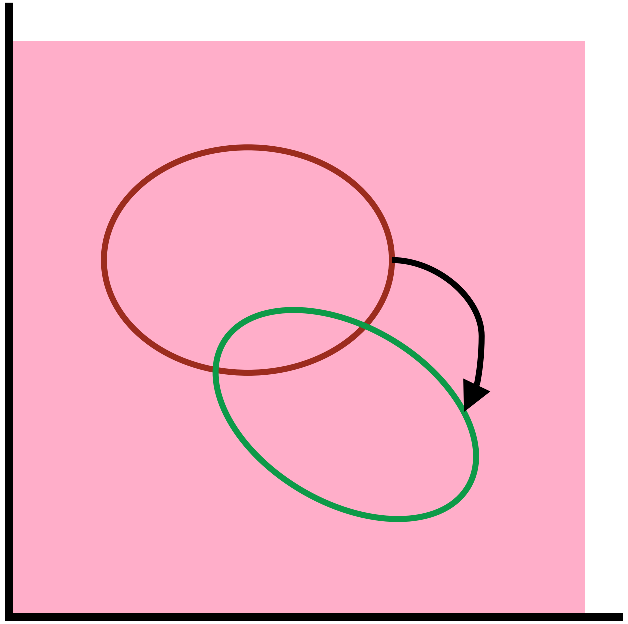
```
#.Output "Hello World".
```



precondition to postcondition

The specification requires transition from any state whatsoever to a state where the output ends with "Hello World".

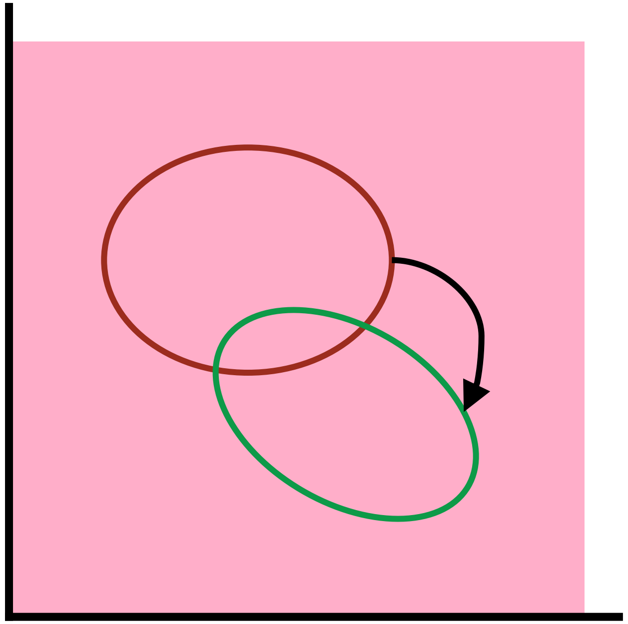
```
#.Swap x and y.
```



precondition to postcondition

The specification requires transition from any state containing variables x and y to a state where the contents of x and y have been exchanged.

Define sets of states either in English, or using Boolean expressions.

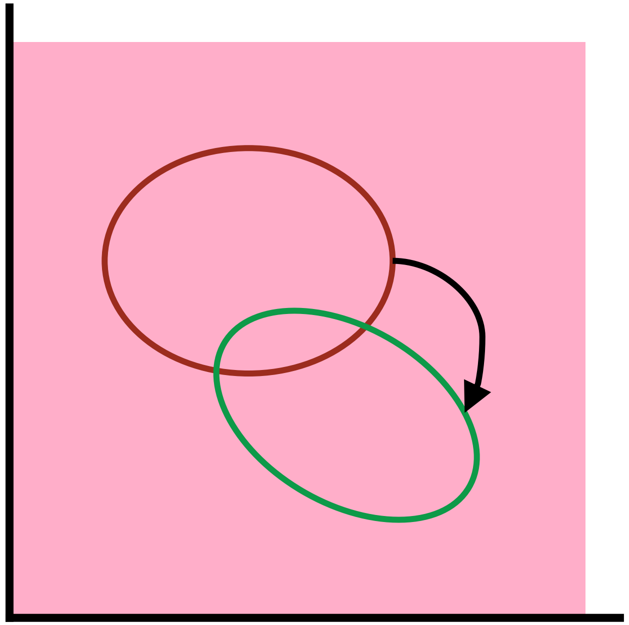


precondition to postcondition

In code, Boolean expressions control execution flow:

```
# Set y to the square root of x if x is
#   not negative, and 0 otherwise.
if x >= 0:
    y = math.sqrt(x)
else:
    y = 0
```

Define sets of states either in English, or using Boolean expressions.



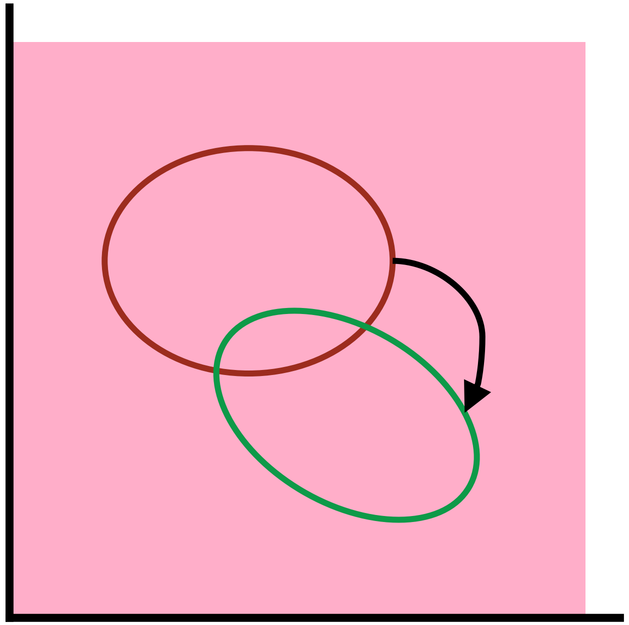
precondition to postcondition

In specifications, Boolean expressions define state sets:

```
#.Given x≥0, let y be the square root of x.
```

Specifically, the set of all states in which the given Boolean expression is **true**.

Transition to *any* state satisfying the postcondition is allowed.



precondition to postcondition

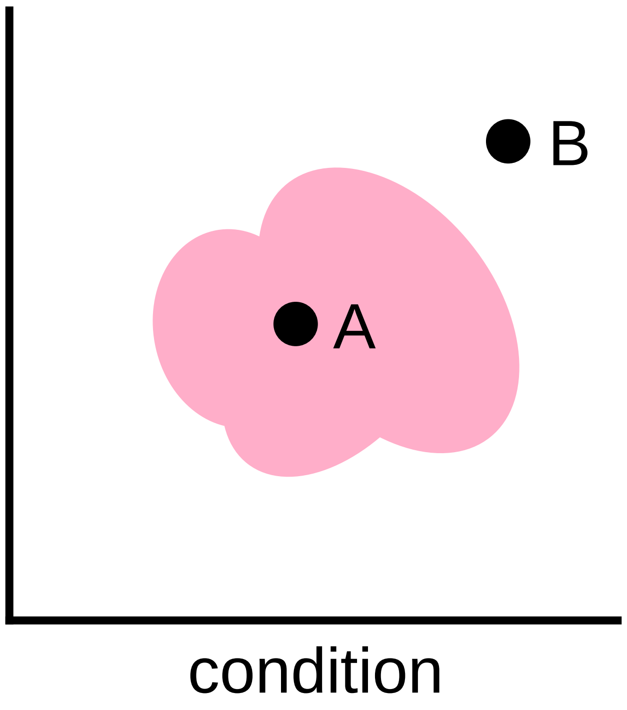
For example,

```
# Given x≥0, let y be the square root of x.
y = math.sqrt(x)
```

or

```
# Given x≥0, let y be the square root of x.
y = -math.sqrt(x)
```

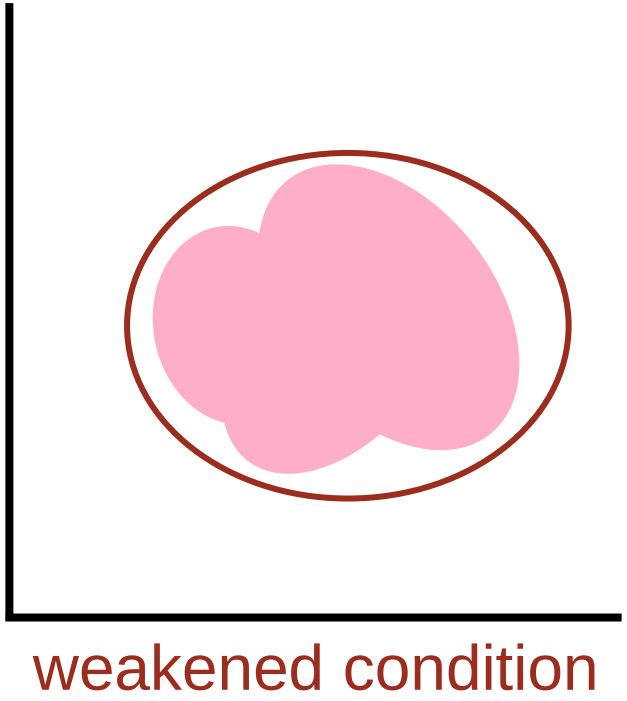
A state either satisfies a condition, or it doesn't.



Statement Specifications

Conditions and Sets of States

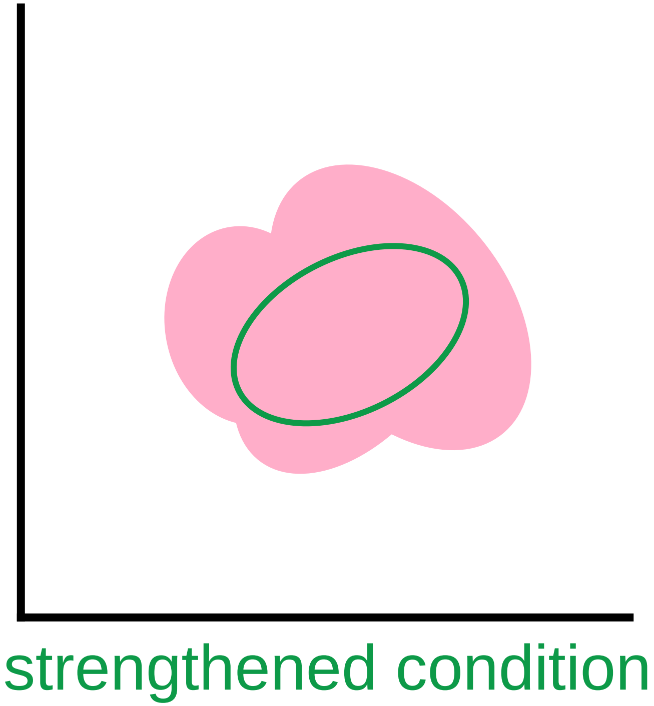
A weakened condition satisfies more states than the original condition.



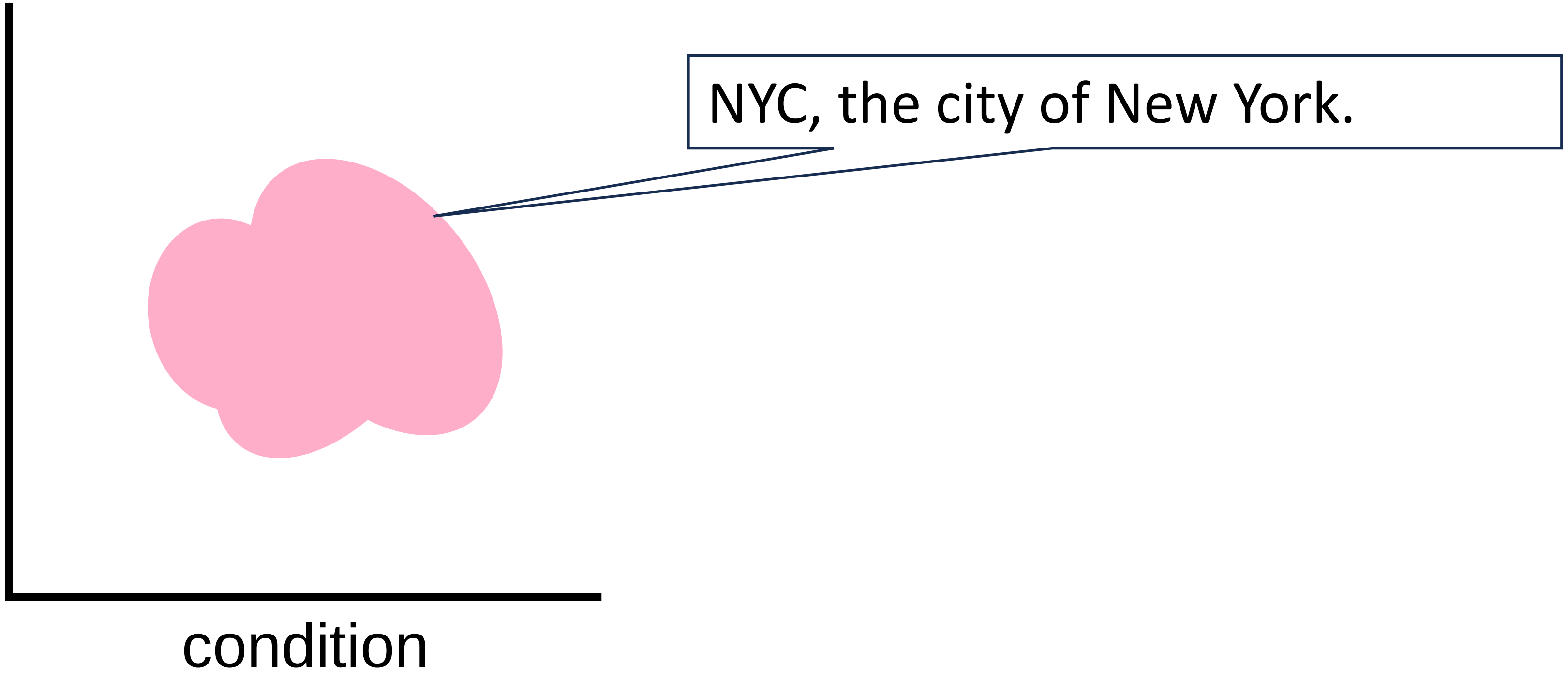
Statement Specifications

Conditions and Sets of States

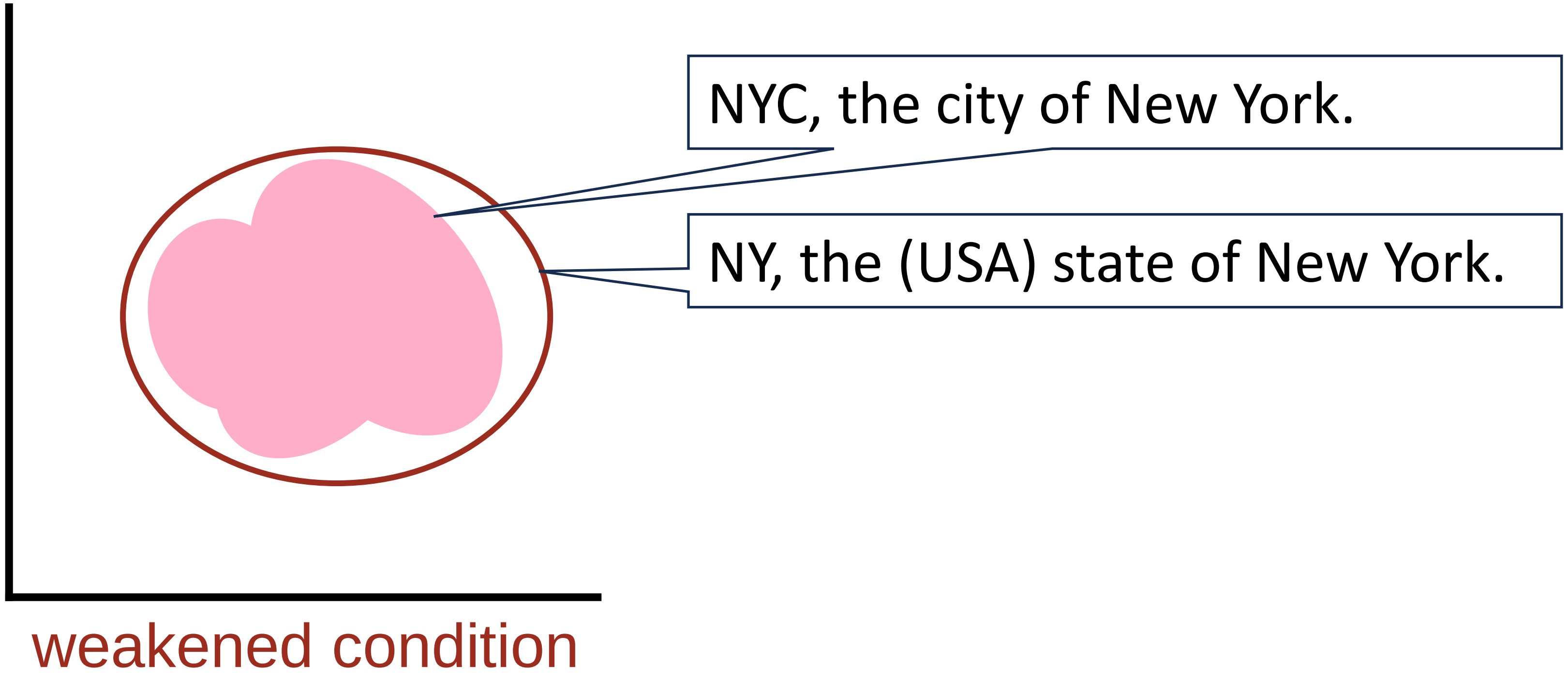
A strengthened condition satisfies fewer states than the original condition.



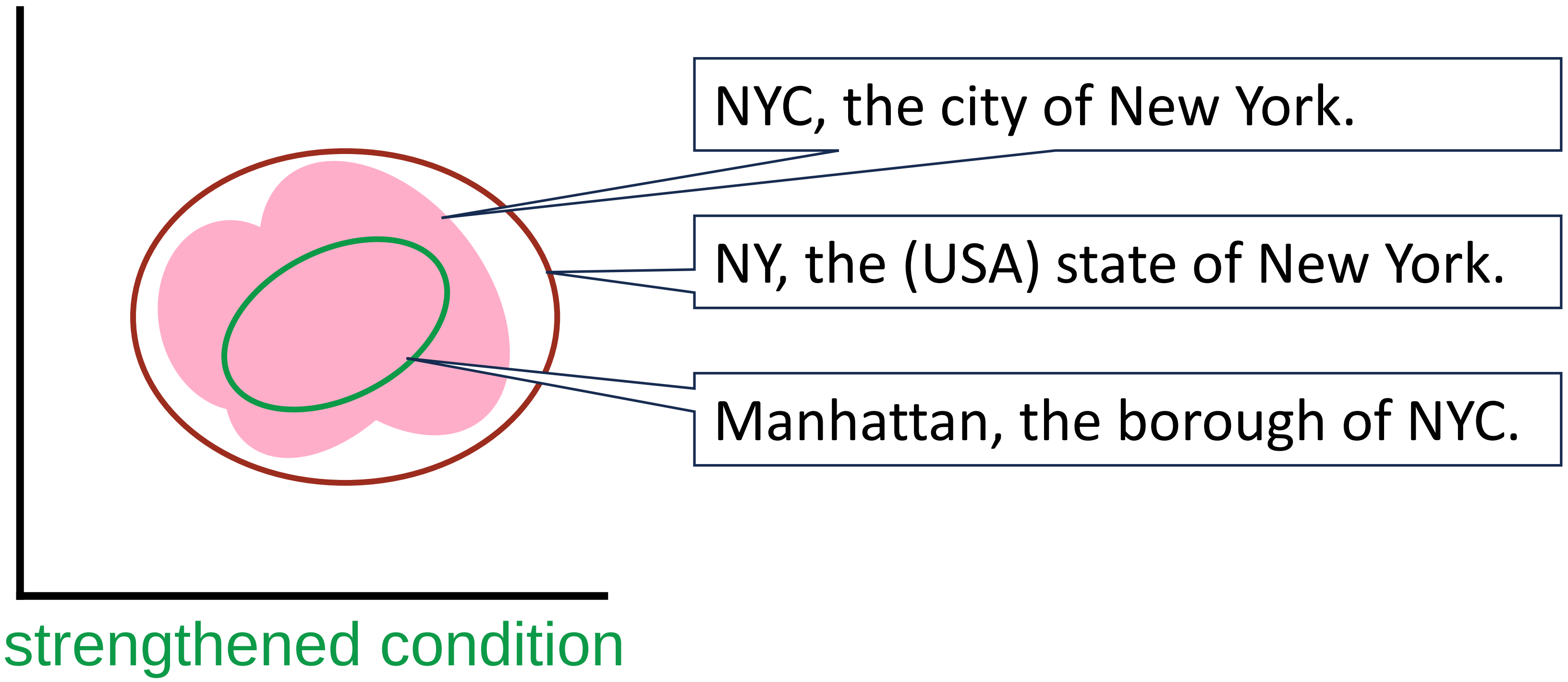
A geographical example:



A geographical example:



A geographical example:



Methods can protect themselves from misuse by their clients by explicitly checking the validity of their arguments, and aborting execution if any are invalid.

Such a protection may derive from a built-in check, e.g., integer division by 0 aborts execution:

```
# Given n!=0, return x/n.
def nth(x:int, n:int) -> int:
    return x / n
```

Consider this computation. If n turns out to be 0 by mistake, method nth will abort execution:

```
# Compute n!=0 such that blah blah.
...
```

```
# Given n!=0, let y be the nth part of x.
y = nth(x, n)
```

```
# Whatever.
...
```

Aborting execution early is far better than having Whatever crash (possibly) much later due to a crazy value of n.

This similar code is just as vulnerable to the error in the computation of n, but without the protection of `nth`, will likely crash in `Whatever`.

```
# Compute n!=0 such that blah blah.
...
```

```
# Given n!=0, do Whatever.
...
```

It can protect itself by doing the same check as `nth` using an **assert**:

```
# Compute n!=0 such that blah blah.
...
```

```
# Given n!=0, do Whatever.
assert n!=0, "blah blah computed a zero n"
...
```

⚠ Abort execution early if the precondition of `Whatever` doesn’t hold

It can protect itself by doing the same check as `nth` using an **assert**:

```
# Compute n!=0 such that blah blah.
...
assert n!=0, "blah blah computed a zero n"
```

```
# Given n!=0, do Whatever.
...
```

⚠ or if the postcondition of blah blah doesn't hold.

It can protect itself by doing the same check as `nth` using an **assert**:

```
# Compute n!=0 such that blah blah.
...
assert n!=0, "blah blah computed a zero n"
```

```
# Given n!=0, do Whatever.
...
```

⚠ Use of **assert** is preferable to debugging.

Long specifications can continue on multiple lines, but need their own #.

```
#.This is a long specification stretching over multiple
#   lines. When it does so, the “continuation lines” have
#   their own hash marks (#), followed by three spaces.
```

```
#.This allows each specification to be read separately and
#   not confused with a next, quite different specification.
```

Declaration Specifications take a data-centric perspective.

```
Declaration-of-one-variable  # Specification.
```

A declaration specification provides a representation invariant for the variable that characterizes the value contained therein. It is a global precondition for every statement in the scope of the variable (except for brief moments before the variable has been updated).

The specification is akin to a glossary entry, and can be used as such. Think of the specification as being exactly what you want to know (or be reminded of) when inspecting or writing code that uses the variable.

Example: Suppose input values are to be read and “processed”.

Here are two specifications that provide different possible representation invariants for the variable `count`.

```
count: int   # Number of input values read so far.

count: int   # Number of input values processed so far.
```

In the first case, count should be incremented immediately upon reading a value. In the second case, count is only incremented when the program gets around to processing the value it has already read.

Example: A group of related variables, called a *data structure*, may share a representation invariant. In this case, it is advantageous to provide a specification for the whole group as well as for the individual components.

```
# A[0..size-1] are the current int items in a list, 0≤size≤max_size.
A: list[int]   # A[] is the receptacle for items of a list.
size: int      # size is the current number of items in A[], 0≤size≤max_size.
max_size: int  # max_size is the maximum number of items storable in A[].
```

The representation invariant characterizes how A, size, and max_size relate to one another.

Alternate form:

Some IDE editors also support a slightly different syntax for Declaration Specifications:

```
Declaration-of-one-variable
"""Specification."""
```

In such editors, you may hover over the variable name (in a use distant from the declaration) and a helpful pop-up containing the specification appears.

Alternate form:

Some IDE editors also support a slightly different syntax for Declaration Specifications:

```
Declaration-of-one-variable
"""Specification."""
```

In such editors, you may hover over the variable name (in a use distant from the declaration) and a helpful pop-up containing the specification appears.

If you use this form of Declaration Specification, it is best to separate it from the next line with blank line, so that it is clear that the specification goes with the variable being declared *before* it.

Method Specifications

A method specification describes the effects (if any) and the return value (if any) of the method in terms of its parameters. This is its postcondition.

```
def name(parameters) -> type:
    """Specification."""
    block
```

A method specification describes the effects (if any) and the return value (if any) of the method in terms of its parameters. This is its postcondition.

```
def name(parameters) -> type:
    """Specification."""
    block
```

Example

```
def sort(A:list[int], n:int) -> None:
    """sort(A, n) rearranges array A[0..n-1] to be in non-decreasing order."""
    ⟨body of sort⟩
```

A method specification describes the effects (if any) and the return value (if any) of the method in terms of its parameters. This is its postcondition.

```
def name(parameters) -> type:
    """Specification."""
    block
```

Example

```
def max(x: int, y: int) -> int:
    """max(x, y) returns the larger of the values x and y."""
    if x < y:
        return y
    else:
        return x
```

A method specification may restrict its parameters. This is its precondition.

```
def name(parameters) -> type:
    """Specification."""
    block
```

Example

```
def find(A: list[int], n: int, v: int) -> int:
    """
    Given int array A[0..n-1] sorted in non-decreasing order, and int v, find(A, n, v)
    returns an index k where A[k]==v, or returns n if v does not occur in A.
    """
    ⟨blank line⟩
    ⟨body of find⟩
```

> As with variable specifications, think of a method specification as being exactly what you want to know (or be reminded of) in a pop up of an IDE's editor either when inspecting code that uses the method, or when contemplating a call to it.

A method specification may restrict its parameters. This is its precondition.

```
def name(parameters) -> type:
    """Specification."""
    block
```

Example

```
def find(A: list[int], n: int, v: int) -> int:
    """
    Given int array A[0..n-1] sorted in non-decreasing order, and int v, find(A, n, v)
    returns an index k where A[k]==v, or returns n if v does not occur in A.
    """
    ⟨blank line⟩
    ⟨body of find⟩
```

A class specification summarizes the class's purpose, functionality, and history. The specifications of the class's public methods (and variables) are implicitly part of the class specification, but a list of them (without specifications) is common.

```
class name:
    """Specification."""
    declarations-statements-and-method-definitions
```

Class specifications are often more descriptive and historical than the other forms of specification.

```
class Rational:
    """
    Rational. A module for the manipulation of rationals, including operations
      for +, -, *, /, conversion to Str, and equality.
    Author: Joe Blow.
    Created: 12/25/2022.
    Revision History: Converted to use unbounded integers, 12/25/2023.
    """
    ⟨blank line⟩
    ⟨body of class Rational⟩
```