Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Introduction

You can learn a programming language but still not know how to program.

Introduction

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You’ve seen finished programs, but struggle to produce one yourself.

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You’ve seen finished programs, but struggle to produce one yourself.

You will grope in the dark until someone shows you a methodology.

Introduction

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You’ve seen finished programs, but struggle to produce one yourself.

You will grope in the dark until someone shows you a methodology.

I aim to provide you with that methodology.

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You’ve seen finished programs, but struggle to produce one yourself.

You will grope in the dark until someone shows you a methodology.

I aim to provide you with that methodology.

Since I don’t want to teach a language, I’ll stick to a tiny, universal one.

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You’ve seen finished programs, but struggle to produce one yourself.

You will grope in the dark until someone shows you a methodology.

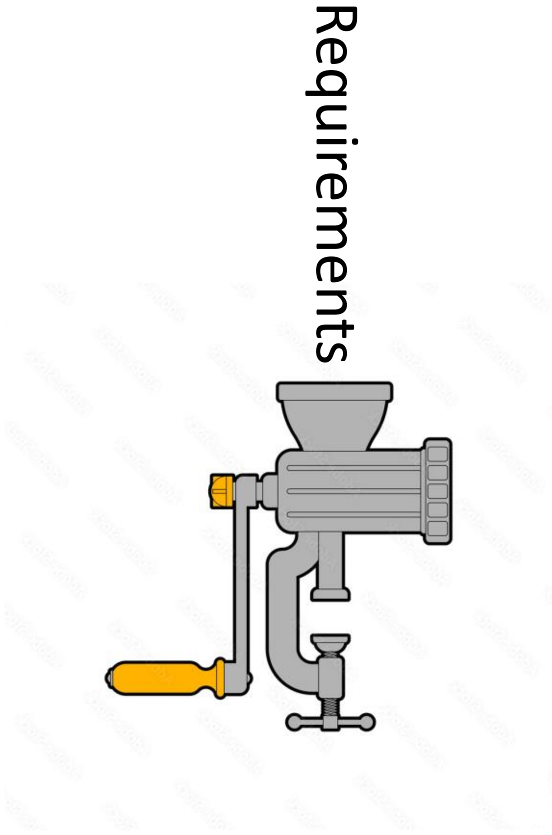
I aim to provide you with that methodology.

Since I don’t want to teach a language, I’ll stick to a tiny, universal one.

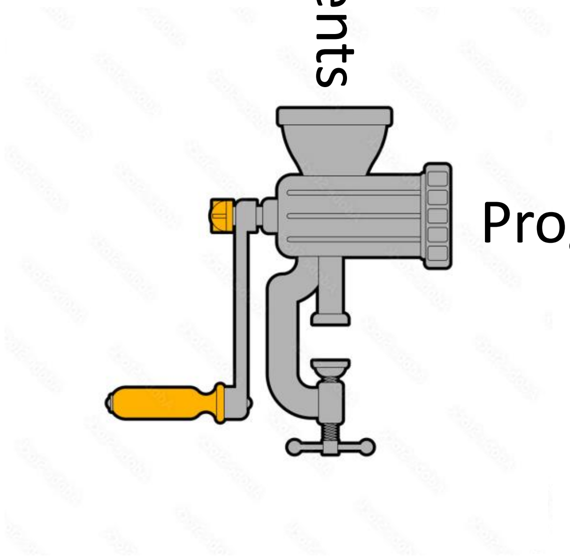
It’s a subset of Java, Python, C/C++, JavaScript, …, (any imperative language).

Can programming be mechanized?

Can programming be mechanized?

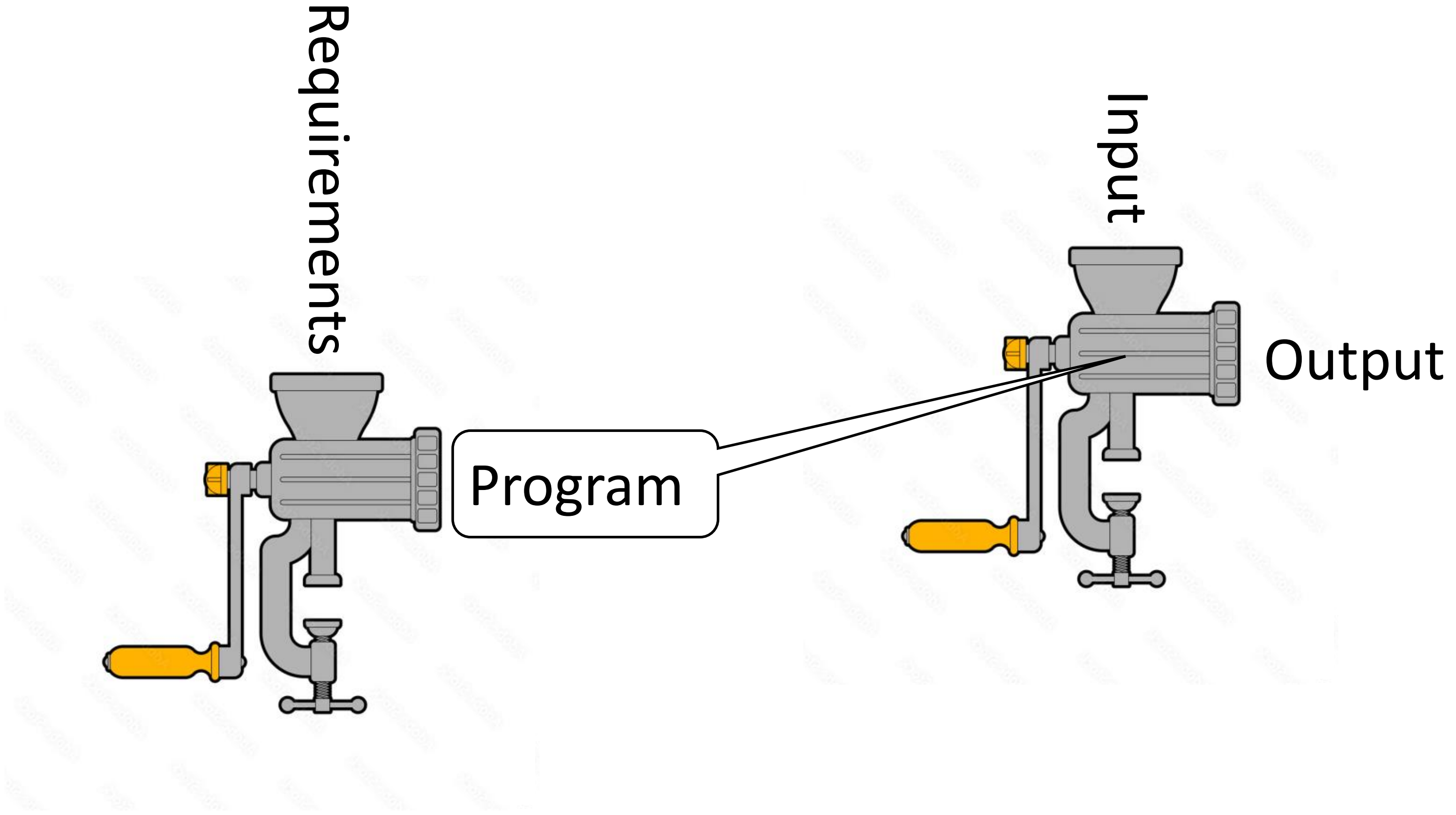


Can programming be mechanized?

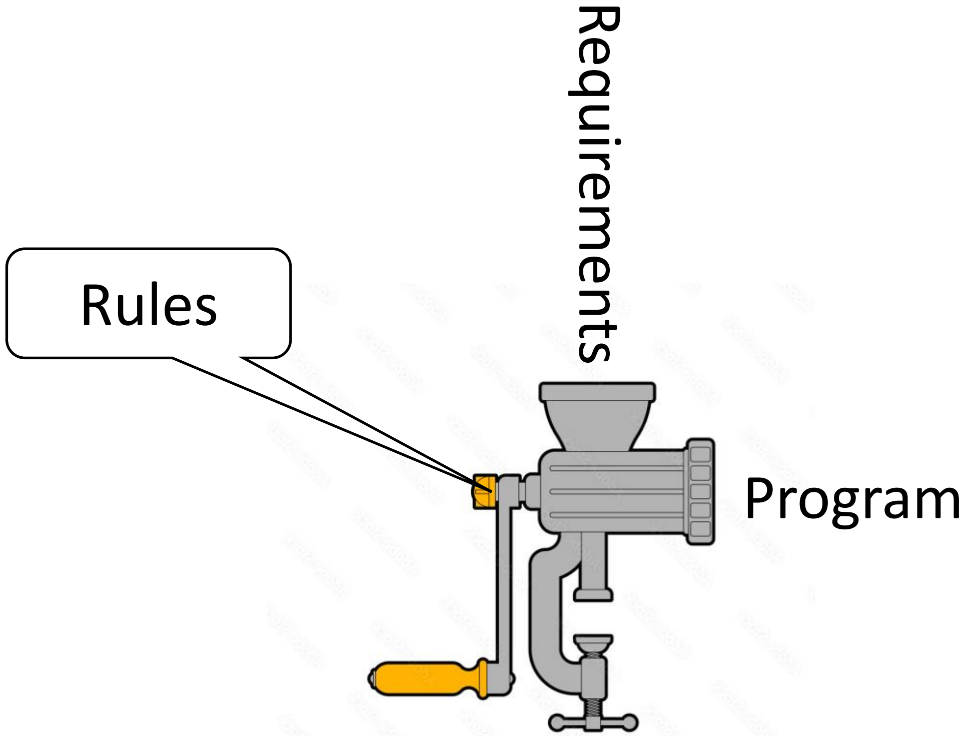


Motivation

Can programming be mechanized?



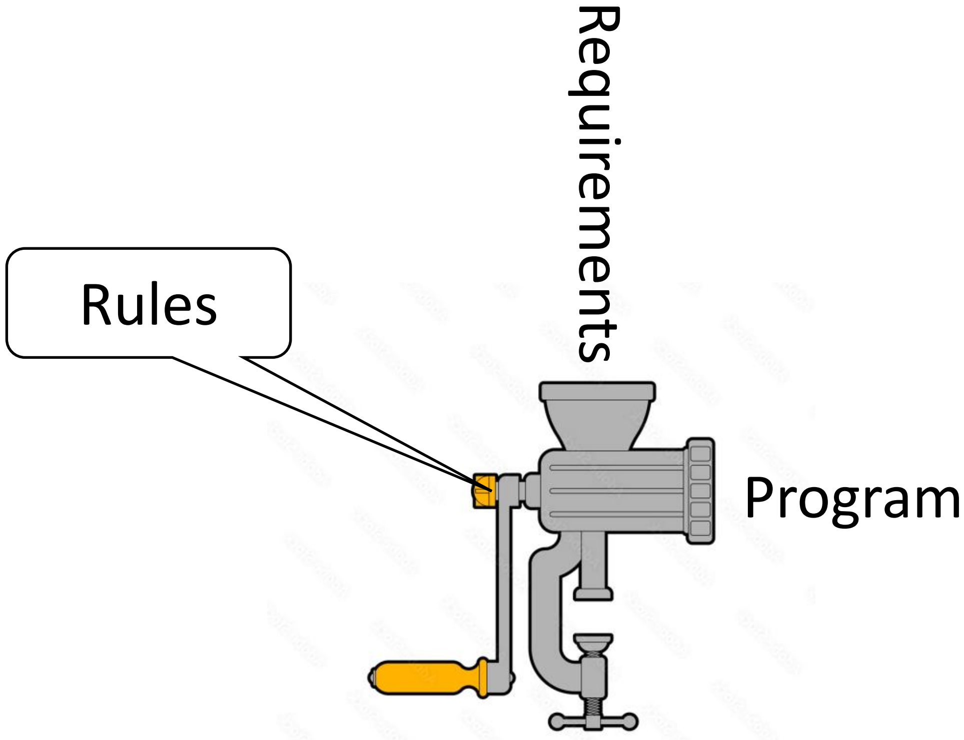
Can programming be mechanized?



Fully-automatic programming would need rules that are:

- Effective
- Produce good code
- Efficient
- Complete

Can programming be mechanized?



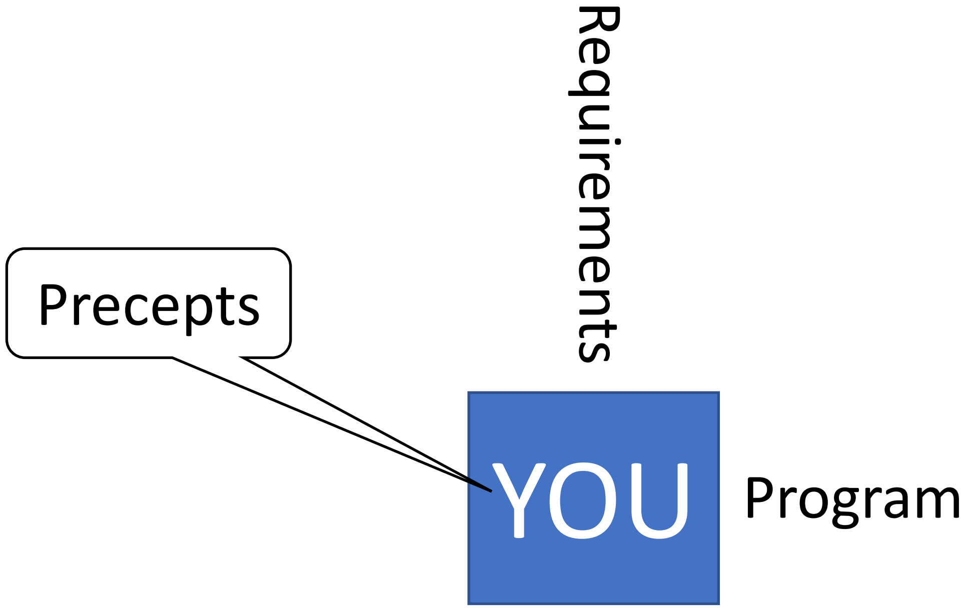
Fall back

- Rules for people
- Make programming
  - Easy
  - Accurate

Precepts

pre·cept

A command or principle intended especially as a general rule of action



Fall back

- Rules for people
- Make programming
  - Easy
  - Accurate

pre·cept

A command or principle intended especially as a general rule of action

First Precept

pre·cept

A command or principle intended especially as a general rule of action

First Precept

☞ **Follow programming precepts.**

pre·cept

A command or principle intended especially as a general rule of action

Second Precept

pre·cept

A command or principle intended especially as a general rule of action

Second Precept

☞ **Ignore precepts, when appropriate.**

pre·cept

A command or principle intended especially as a general rule of action

☞ **Code with deliberation. Be mindful.**

Sample precept

Sample precept

☞ **Aspire to making code self-documenting by choosing descriptive names.**

Sample precept, with an application:

```
amount = price * quantity
```

☞ **Aspire to making code self-documenting by choosing descriptive names.**

Same precept, with a different application:

```
piece = board[row + deltaRow[direction]][column + deltaColumn[direction]]
```

☞ **Aspire to making code self-documenting by choosing descriptive names.**

Same precept, with a different application:

```
piece = board[row + deltaRow[direction]][column + deltaColumn[direction]]

piece = B[r + deltaR[d]][c + deltaC[d]]
```

☞ **Aspire to making code self-documenting by choosing descriptive names.**

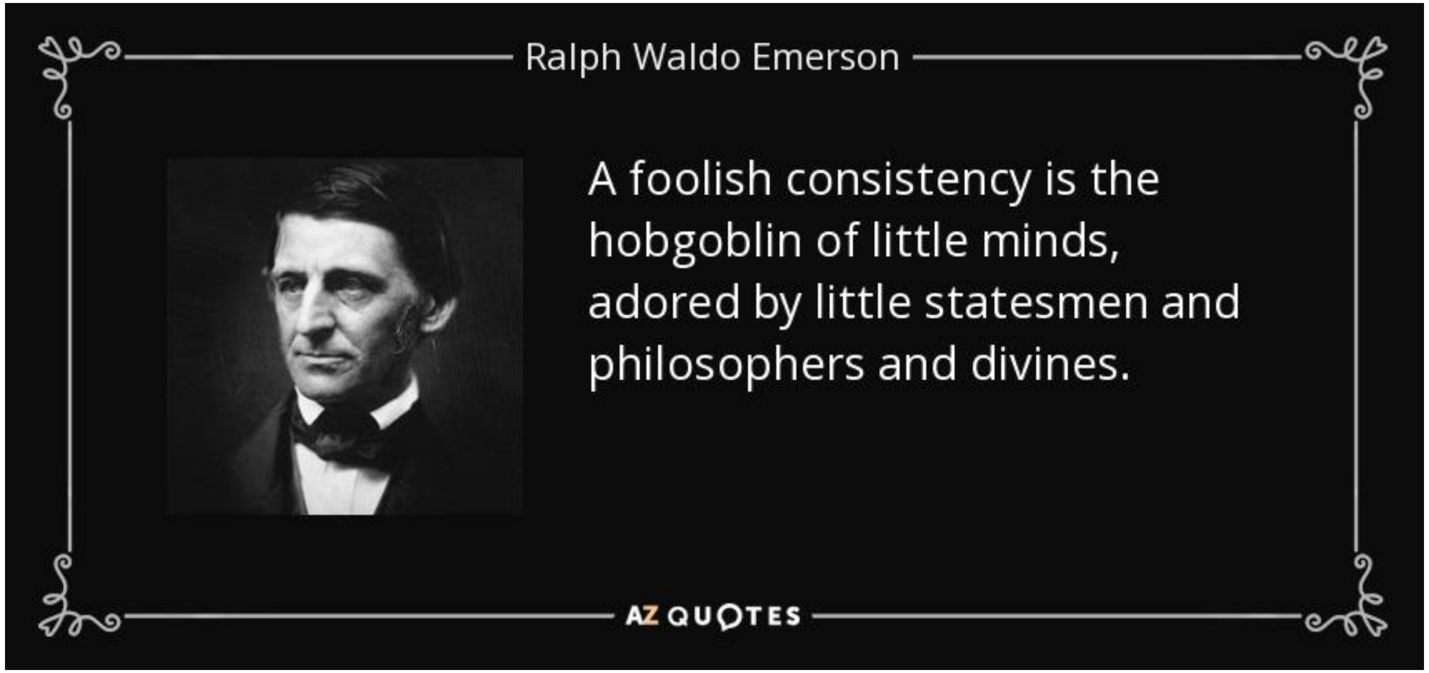
Alternative precept

```
piece = board[row + deltaRow[direction]][column + deltaColumn[direction]]
```

```
piece = B[r + deltaR[d]][c + deltaC[d]]
```

☞ **Use single-letter variable names when it makes code more understandable.**

Precepts



☞ **Resolve contradictory precepts with care.**

Exercise judgement

Make tradeoffs

Don’t make decisions casually

Indulge in personal preference

☞ **Resolve contradictory precepts with care.**

Precepts

☞ **Be humble. Programming is hard and error prone. Respect it.**

Despite your humility, aim for perfection

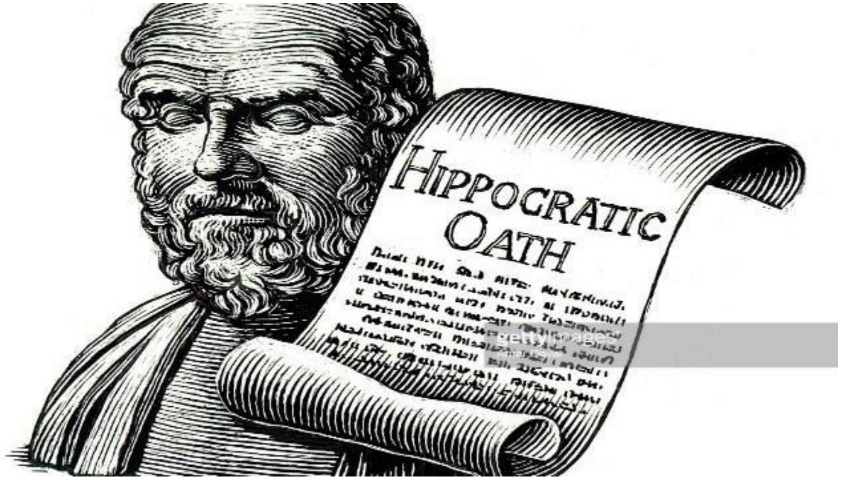
- The quality of the code you write
- The quality of the process you use to write it

☞ **Be humble. Programming is hard and error prone. Respect it.**

Process quality

☞ **Aspire to code it right the first time.**

Process quality: Hippocratic Coding



☞ **Aspire to code it right the first time. Do no harm. Avoid writing code that must be redone.**

Precepts

An approach to Hippocratic Coding: Patterns

☞ **Master stylized code patterns, and use them.**

An approach to Hippocratic Coding: Patterns

A *pattern* is a *structure* containing *placeholders.*

The *structure* is an arrangement of *computational elements*.

The *placeholders* are named slots to be filled in:

They may be words or phrases in *italics*.

They may be *comments*, which are hash marks (#) followed by text.

Pattern: compute-use

```
#.Compute.
#.Use.
```

☞ **Master stylized code patterns, and use them.**

- The *structure* of the *compute-use pattern* is a sequence of two *statements* that command actions to be performed one after the other.
- The *placeholders* describe the actions: compute something, then use it.

Pattern: compute-use

```
#.Compute.
#.Use.
```

☞ **Master stylized code patterns, and use them.**

Pattern: compute-use

```
#.Compute.
#.Use.
```

- The *structure* of the *compute-use pattern* is a sequence of two *statements* that command actions to be performed one after the other.
- The *placeholders* describe the actions: compute something, then use it.
- The dot (`.`) at the end of a *placeholder* is just a period that doesn't signify anything.
- The dot (`.`) immediately after the hash mark (`#.`) is notation used in this book to signify that the *placeholder* has not yet been filled in.

☞ **Master stylized code patterns, and use them.**

Pattern: compute-use

```
#.Compute.
#.Use.
```

- The *structure* of the *compute-use pattern* is a sequence of two *statements* that command actions to be performed one after the other.
- The *placeholders* describe the actions: compute something, then use it.
- The dot (`.`) at the end of a *placeholder* is just a period that does not signify anything.
- The dot (`.`) immediately after the hash mark (`#.`) is notation used in this book to signify that the *placeholder* remains to be filled in.

How does this pattern help?

☞ **Master stylized code patterns, and use them.**

Sample programming problem

☞ **Master stylized code patterns, and use them.**

Apply the compute-use pattern

```
#.Compute k.
#.Use k.
```

☞ **Master stylized code patterns, and use them.**

How does this *pattern* help?

- It divides the problem into two smaller parts.
- It describes those parts, and clarifies that the first step computes something named k, and the second step uses k.

Apply the compute-use pattern

```
#.Compute k.
#.Use k.
```

☞ **Master stylized code patterns, and use them.**

How does this *pattern* help?

- It divides the problem into two smaller parts.
- It describes those parts, and clarifies that the first step computes something named k, and the second step uses k.
- It's Hippocratic: A baby step that does no harm.

Apply the compute-use pattern

```
#.Compute k.
#.Use k.
```

☞ **Master stylized code patterns, and use them.**

How does this *pattern* help?

- It divides the problem into two smaller parts.
- It describes those parts, and clarifies that the first step computes something named k, and the second step uses k.
- It's Hippocratic: A baby step that does no harm.
- We can then:
  - Replace the first *placeholder* with code.

Apply the compute-use pattern

```
k = thus-and-such
#.Use k.
```

☞ **Master stylized code patterns, and use them.**

Apply the compute-use pattern

```
k = thus-and-such
if k-has-some-desired-property:
    #.Do-this-and-that.
```

How does this *pattern* help?

- It divides the problem into two smaller parts.
- It describes those parts, and clarifies that the first step computes something named k, and the second step uses k.
- It's Hippocratic: A baby step that does no harm.
- We can then:
  - Replace the first *placeholder* with code.
  - Replace the second *placeholder* with code.

  and we're making progress.

☞ **Master stylized code patterns, and use them.**

Apply the compute-use pattern

```
k = thus-and-such
if k-has-some-desired-property:
    #.Do-this-and-that.
```

How does this *pattern* help?

- It divides the problem into two smaller parts.
- It describes those parts, and clarifies that the first step computes something named k, and the second step uses k.
- It's Hippocratic: A baby step that does no harm.
- We can then:
  - Replace the first *placeholder* with code.
  - Replace the second *placeholder* with code.

  and we're making progress.
- The *Compute* and *Use placeholders* are gone, but the *compute-use pattern* is the skeleton that underlies the code.

☞ **Master stylized code patterns, and use them.**

- An alternative to *replacing* a *placeholders* is to …

Apply the compute-use pattern

```
#.Compute k.
#.Use k.
```

☞ **Master stylized code patterns, and use them.**

- An alternative to *replacing* a *placeholders* is to *amplify* it with specifics that say exactly what a step must do, effectively turning it into a *specification*.

Apply the compute-use pattern

```
#.Compute k, some specific aspect of the big-hairy-mess.
#.Use k, the specific aspect of the big-hairy-mess that has been computed.
```

☞ **Master stylized code patterns, and use them.**

- An alternative to *replacing* a *placeholder* is to *amplify* it with specifics that say exactly what a step must do, effectively turning it into a *specification*.
- The *specification* can then be *implemented*, either by code, or by an instance of another pattern, and remains to show the intent of its *refinement*.

Apply the compute-use pattern

```
# Compute k, some specific aspect of the big-hairy-mess.
k = thus-and-such

# Use k, the specific aspect of the big-hairy-mess that has been computed.
if k-has-some-desired-property:
    #.Do-this-and-that.
```

☞ **Master stylized code patterns, and use them.**

Apply the compute-use pattern

```
# Compute k, some specific aspect of the big-hairy-mess.
k = thus-and-such

# Use k, the specific aspect of the big-hairy-mess that has been computed.
if k-has-some-desired-property:
    #.Do-this-and-that.
```

- An alternative to *replacing* a *placeholder* is to *amplify* it with specifics that say exactly what a step must do, effectively turning it into a *specification*.
- The *specification* can then be *implemented*, either by code, or by an instance of another pattern, and remains to show the intent of its *refinement*.

- When a *specification* is *implemented*:
  - The dot (.) gets removed.

☞ **Master stylized code patterns, and use them.**

Apply the compute-use pattern

```
# Compute k, some specific aspect of the big-hairy-mess.
k = thus-and-such

# Use k, the specific aspect of the big-hairy-mess that has been computed.
if k-has-some-desired-property:
    #.Do-this-and-that.
```

- An alternative to *replacing* a *placeholder* is to *amplify* it with specifics that say exactly what a step must do, effectively turning it into a *specification*.
- The *specification* can then be *implemented*, either by code, or by an instance of another pattern, and remains to show the intent of its *refinement*.

- When a *specification* is implemented:
  - The dot (`.`) gets removed.
  - A blank line is inserted, if necessary, to separate the steps from one another.

☞ **Master stylized code patterns, and use them.**

Another pattern: indeterminate iteration

```
# Enumerate from start.
k = start
while condition:
    k += 1
```

☞ **Master stylized code patterns, and use them.**

Another pattern: indeterminate iteration

```
# Enumerate from start.
k = start
while condition:
    k += 1
```

Effect

Initialize k to *start*

Repeatedly add 1 to k

provided *condition* is **True**

☞ **Master stylized code patterns, and use them.**

Yet another pattern: general iterative computation

```
#.Initialize.
while not-finished:
    #.Compute.
    #.Go-on-to-next.
```

☞ **Master stylized code patterns, and use them.**

Yet another pattern: general iterative computation

```
#.Initialize.
while not-finished:
    #.Compute.
    #.Go-on-to-next.
```

Effect

Get ready by initializing

Repeatedly make progress by:

- computing something
- moving on to the next thing

☞ **Master stylized code patterns, and use them.**

Indeterminate iteration is a special case of general iterative computation.

```
#.Initialize.
while not-finished:
    #.Compute.
    #.Go-on-to-next.
```

```
k = start
while condition:
    k += 1
```

☞ **Master stylized code patterns, and use them.**

Indeterminate iteration is a special case of general iterative computation.

```
#.Initialize.
while not-finished:
    #.Compute.
    #.Go-on-to-next.
```

```
k = start
while condition:
    k += 1
```

☞ **Master stylized code patterns, and use them.**

Indeterminate iteration is a special case of general iterative computation.

```
#.Initialize.
while not-finished:
    #.Compute.
    #.Go-on-to-next.
```

```
k = start
while condition:
    k += 1
```

☞ **Master stylized code patterns, and use them.**

Indeterminate iteration is a special case of general iterative computation.

```
#.Initialize.
while not-finished:
    #.Compute.
    #.Go-on-to-next.
```

```
k = start
while condition:
    k += 1
```

☞ **Master stylized code patterns, and use them.**

Indeterminate iteration is a special case of general iterative computation.

```
#.Initialize.
while not-finished:

    #.Go-on-to-next.
```

```
k = start
while condition:
    k += 1
```

☞ **Master stylized code patterns, and use them.**

Shorthand: iterative computation with index increasing by 1

```
for name in range(expression₁, expression₂):
    compute
```

☞ **Master stylized code patterns, and use them.**

Shorthand: iterative computation with index increasing by 1

```
for name in range(expression1, expression2):
    compute
```

```
#.Initialize.
while not-finished:
    #.Compute.
    #.Go-on-to-next.
```

☞ **Master stylized code patterns, and use them.**

Shorthand: iterative computation with index increasing by 1

```
for name in range(expression1, expression2):
    compute
```

```
name = expression1
while not-finished:
    #.Compute.
    #.Go-on-to-next.
```

☞ **Master stylized code patterns, and use them.**

Shorthand: iterative computation with index increasing by 1

```
for name in range(expression_1, expression_2):
    compute
```

```
name = expression_1
while not-finished:
    #.Compute.
    name += 1
```

☞ **Master stylized code patterns, and use them.**

Shorthand: iterative computation with index increasing by 1

```
for name in range(expression1, expression2):
    compute
```

```
name = expression1
while name < expression2:
    #.Compute.
    name += 1
```

☞ **Master stylized code patterns, and use them.**

Shorthand: iterative computation with index increasing by 1

```
for name in range(expression_1, expression_2):
    compute
```

```
name = expression_1
while name < expression_2:
    #.Compute.
    name += 1
```

☞ **Master stylized code patterns, and use them.**

Another Shorthand: iterative computation with index decreasing by 1

```
for name in range(expression1, expression2, -1):
    compute
```

```
name = expression1
while name > expression2:
    #.Compute.
    name -= 1
```

☞ **Master stylized code patterns, and use them.**

Key Distinction: determinate iteration vs indeterminate iteration

Determinate: for when the number of iterations is known beforehand

```
for name in range(expression1, expression2):
    compute
```

Indeterminate: for when the number of iterations is not known beforehand

```
k = start
while condition:
    k += 1
```

☞ **Master stylized code patterns, and use them.**

- "Known" in the sense that the number of iterations is determined by the values of *expression*$_1$ and *expression*$_2$.

Key Distinction: determinate iteration vs indeterminate iteration

Determinate: for when the number of iterations is known beforehand

```
for name in range(expression1, expression2):
    compute
```

Indeterminate: for when the number of iterations is not known beforehand

```
k = start
while condition:
    k += 1
```

☞ **Master stylized code patterns, and use them.**

Another approach to Hippocratic Coding: Analysis

☞ **Aspire to code it right the first time. Do no harm. Avoid writing code that must be redone.**

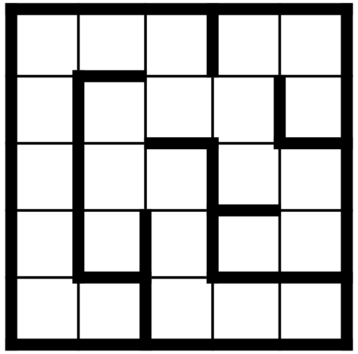
Another approach to Hippocratic Coding: Analysis

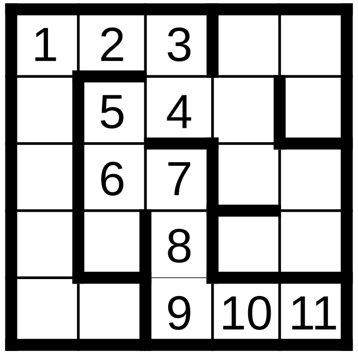
☞ **Analyze first.**

Example: Running a Maze

Background. Define a maze to be a square two-dimensional grid of cells separated (or not) from adjacent cells by walls. One can move between adjacent cells if and only if no wall divides them. A solid wall surrounds the entire grid of cells, so there is no escape from the maze.

Problem Statement. Write a program that inputs a maze, and outputs a direct path from the upper-left cell to the lower-right cell if such a path exists, or outputs “Unreachable” otherwise. A path is direct if it never visits any cell more than once.





Analysis

- Problem
- Architecture
- Data
- Components

☞ **Analyze first.**

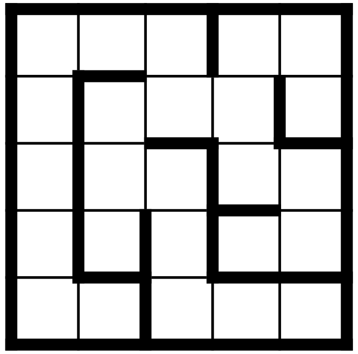
Problem

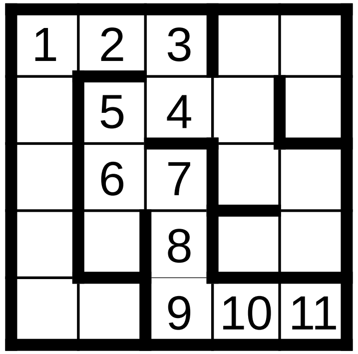
☞ **Make sure you understand the problem.**

Example: Running a Maze

Background. Define a maze to be a square two-dimensional grid of cells separated (or not) from adjacent cells by walls. One can move between adjacent cells if and only if no wall divides them. A solid wall surrounds the entire grid of cells, so there is no escape from the maze.

Problem Statement. Write a program that inputs a maze, and outputs a direct path from the upper-left cell to the lower-right cell if such a path exists, or outputs "Unreachable" otherwise. A path is direct if it never visits any cell more than once.

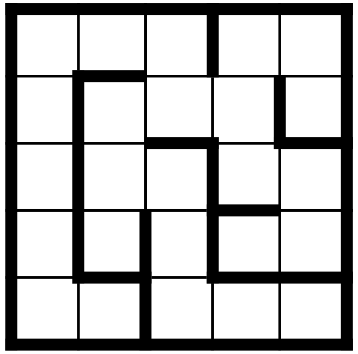


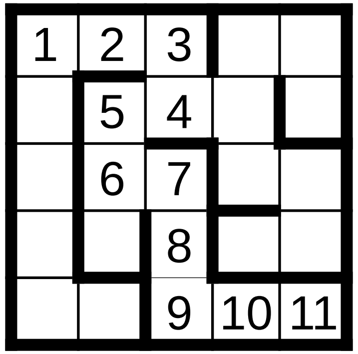


☞ **Make sure you understand the problem.**

Example: Running a Maze

- Do I understand each noun: *maze*, *grid*, *cell*, *wall*, *path*, and *direct path*?
- Do I understand the verbs: Specifically, how does one *move* between cells?
- How is a maze represented in the input?
- Is there any upper limit on the size of a maze? Is there a lower limit?
- What is the expected program behavior if the input is not well-formed?
- Is a *direct* path the same as a *shortest* path?
- What if there is more than one direct path?
- How is a path to be displayed in the output?





☞ **Make sure you understand the problem.**

Architecture: What sort of computation will it be?

Architecture: What sort of computation will it be?

- **Online**. Read a sequence of inputs, and process them on the fly.
- **Offline**. Read all inputs, perform a computation, then output the result.
- **Other**.

Architecture: Offline computation pattern

```
#.Input.
#.Compute.
#.Output.
```

Architecture: Restate the problem within the architectural structure.

```
#.Input a maze of arbitrary size, or output “malformed input” and stop if the
#   input is improper. Input format: TBD.
#.Compute a direct path through the maze, if one exists.
#.Output the direct path found, or “unreachable” if there is none. Output
#   format: TBD.
```

Programs: Instructions for manipulating values

Instructions: code

Values: data

Patterns and Architecture: Code-centered perspective

☞ **Dovetail thinking about code and data.**

Code

```
#.Input a maze of arbitrary size, or output “malformed input” and stop if the
#   input is improper. Input format: TBD.
#.Compute a direct path through the maze, if one exists.
#.Output the direct path found, or “unreachable” if there is none. Output
    format: TBD.
```

☞ **Dovetail thinking about code and data.**

Data

maze

```
#.Input a maze of arbitrary size, or output “malformed input” and stop if the
#   input is improper. Input format: TBD.
#.Compute a direct path through the maze, if one exists.
#.Output the direct path found, or “unreachable” if there is none. Output
    format: TBD.
```

path

☞ **Dovetail thinking about code and data.**

External Data

external data (maze)

```
#.Input a maze of arbitrary size, or output “malformed input” and stop if the
#   input is improper. Input format: TBD.
#.Compute a direct path through the maze, if one exists.
#.Output the direct path found, or “unreachable” if there is none. Output
    format: TBD.
```

external data (path)

☞ **Dovetail thinking about code and data.**

Variables

```
#.Input a maze of arbitrary size, or output “malformed input” and stop if the
#   input is improper. Input format: TBD.
```

maze

```
#.Compute a direct path through the maze, if one exists.
```

path

```
#.Output the direct path found, or “unreachable” if there is none. Output
#   format: TBD.
```

☞ **Specify how individual program steps will cooperate with one another.**

Internal Data

```
#.Input a maze of arbitrary size, or output “malformed input” and stop if the
#   input is improper. Input format: TBD.
```

internal data (maze)

```
#.Compute a direct path through the maze, if one exists.
```

internal data (path)

```
#.Output the direct path found, or “unreachable” if there is none. Output
#   format: TBD.
```

☞ **A program’s internal data representation is central to the code; consider it early.**

External Data

external data (maze)

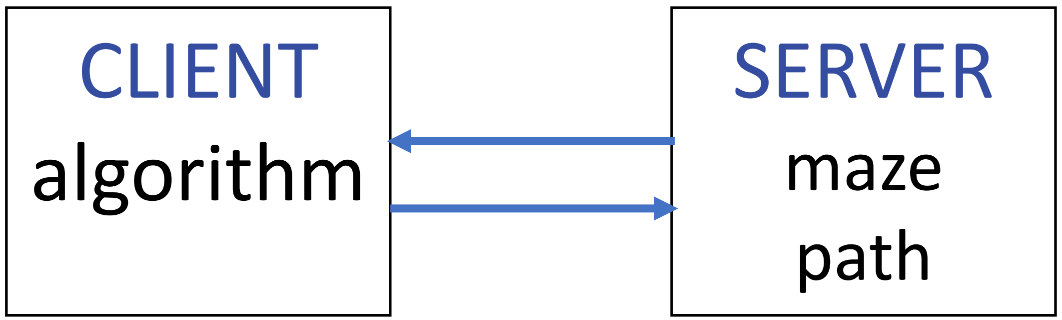
```
#.Input a maze of arbitrary size, or output “malformed input” and stop if the
#   input is improper. Input format: TBD.
#.Compute a direct path through the maze, if one exists.
#.Output the direct path found, or “unreachable” if there is none. Output
   format: TBD.
```

external data (path)

☞ **Consider a program’s external data representation late.**

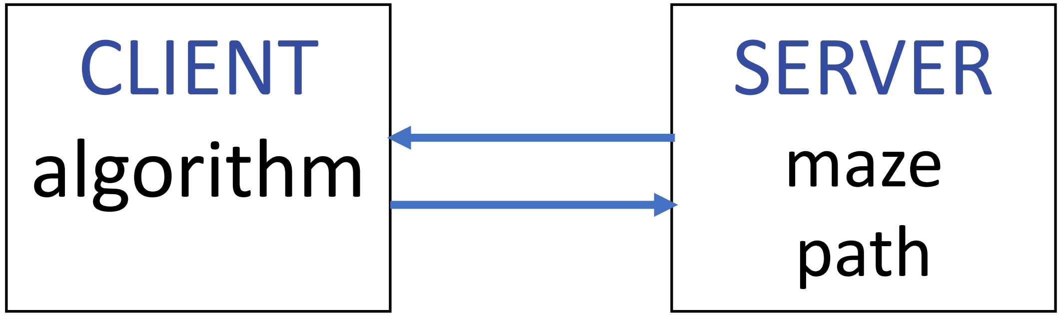
Components

- A program can be organized into components.
- Distinguish between the maze-running algorithm (a client of data) and the data itself (housed in a server).
- What operations are needed by the client?
- What operations can be provided by the server?
- Resolve differences by negotiation.



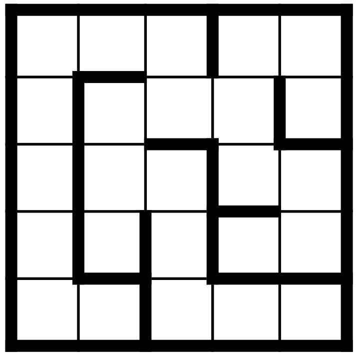
Components

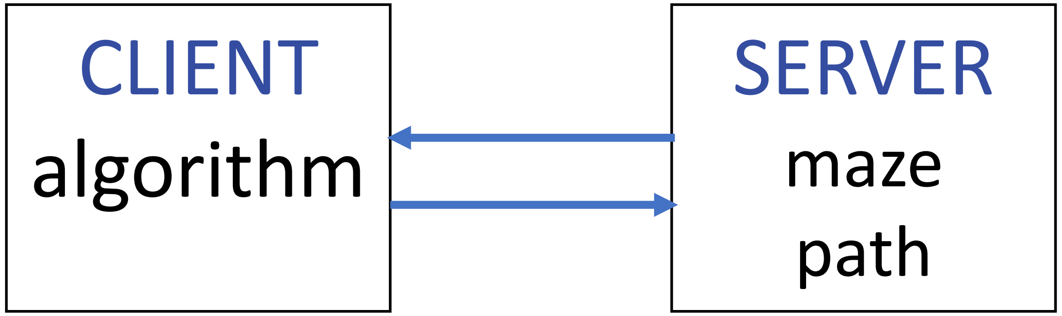
- Some aspects of data are static, i.e., don’t change (maze)
- The client learns of static data by queries.
- Other aspects of data are dynamic, i.e., change (path)
- The client is an actor that effects changes by actions:
  - extend path (if possible); retract path (if necessary)
  - The cumulative effect of actions is recorded in state.



Components

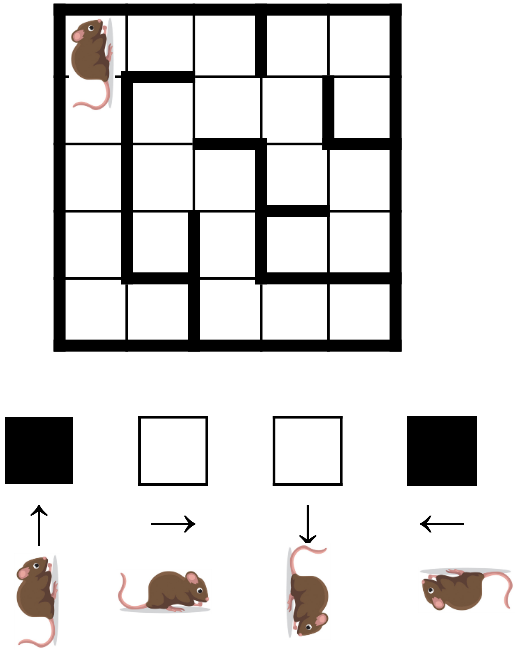
- A client may have/want global perspective
  - algorithm is aware of the full maze

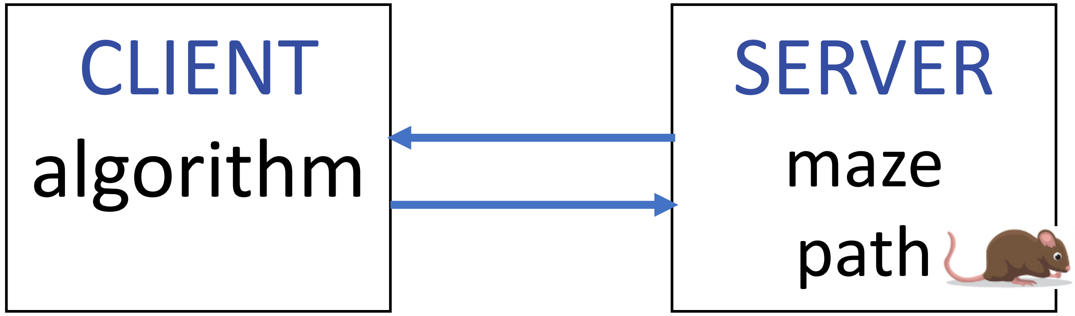




Components

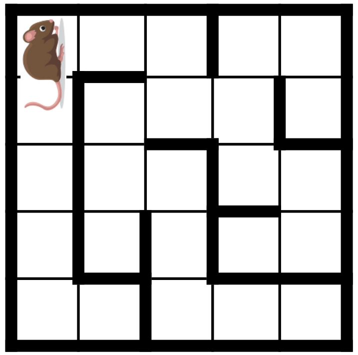
- A client may have/want global perspective
  - algorithm is aware of the full maze
- Other clients have/want only local perspective
  - rat is unaware of full maze

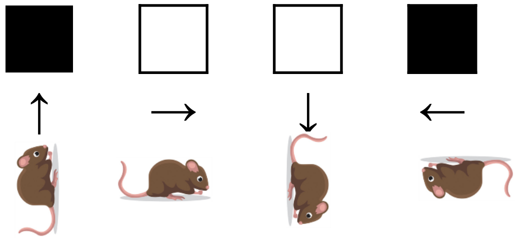


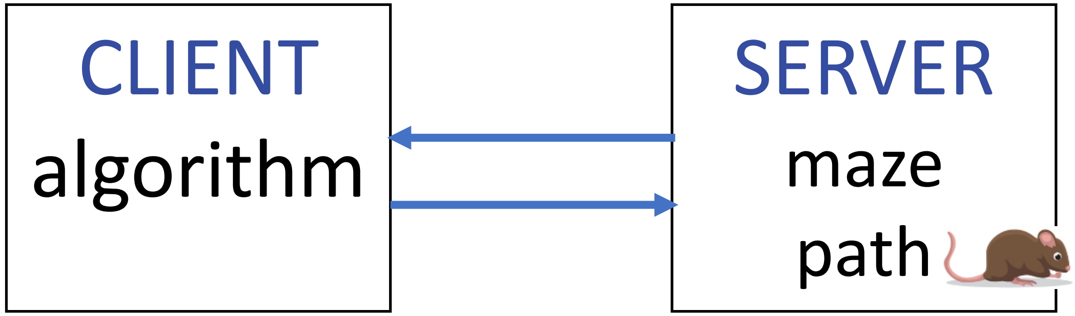


Components

- A client may have/want global perspective
  - algorithm is aware of the full maze
- Other clients have/want only local perspective
  - rat is unaware of full maze







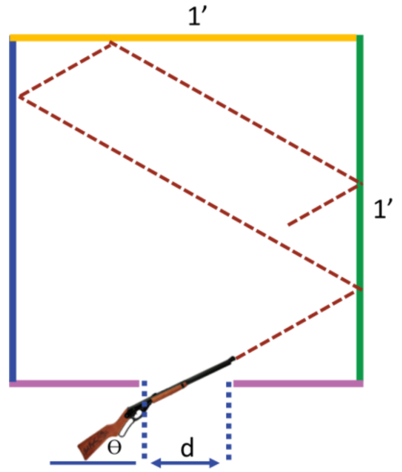
Another programming problem

☞ **Analyze first.**

Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

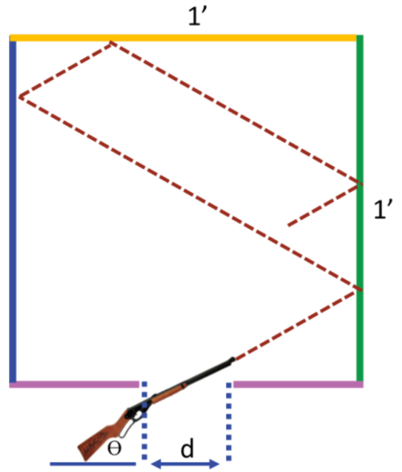
Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.



Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.



☞ **Analyze first.**

An analogous example: Output the sum of the integers between 1 and n.

An analogous example: Output the sum of the integers between 1 and n.

```
#.Output the sum of 1 through n.
```

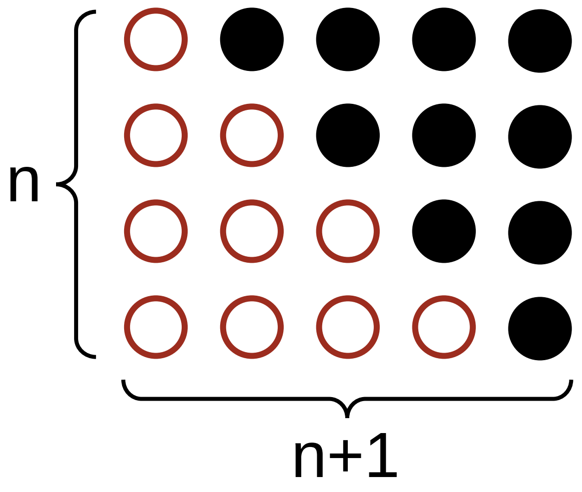
An analogous example: Output the sum of the integers between 1 and n.

```
# Output the sum of 1 through n.
sum = 0
for k in range(1, n + 1):
    sum = sum + k
print(sum)
```

} knee-jerk, brute force

An analogous example: Output the sum of the integers between 1 and n.

```
# Output the sum of 1 through n.
sum = 0
for k in range(1, n + 1):
    sum = sum + k
print(sum)
```

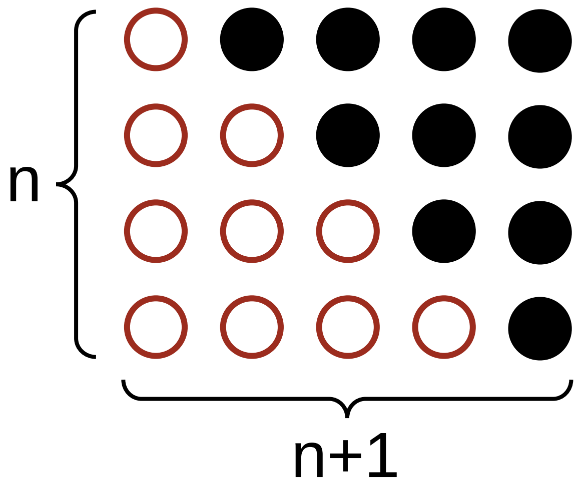


☞ **Analyze first.**

An analogous example: Output the sum of the integers between 1 and n.

```
# Output the sum of 1 through n.
print(n * (n + 1) // 2)
```

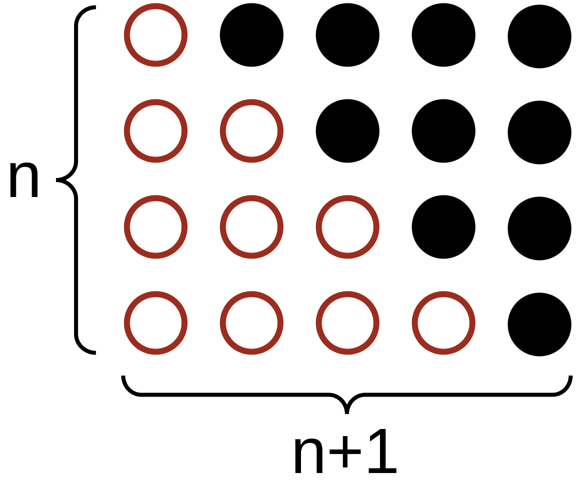
Integer division, with fractional part truncated.



☞ **Analyze first.**

An analogous example: Output the sum of the integers between 1 and n.

```
# Output the sum of 1 through n.
print(n * (n + 1) // 2)
```

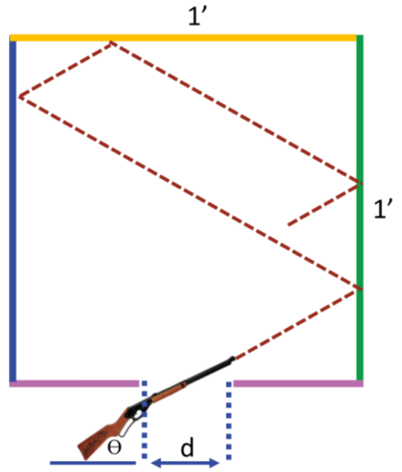


☞ **Sometimes iteration is unnecessary because a closed-form solution is available.**

Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

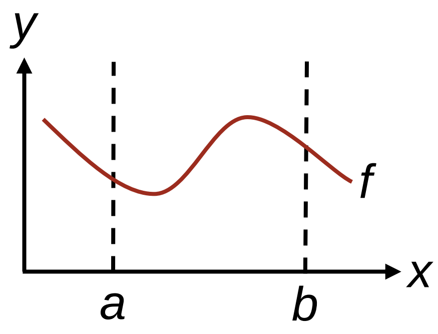
Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.



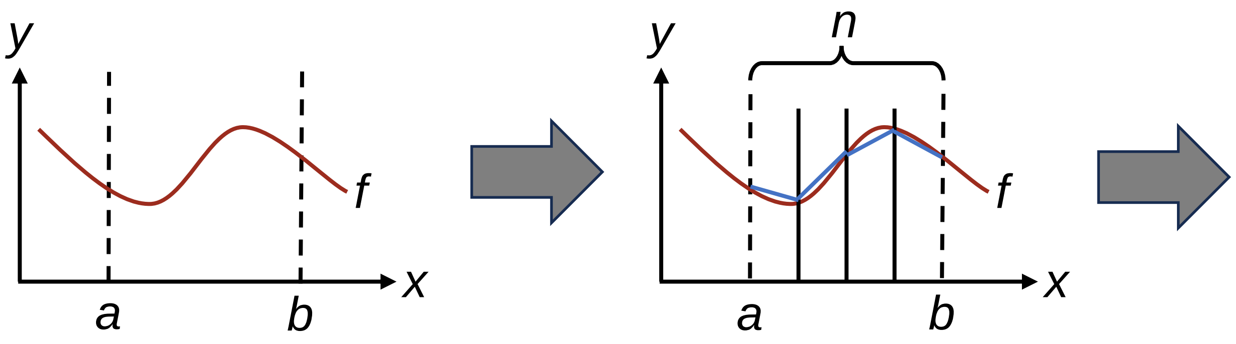
☞ **Sometimes iteration is unnecessary because a closed-form solution is available.**

Another analogy: A possible source of inspiration.

Another analogy: Computing the arc length *s* of a curve y=*f(x),* between a and b.

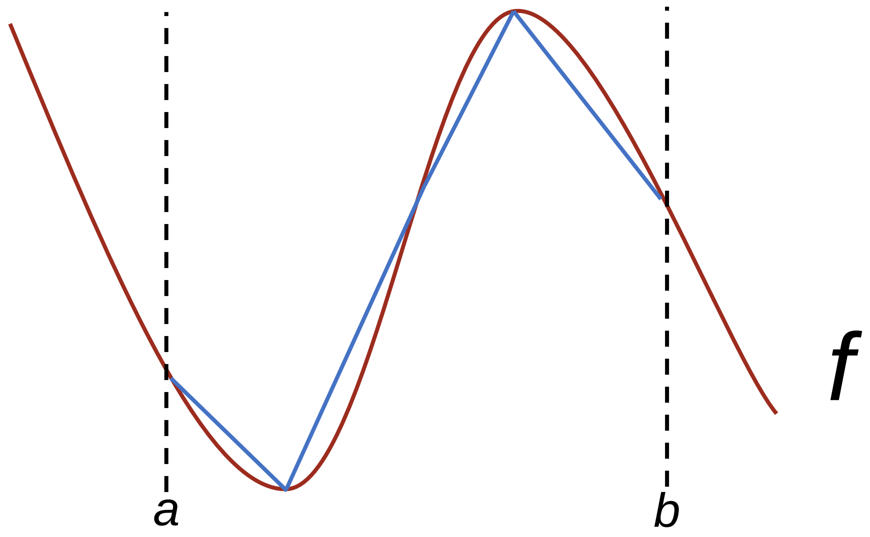


Another analogy: Computing the arc length *s* of a curve y=*f(x),* between a and b.

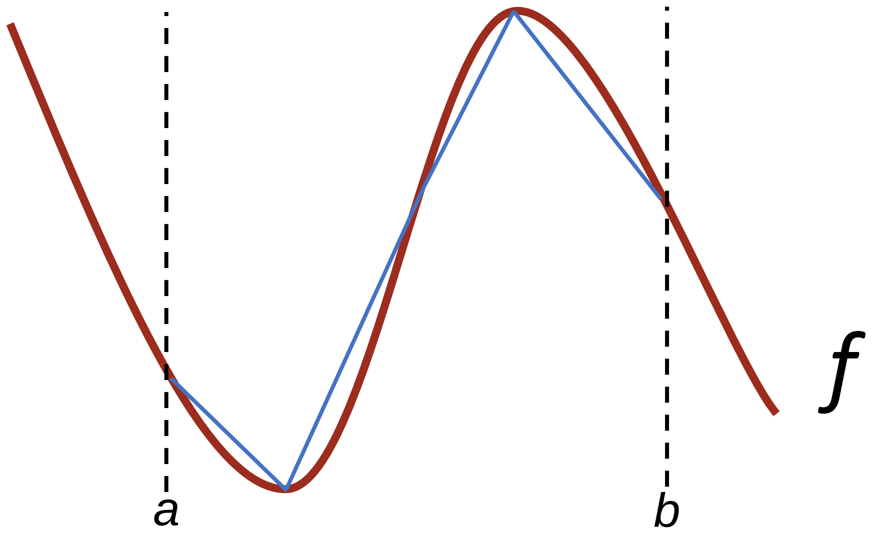


$$s = \int_a^b \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\ dx$$

Another analogy: Where does the analogy faulter?

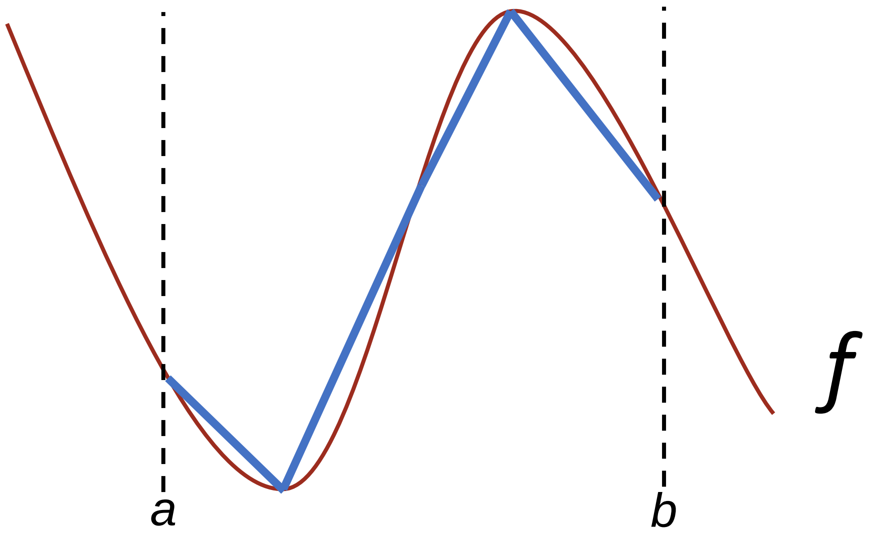


Another analogy: In the calculus problem, we seek s, the length of f.



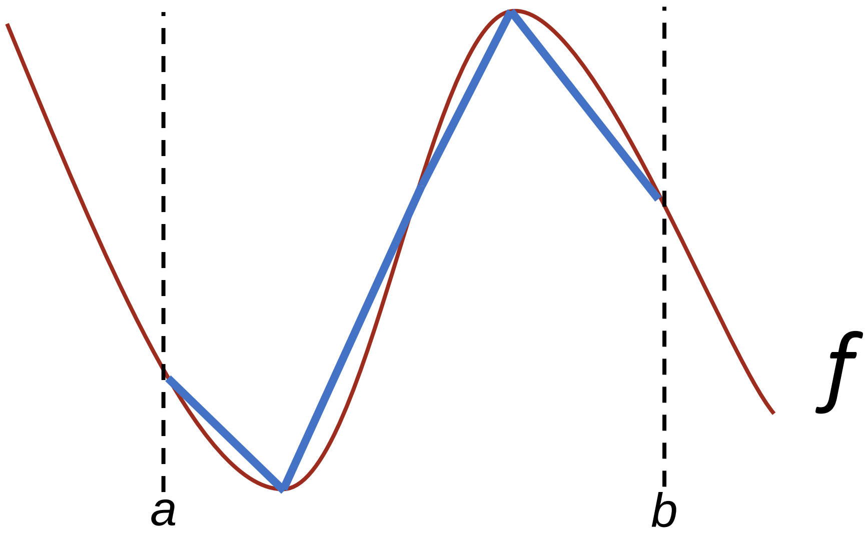
$$s = \int_a^b \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx$$

Another analogy: In the bee-bee problem, we seek the sum of the piece lengths.



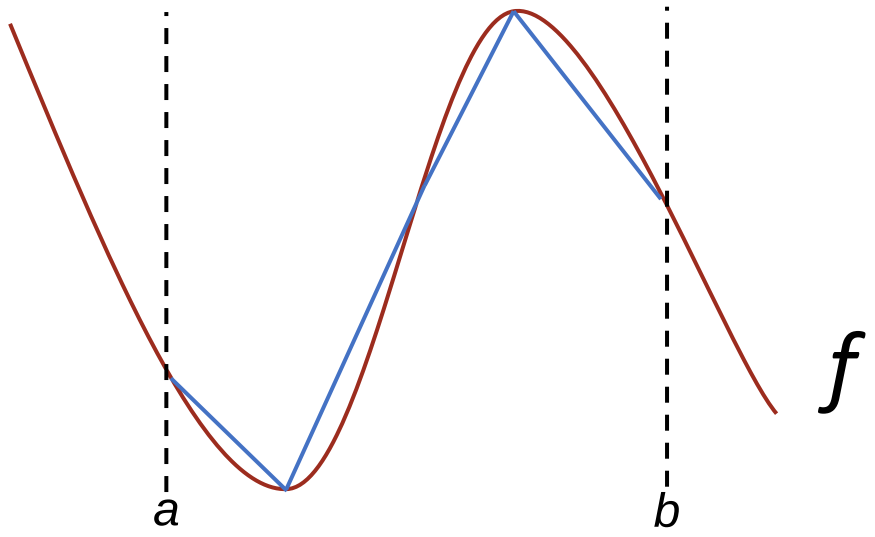

```
# Output the sum of 1 through n.
print(n * (n + 1) // 2)
```

Another analogy: In general, they are only the same in the infinite limit.

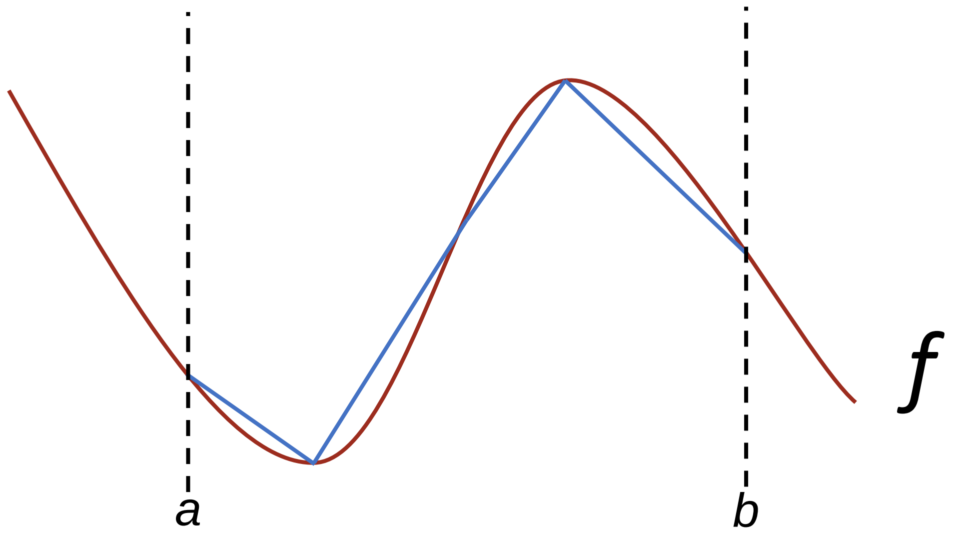



```
# Output the sum of 1 through n.
print(n * (n + 1) // 2)
```

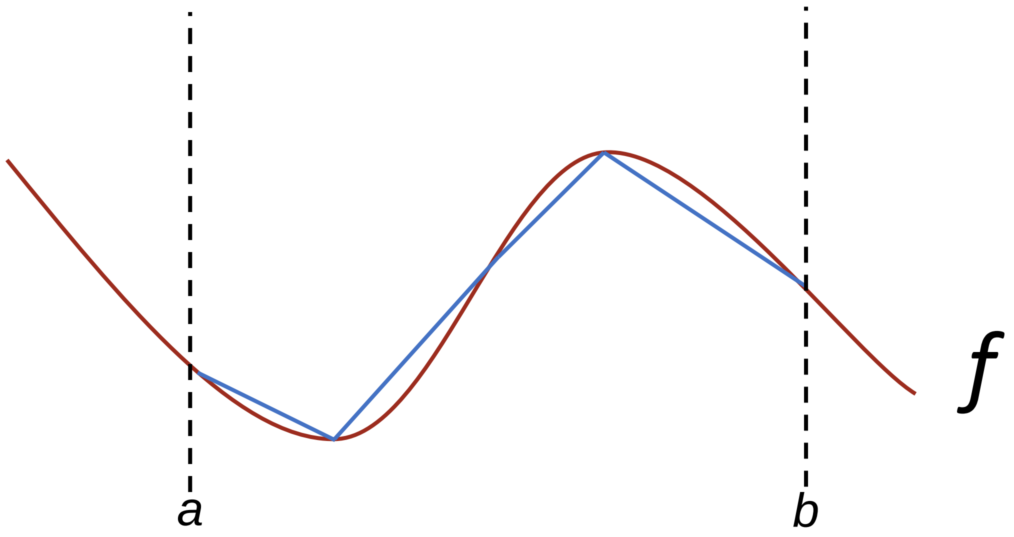
Another analogy: How can we unify the two disparate points of view?



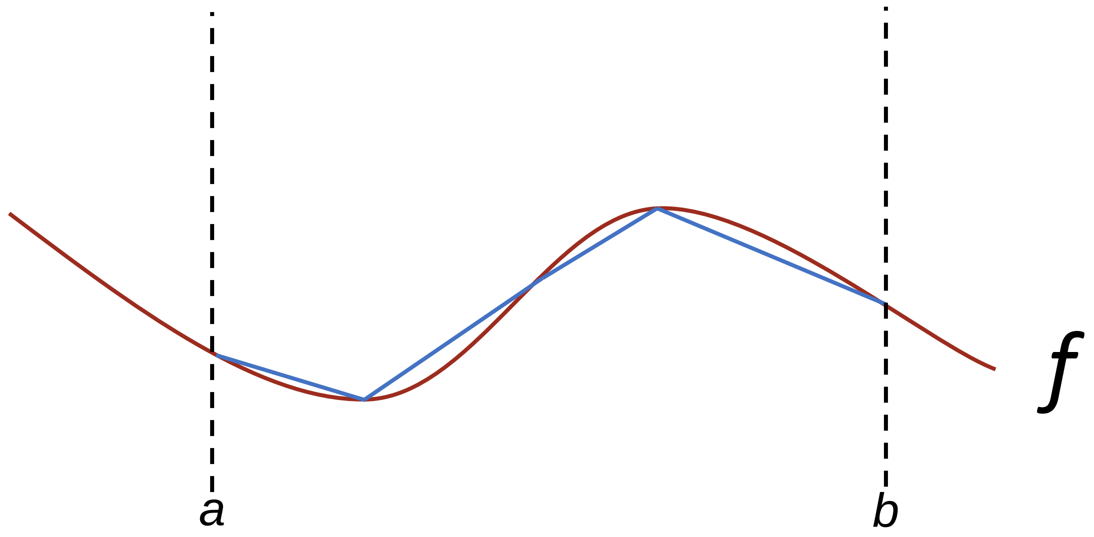
Another analogy: By finding an instance of the problem where they are the same.



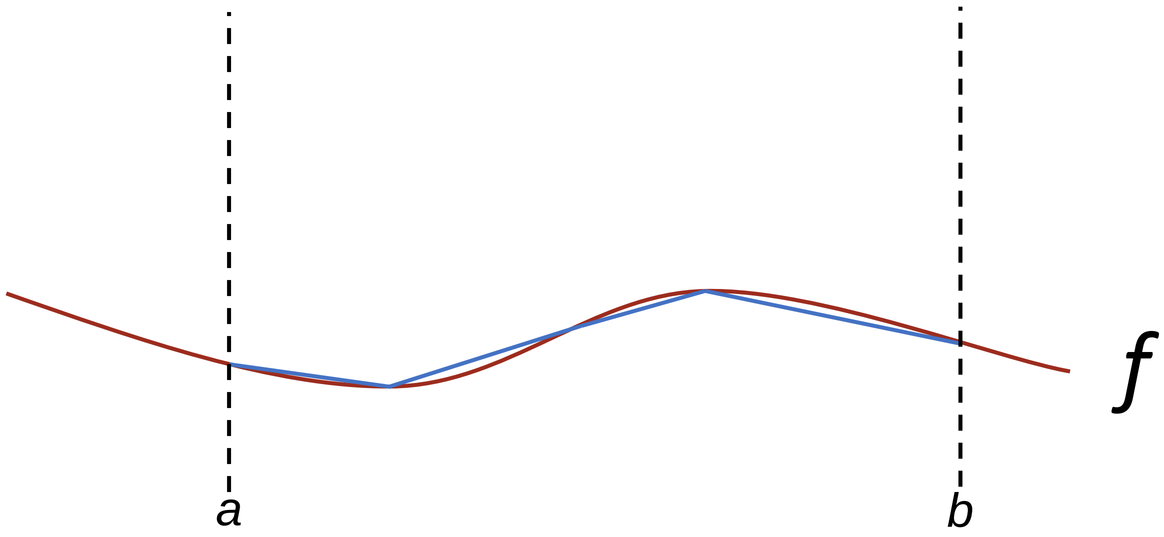
Another analogy: By finding an instance of the problem where they are the same.



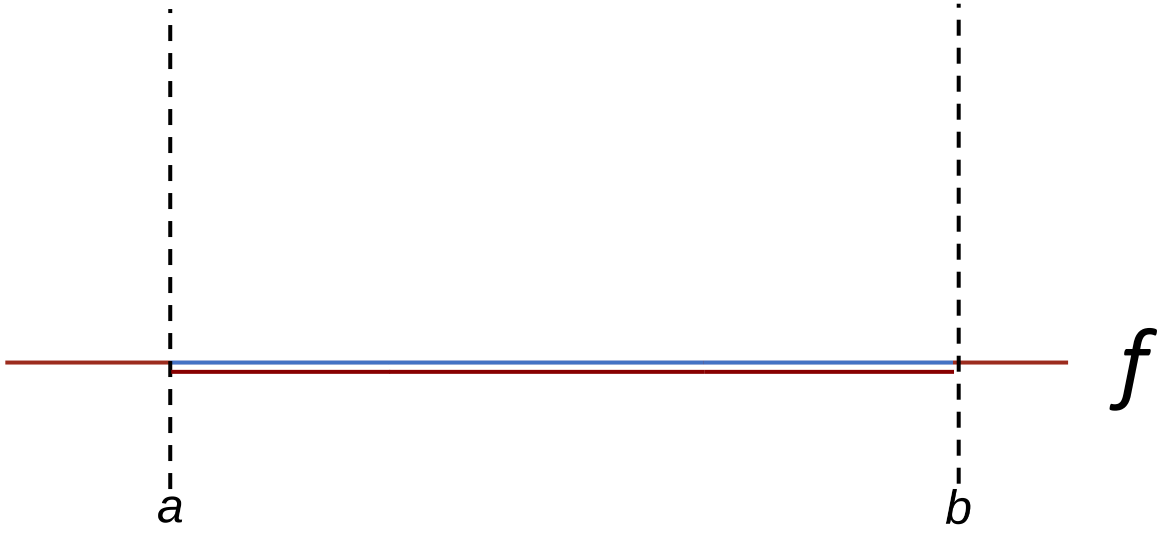
Another analogy: By finding an instance of the problem where they are the same.



Another analogy: By finding an instance of the problem where they are the same.



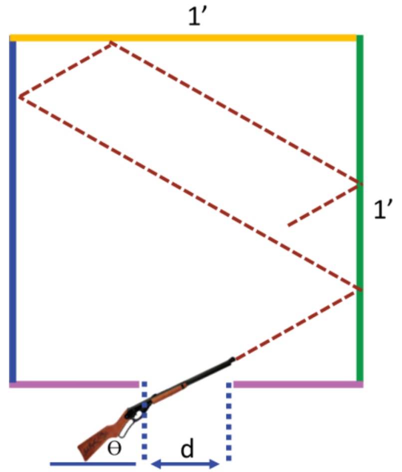
Another analogy: By finding an instance of the problem where they are the same.



Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.

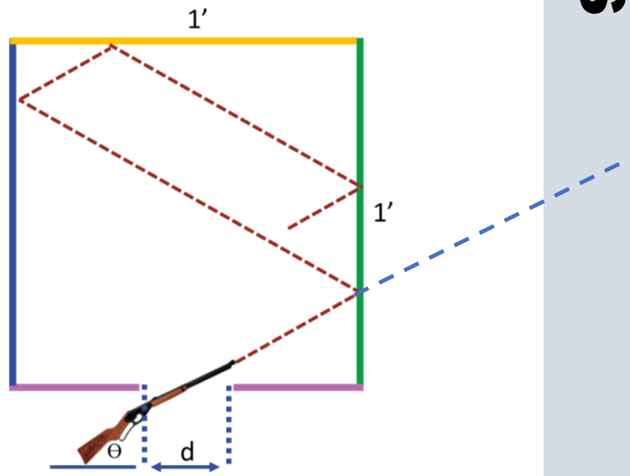


☞ **Solve a different problem, and use that solution to solve the original problem.**

Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.

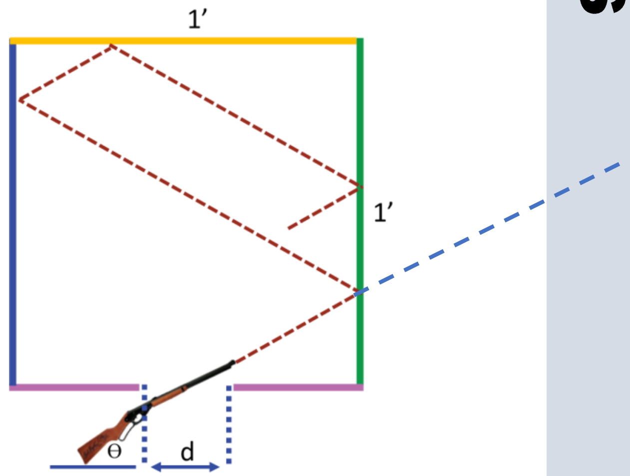


☞ **Solve a different problem, and use that solution to solve the original problem.**

Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.



☞ **Solve a different problem, and use that solution to solve the original problem.**

Analysis

Hippocratic Coding:

☞ **Aspire to code it right the first time.**

Hippocratic Coding:

☞ **Aspire to code it right the first time.**

Patterns:

☞ **Master stylized code patterns, and use them.**

Hippocratic Coding:

☞ **Aspire to code it right the first time.**

Patterns:

☞ **Master stylized code patterns, and use them.**

Analysis:

☞ **Analyze first.**

Hippocratic Coding:

☞ **Aspire to code it right the first time.**

Patterns:

☞ **Master stylized code patterns, and use them.**

Analysis:

☞ **Analyze first.**

Process:

☞ **Mitigate errors.**

Process: Don’t make mistakes

- Hope for the best, but
- Plan for the worst.

☞ **Avoid debugging like the plague.**

Process: Find mistakes as soon as possible

☞ **Test programs incrementally.**

Process: Stay in control

- Define relevant subproblems that can be tested.
- Preserve end-to-end correctness

☞ **Never be (very) lost. Don’t stray far from a correct (albeit, partial) program.**

Process: End-to-end correctness of subproblems

```
#.Input a maze of arbitrary size, or output “malformed input” and stop if the
#   input is improper. Input format: TBD.
#.Compute a direct path through the maze, if one exists.
#.Output the direct path found, or “unreachable” if there is none. Output
#   format: TBD.
```

☞ **Never be (very) lost. Don’t stray far from a correct (albeit, partial) program.**

Process: End-to-end correctness of subproblems

```
#.Input a maze of arbitrary size, or output “malformed input” and stop if the
#   input is improper. Input format: TBD.
#.Compute a direct path through the maze, if one exists.
#.Output the direct path found, or “unreachable” if there is none. Output
#   format: TBD.
```

☞ **Never be (very) lost. Don’t stray far from a correct (albeit, partial) program.**

Process: End-to-end correctness of subproblems

jury rig a specific maze

```
#.Compute a direct path through the maze, if one exists.
```

provide simple diagnostic output

☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**

Process: End-to-end correctness of subproblems

jury rig a specific maze

#.Compute a ~~direct~~ path through the maze, if one exists.

provide simple diagnostic output

☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**

Process: Undo if necessary

☞ **Don’t be wedded to code. Revise and rewrite when you discover a better way.**

Process: Stepwise refinement

☞ **Program top-down, outside-in.**

Example: Print the integer part of the square root of an integer `n≥0`.

Example: Print the integer part of the square root of an integer `n≥0`.

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

```
#.Given n≥0, output the Integer Square Root of n.                1
```

Example: Print the integer part of the square root of an integer `n≥0`.

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

Example

#.Given n≥0, output the Integer Square Root of n.	1

Example

#.Given n≥0, output the Integer Square Root of n.	1

☞ **Make sure you understand the problem.**

Example

```
#.Given n≥0, output the Integer Square Root of n.
```
1

Q. Where did n come from?

☞ **Make sure you understand the problem.**

```
#.Given n≥0, output the Integer Square Root of n.
```

1

Q. Where did n come from?

- A1. It is a program variable.

☞ **Make sure you understand the problem.**

#.Given n≥0, output the Integer Square Root of n.	1

Q. Where did n come from?

- A1. It is a program variable.
- A2. It is assumed to already contain a value ≥ 0.

☞ **Make sure you understand the problem.**

```
#.Given n≥0, output the Integer Square Root of n.                    1
```

Q. Where did n come from?

- A1. It is a program variable.
- A2. It is assumed to already contain a value ≥ 0.
- A3. We are only asked to write a program segment.

☞ **Make sure you understand the problem.**

#.Given n≥0, output the Integer Square Root of n. 1

☞ **Make sure you understand the problem.**

```
#.Given n≥0, output the Integer Square Root of n.
```

1

Q. Can’t we just do this using a few library routines?

☞ **Make sure you understand the problem.**

```
# Given n≥0, output the Integer Square Root of n.
print(math.floor(math.sqrt(n)))
```

2

Q. Can't we just do this using a few library routines?

- A. Yes.

☞ **Make sure you understand the problem.**

```
# Given n≥0, output the Integer Square Root of n.
print(math.floor(math.sqrt(n)))
```

2

Q. Can’t we just do this using a few library routines?

- A. Yes.
- But that would deprive us of a good example.
- So, we amend our problem statement.

☞ **Make sure you understand the problem.**

```
# Given n≥0, output the Integer Square Root of n.
print(math.floor(math.sqrt(n)))
```

2

Example: Print the integer part of the square root of an integer n≥0 without using built-in functions.

Q. Can't we just do this using a few library routines?

- A. Yes.
- But that would deprive us of a good example.
- So, we amend our problem statement.

☞ **Make sure you understand the problem.**

```
#.Given n≥0, output the Integer Square Root of n.
```
1

Example: Print the integer part of the square root of an integer n≥0 without using built-in functions.

```
#.Given n≥0, output the Integer Square Root of n.
```

1

☞ **Master stylized code patterns, and use them.**

```
#.Given n≥0, output the Integer Square Root of n.
```

1

☞ **Master stylized code patterns, and use them.**

Example

```
#.Given n≥0, output the Integer Square Root of n.
```

1

```
#.Compute.
#.Use.
```

☞ **Master stylized code patterns, and use them.**

Example

```
#.Given n≥0, output the Integer Square Root of n.
```
1

```
#.Compute r.
#.Use r.
```

☞ **Specify how individual program steps will cooperate with one another.**

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------
#.Let r be the integer part of the square root of n≥0.
print(r)
```

2

```
# Compute r.
# Use r.
```

☞ **Specify how individual program steps will cooperate with one another.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
#.Let r be the integer part of the square root of n≥0.
print(r)
```

2

- Recall the *compute-use pattern* underlying this code, and notice how the dot (`.`) in:

  `#.Let r be ...`

  helps you to read the line as an as-yet *unimplemented specification*, not as the *specification* of the line that follows it.

Example

```
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
#.Let r be the integer part of the square root of n≥0.
print(r)
```

2

- Recall the *compute-use pattern* underlying this code, and notice how the dot (`.`) in:

  `#.Let r be ...`

  helps you to read the line as an as-yet *unimplemented specification*, not as the *specification* of the line that follows it.
- The line of dashes (`-`) separates the top-level *specification* from the two subordinate steps that *implement* it.

Example

```
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
#.Let r be the integer part of the square root of n≥0.
print(r)
```

2

Example

```
# Given n≥0, output the Integer Square Root of n.
# -----------------------------------------------
#.Let r be the integer part of the square root of n≥0.
print(r)
```

2

☞ **Master stylized code patterns, and use them.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
#.Let r be the integer part of the square root of n≥0.
print(r)
```

2

☞ **If you “smell a loop”, write it down.**

```
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
#.Let r be the integer part of the square root of n≥0.
print(r)
```

2

Example

☞ **Decide first whether an iteration is indeterminate (use while) or determinate (use for).**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
#.Let r be the integer part of the square root of n≥0.
print(r)
```
2

```
# Indeterminate iteration
k = start
while condition:
    k += 1
```

Which?

```
# Determinate iteration
for k in range(expression_1, expression_2):
    compute
```

☞ **Decide first whether an iteration is indeterminate (use while) or determinate (use for).**

Example

```
# Given n≥0, output the Integer Square Root of n.
# -------------------------------------------
#.Let r be the integer part of the square root of n≥0.
print(r)
```

2

```
# Indeterminate iteration
k = start
while condition:
    k += 1
```

Which?

```
# Determinate iteration
for k in range(expression1, expression2):
    compute
```

☞ **Beware of for-loop abuse; if in doubt, err in favor of while.**

```
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
#.Let r be the integer part of the square root of n≥0.
print(r)
```

2

```
# Indeterminate iteration
k = start
while condition:
    k += 1
```

~~# Determinate iteration
for k in range($expression_1$, $expression_2$):
    compute~~

☞ **Beware of for-loop abuse; if in doubt, err in favor of while.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition:
    r += 1

print(r)
```

3

```
# Indeterminate iteration
k = start
while condition:
    k += 1
```

☞ **Beware of for-loop abuse; if in doubt, err in favor of while.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition:
    r += 1

print(r)
```

3

- With the dot (.) now removed from:
  `#.Let r be ...`
  the comment now reads as the *specification* of the *implementation* lines that follow it.

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition:
    r += 1

print(r)
```

3

- With the dot (.) now removed from:

  `#.Let r be ...`

  the comment now reads as the *specification* of the implementation lines that follow it.
- A blank line has been introduced to separate the separate steps from one another.

Example

```
# Given n≥0, output the Integer Square Root of n.
# ------------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```

3

Example

3

```
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```

☞ **There is no shame in reasoning with concrete examples.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# -----------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```

3

☞ **Elaborate the expected input/output mapping explicitly.**

```
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```

3

r	r*r	n
0	0	0

☞ **Elaborate the expected input/output mapping explicitly.**

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```

3

r	r*r	n
0	0	0
1	1	1, 2, 3

☞ **Elaborate the expected input/output mapping explicitly.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```

3

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8

☞ **Elaborate the expected input/output mapping explicitly.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```

3

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

☞ **Elaborate the expected input/output mapping explicitly.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```

3

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?

- 4, 5, 6, 7, or 8.

☞ **Elaborate the expected input/output mapping explicitly.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```

3

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?

- 4, 5, 6, 7, or 8.

When r=2, for which n do we **continue**?

- 9, 10, 11, …

☞ **Elaborate the expected input/output mapping explicitly.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```

```
3
```

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?

- 4, 5, 6, 7, or 8.

When r=2, for which n do we **continue**?

- 9, 10, 11, …

What is special about 9?

- It is the square of 3.

☞ **Elaborate the expected input/output mapping explicitly.**

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```

3

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?

- 4, 5, 6, 7, or 8.

When r=2, for which n do we **continue**?

- 9, 10, 11, …

What is special about 9?

- It is the square of 3.

But what is special about 3?

- It is one more than 2, the value of r.

☞ **Elaborate the expected input/output mapping explicitly.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while condition :
    r += 1

print(r)
```
3

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?

- 4, 5, 6, 7, or 8.

When r=2, for which n do we **continue**?

- 9, 10, 11, …

What is special about 9?

- It is the square of 3.

But what is special about 3?

- It is one more than 2, the value of r.

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) ____ n :
    r += 1

print(r)
```
4

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?

- 4, 5, 6, 7, or 8.

When r=2, for which n do we **continue**?

- 9, 10, 11, …

What is special about 9?

- It is the square of 3.

But what is special about 3?

- It is one more than 2, the value of r.

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) ____ n :
    r += 1

print(r)
```

4

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) [____] n :
    r += 1

print(r)
```

4

Example

```
# Given n≥0, output the Integer Square Root of n.
# -----------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) ____ n :
    r += 1

print(r)
```

4

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) ____ n :
    r += 1

print(r)
```

4

Elaborate and eliminate choices for the relation

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) ____ n :
    r += 1

print(r)
```

4

Elaborate and eliminate choices for the relation

==, != No. Given r, must be true for many n, and false for many n.

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) ____ n :
    r += 1

print(r)
```

4

Elaborate and eliminate choices for the relation

==, !=	No. Given r, must be true for many n, and false for many n.
>, >=	No. Must keep going for little r and big n.

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) ____ n :
    r += 1

print(r)
```

4

Elaborate and eliminate choices for the relation

- ==, != No. Given r, must be true for many n, and false for many n.
- >, >= No. Must keep going for little r and big n.
- < No. Must keep going for "equal n" case

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) ____ n :
    r += 1

print(r)
```

4

Elaborate and eliminate choices for the relation

==, !=	No. Given r, must be true for many n, and false for many n.
>, >=	No. Must keep going for little r and big n.
<	No. Must keep going for “equal n” case.
<=	Yes.

☞ **Alternate between concrete reasoning and abstract reasoning.**

```
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) <= n :
    r += 1

print(r)
```


Elaborate and eliminate choices for the relation

==, !=	No. Given r, must be true for many n, and false for many n.
>, >=	No. Must keep going for little r and big n.
<	No. Must keep going for “equal n” case.
<=	Yes.

☞ **Alternate between concrete reasoning and abstract reasoning.**

```
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) <= n :
    r += 1

print(r)
```

5

Pragmatics

We developed a code fragment in isolation, and ignored several practical questions:

A. Where will the integer n come from?
B. In what packaging will we run the code fragment?
C. What, if any, additional details must be addressed before the program can run?

We refer to these matters as “Pragmatics”.

```
#.Obtain an integer n≥0 from the user.
# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) <= n:
    r += 1

print(r)
```

6

Pragmatics

We developed a code fragment in isolation, and ignored several practical questions:

A. Where will the integer n come from?
   Obtain the integer value of n interactively from the user.

```
#.Obtain an integer n≥0 from the user.
# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) <= n:
    r += 1

print(r)
```

6

Pragmatics

We developed a code fragment in isolation, and ignored several practical questions:

A. Where will the integer n come from?
Obtain the integer value of n interactively from the user.

```
# Obtain an integer n≥0 from the user.
n = int(input("Enter integer:"))

# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) <= n:
    r += 1

print(r)
```

7

Pragmatics

We developed a code fragment in isolation, and ignored several practical questions:

A. Where will the integer n come from?
   Obtain the integer value of n interactively from the user in response to a prompt.

4

```
# Obtain an integer n≥0 from the user.
n = int(input("Enter integer:"))

# Given n≥0, output the Integer Square Root of n.
# ---------------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) <= n:
    r += 1

print(r)
```

8

Pragmatics

We developed a code fragment in isolation, and ignored several practical questions:

B. In what context will we run the finished code?
   As the body of a method named `main`.

```
# Obtain an integer n≥0 from the user.
n = int(input("Enter integer:"))

# Given n≥0, output the Integer Square Root of n.
# ----------------------------------------------
# Let r be the integer part of the square root of n≥0.
r = 0
while (r + 1) * (r + 1) <= n:
    r += 1

print(r)
```

4

8

Pragmatics

We developed a code fragment in isolation, and ignored several practical questions:

B. In what context will we run the finished code?
As the body of a method named `main`.

8

```
def main():
    # Obtain an integer n≥0 from the user.
    n = int(input("Enter integer:"))

    # Given n≥0, output the Integer Square Root of n.
    # -----------------------------------------------
    # Let r be the integer part of the square root of n≥0.
    r = 0
    while (r + 1) * (r + 1) <= n:
        r += 1

    print(r)
```

Pragmatics

We developed a code fragment in isolation, and ignored several practical questions:

B. In what context will we run the finished code?
   As the body of a method named `main`.

9

```python
def main():
    """Output the Integer Square Root of an integer input."""
    # Obtain an integer n≥0 from the user.
    n = int(input("Enter integer:"))

    # Given n≥0, output the Integer Square Root of n.
    # ---------------------------------------------
    # Let r be the integer part of the square root of n≥0.
    r = 0
    while (r + 1) * (r + 1) <= n:
        r += 1

    print(r)
```

Pragmatics

We developed a code fragment in isolation, and ignored several practical questions:

B. In what context will we run the finished code?
As the body of a method named `main`.
The functionality of method `main` is documented in a "docstring". It's like a top-level specification.

```
def main():
    """Output the Integer Square Root of an integer input."""
    # Obtain an integer n≥0 from the user.
    n = int(input("Enter integer:"))

    # Given n≥0, output the Integer Square Root of n.
    # -----------------------------------------------
    # Let r be the integer part of the square root of n≥0.
    r = 0
    while (r + 1) * (r + 1) <= n:
        r += 1

    print(r)
```

9

Pragmatics

We developed a code fragment in isolation, and ignored several practical questions:

C. What, if any, additional details must be addressed before the program can run?
None. The program is ready to run.
The functionality of method `main` is documented in a “docstring”. It’s like a top-level specification.
There are a few optional thing we may choose to do:
- Type annotations
- Static type checking.

```
def main():
    """Output the Integer Square Root of an integer input."""
    # Obtain an integer n≥0 from the user.
    n = int(input("Enter integer:"))

    # Given n≥0, output the Integer Square Root of n.
    # ---------------------------------------------
    # Let r be the integer part of the square root of n≥0.
    r = 0
    while (r + 1) * (r + 1) <= n:
        r += 1

    print(r)
```


Pragmatics

- Type annotations:
  - In Python, you don't need to explicitly declare variables like you do in some other languages (e.g., Java, C++). Instead, you create a variable simply by assigning a value to it.

```python
def main() -> None:
    """Output the Integer Square Root of an integer input."""
    # Obtain an integer n≥0 from the user.
    n: int = int(input("Enter integer:"))

    # Given n≥0, output the Integer Square Root of n.
    # ----------------------------------------------
    # Let r be the integer part of the square root of n≥0.
    r: int = 0
    while (r + 1) * (r + 1) <= n:
        r += 1

    print(r)
```

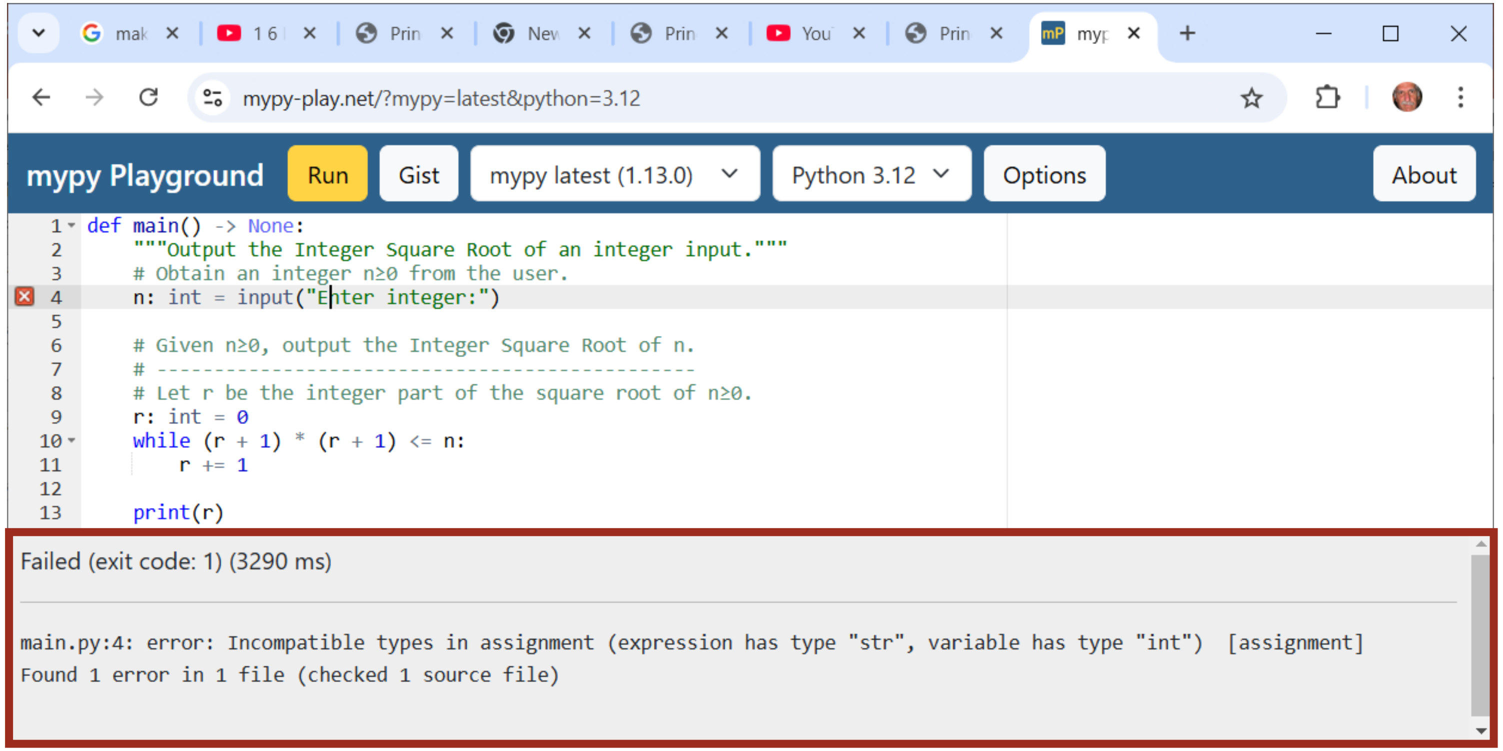
Pragmatics

- Type annotations:
  - In Python, you don't need to explicitly declare variables like you do in some other languages (e.g., Java, C++). Instead, you create a variable simply by assigning a value to it, as above.
  - An optional *type-annotation* associated with the lexically earliest assignment to a variable is recommended. Such an assignment can be considered the variable's *declaration*.
  - Optional *type-annotations* for method parameters and return types are also recommended.

```
def main() -> None:
    """Output the Integer Square Root of an integer input."""
    # Obtain an integer n≥0 from the user.
    n: int = input("Enter integer:")

    # Given n≥0, output the Integer Square Root of n.
    # -----------------------------------------------
    # Let r be the integer part of the square root of n≥0.
    r: int = 0
    while (r + 1) * (r + 1) <= n:
        r += 1

    print(r)
```


Pragmatics

- Static type checking
  - Although some annotations seem superfluous, redundancy can be a boon, e.g., suppose in the assignment to `n`, we forgot the `int(…)` that turns an `Str` into an **`int`**.



Pragmatics

- Static type checking
  - Although some annotations seem superfluous, redundancy can be a boon, e.g., suppose in the assignment to n, we forgot the `int(…)` that turns a `str` into an **`int`**.
  - The tool **mypy** detects such errors *before* program execution, and can save you much grief.

Pragmatics

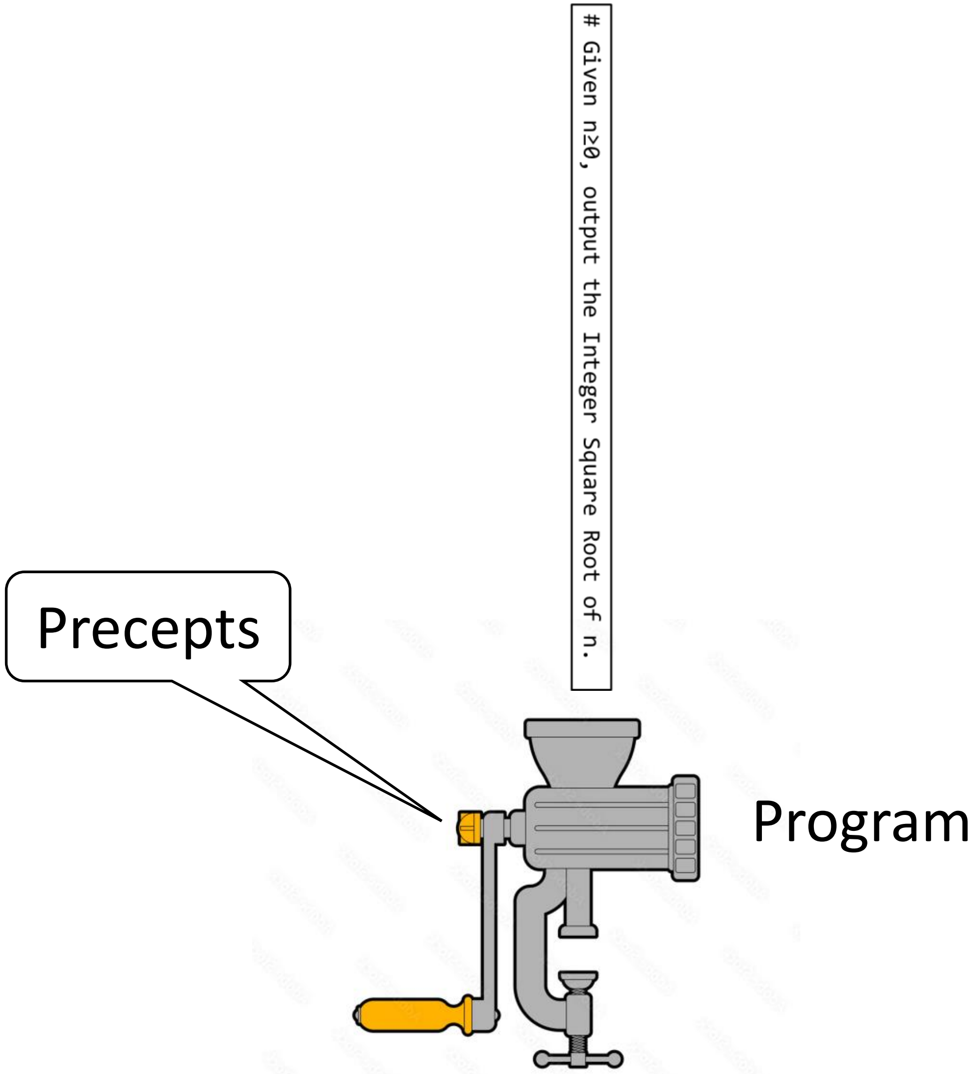
Pragmatics

```
# Given n≥0, output the Integer Square Root of n.
```

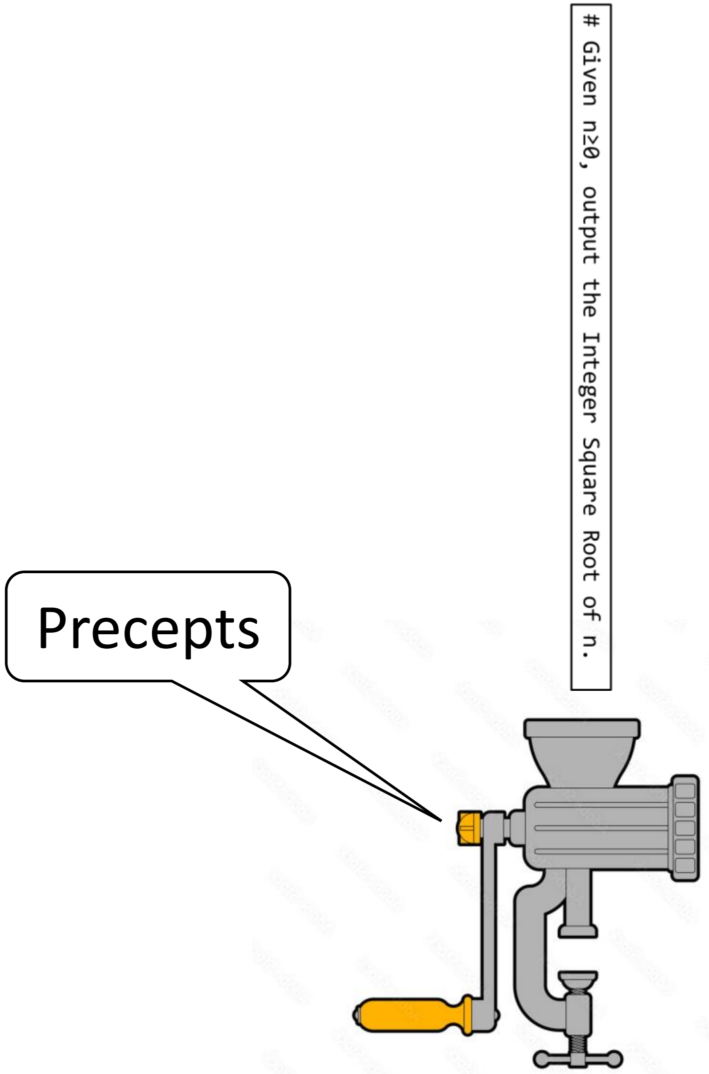
YOU

Program

Pragmatics


Precepts
Given n≥0, output the Integer Square Root of n.
Program

Pragmatics



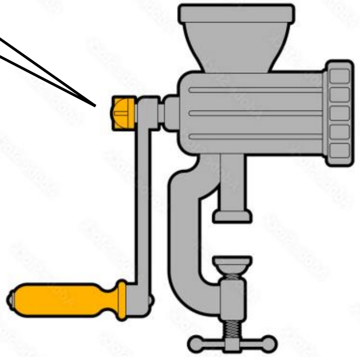

```python
def main() -> None:
    """Output the Integer Square Root of an integer input."""
    # Obtain an integer n≥0 from the user.
    n: int = input("Enter integer:")

    # Given n≥0, output the Integer Square Root of n.
    # -----------------------------------------------
    # Let r be the integer part of the square root of n≥0.
    r: int = 0
    while (r + 1) * (r + 1) <= n:
        r += 1

    print(r)
```

9

Pragmatics

Precepts

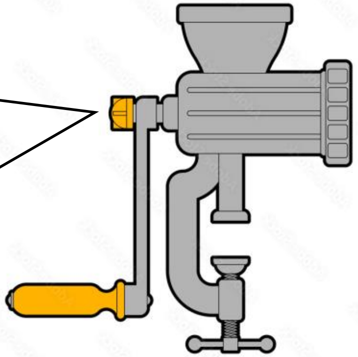
```
# Given n≥0, output the Integer Square Root of n.
```



```
def main() -> None:
    """Output the Integer Square Root of an integer input."""
    # Obtain an integer n≥0 from the user.
    n: int = input("Enter integer:")

    # Given n≥0, output the Integer Square Root of n.
    # -----------------------------------------------
    # Let r be the integer part of the square root of n≥0.
    r: int = 0
    while (r + 1) * (r + 1) <= n:
        r += 1

    print(r)
```

9

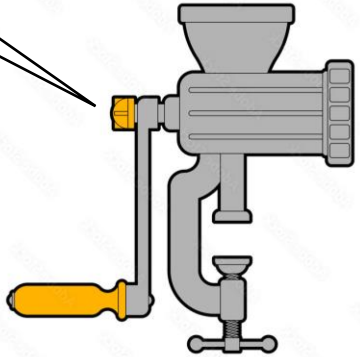
Input



Output

Pragmatics

Precepts

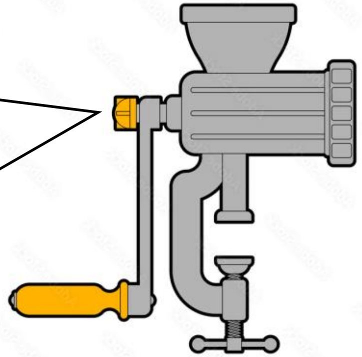


Given n≥0, output the Integer Square Root of n.

```
def main() -> None:
    """Output the Integer Square Root of an integer input."""
    # Obtain an integer n≥0 from the user.
    n: int = input("Enter integer:")

    # Given n≥0, output the Integer Square Root of n.
    # ------------------------------------------------
    # Let r be the integer part of the square root of n≥0.
    r: int = 0
    while (r + 1) * (r + 1) <= n:
        r += 1

    print(r)
```

9

7

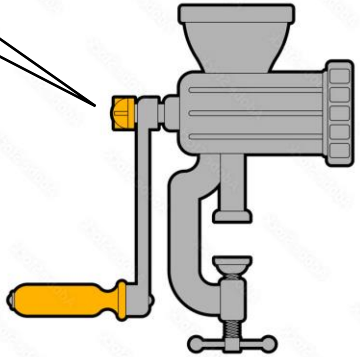


Output

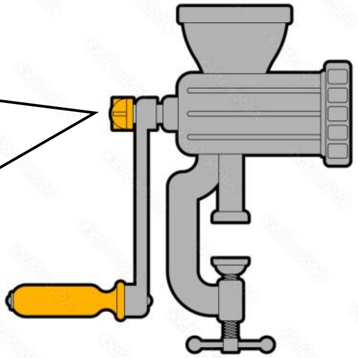
Pragmatics

Precepts

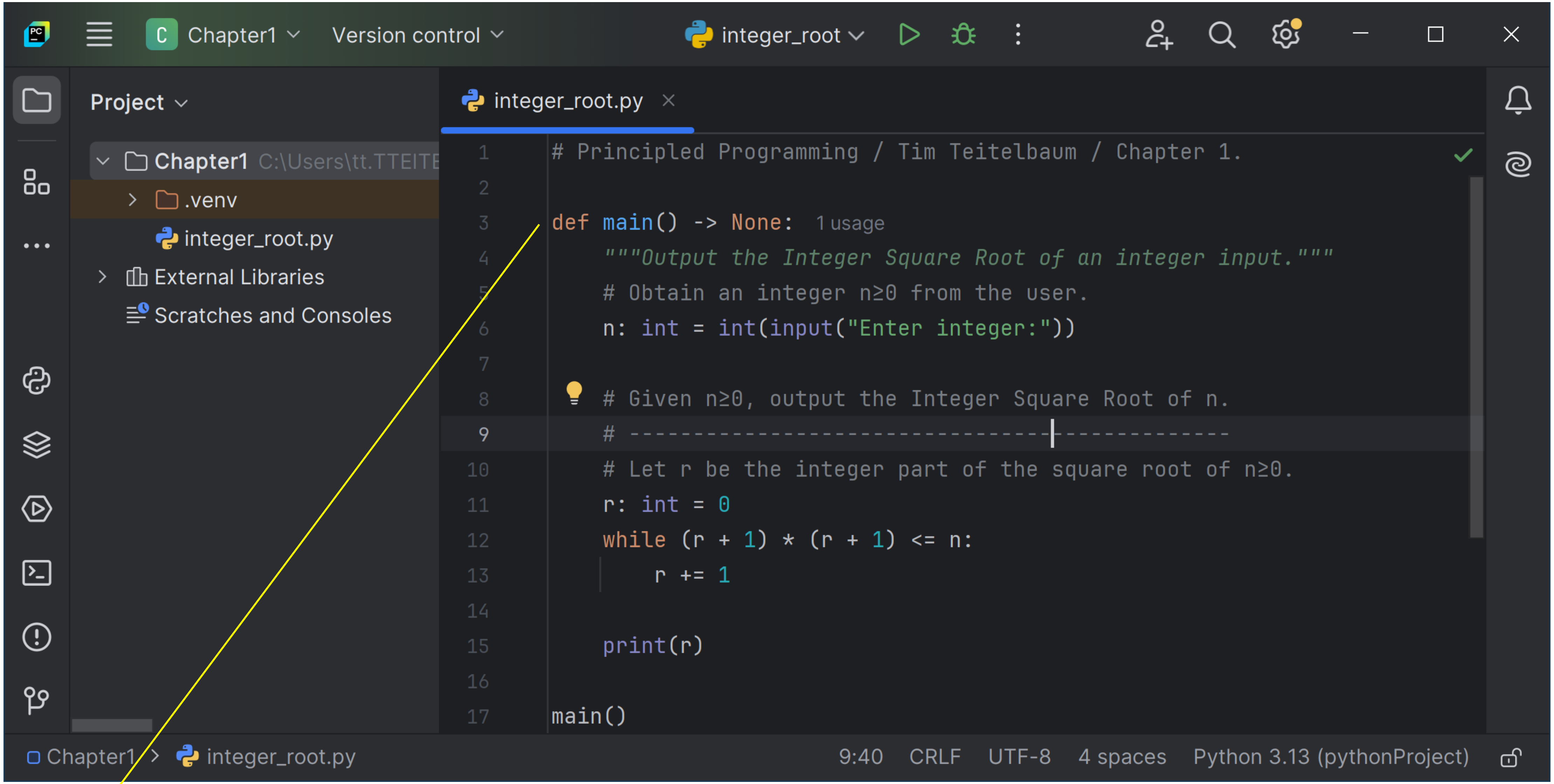
Given n≥0, output the Integer Square Root of n.



```
def main() -> None:
    """Output the Integer Square Root of an integer input."""
    # Obtain an integer n≥0 from the user.
    n: int = input("Enter integer:")

    # Given n≥0, output the Integer Square Root of n.
    # -----------------------------------------------
    # Let r be the integer part of the square root of n≥0.
    r: int = 0
    while (r + 1) * (r + 1) <= n:
        r += 1

    print(r)
```

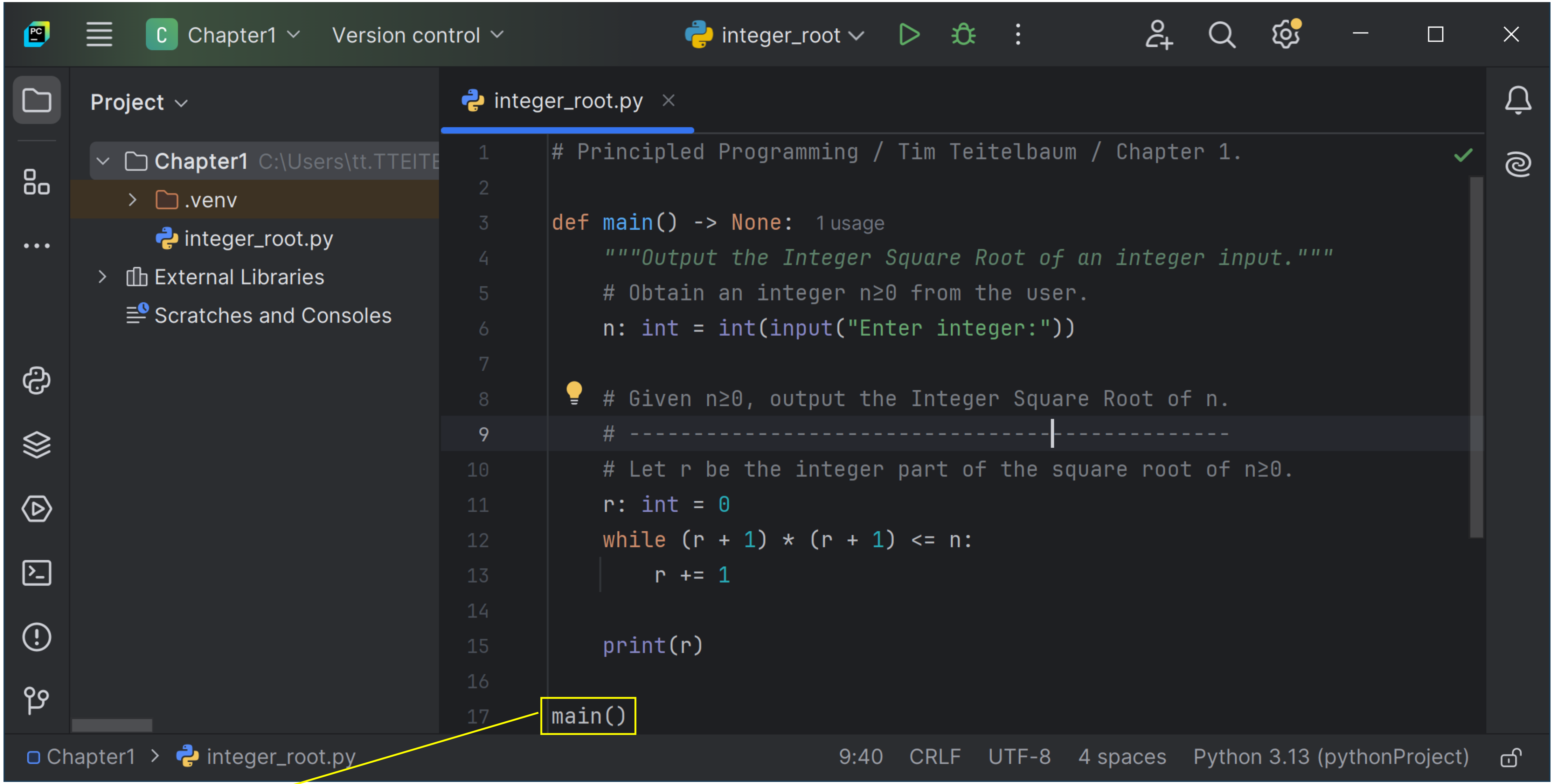


Pragmatics

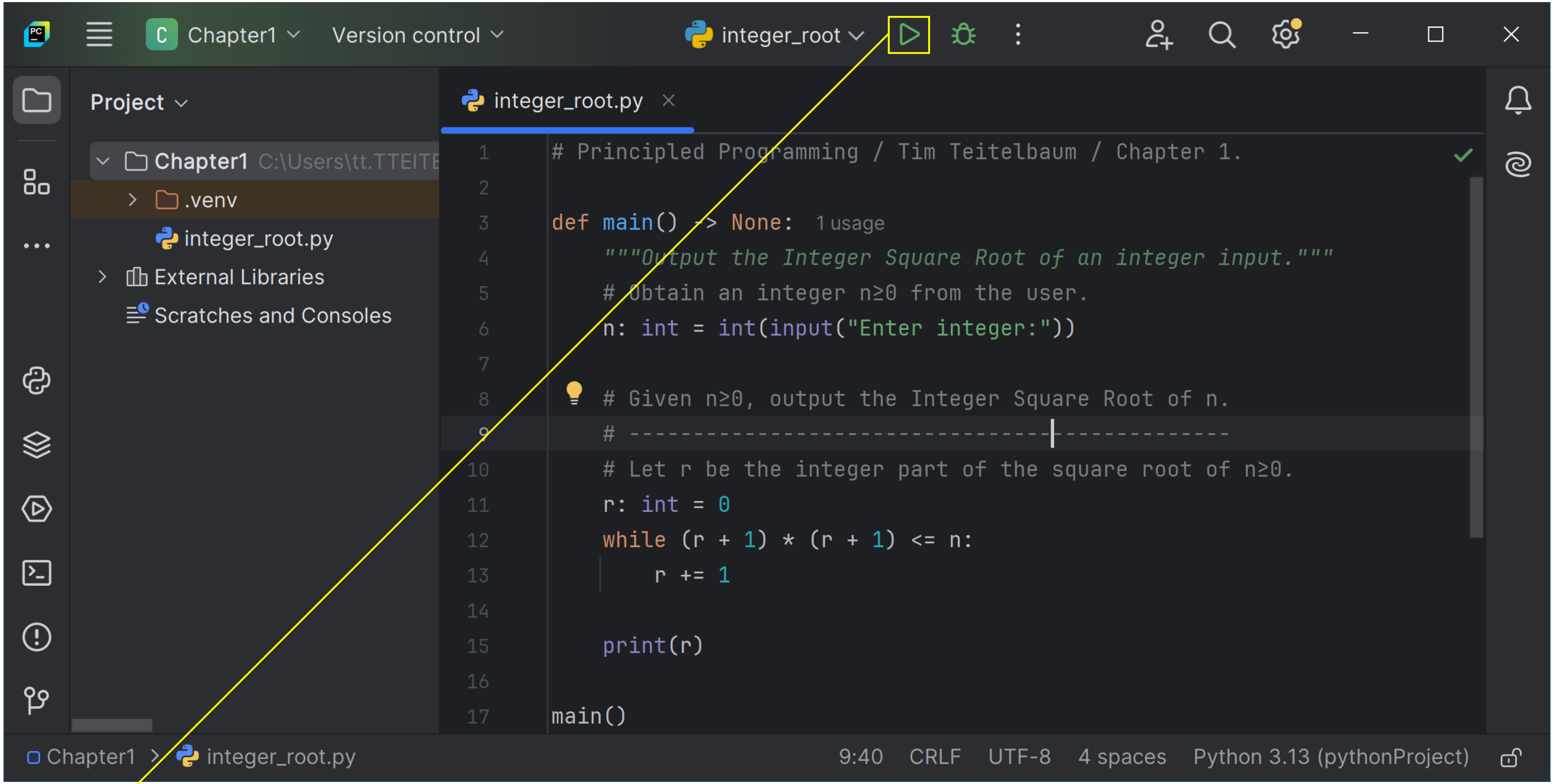


Code is typically edited and executed in an Integrated Development Environment (IDE), for example, PyCharm.

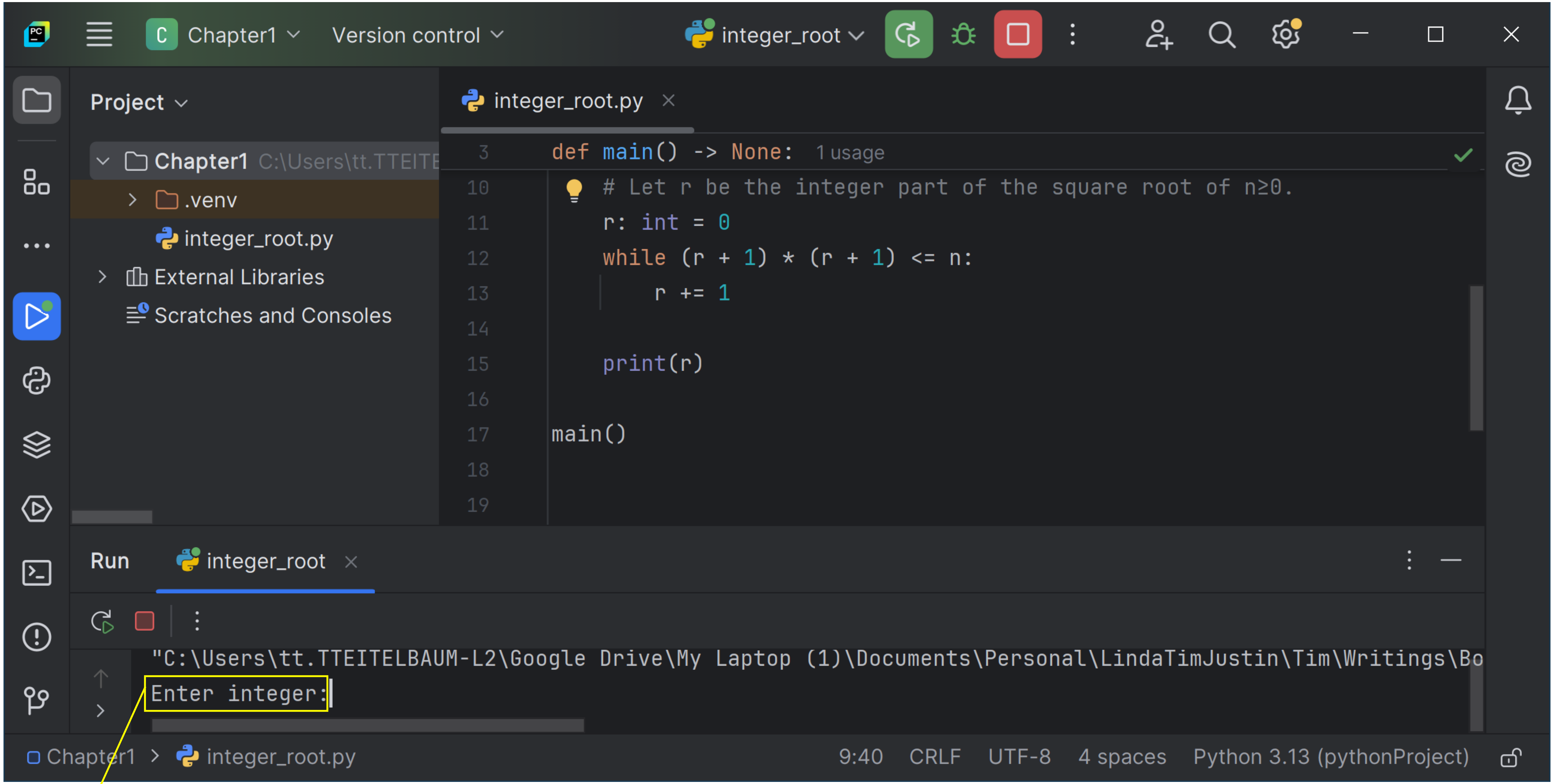
Pragmatics



Here, we had appended a line to the end of the code file that invokes method `main`.

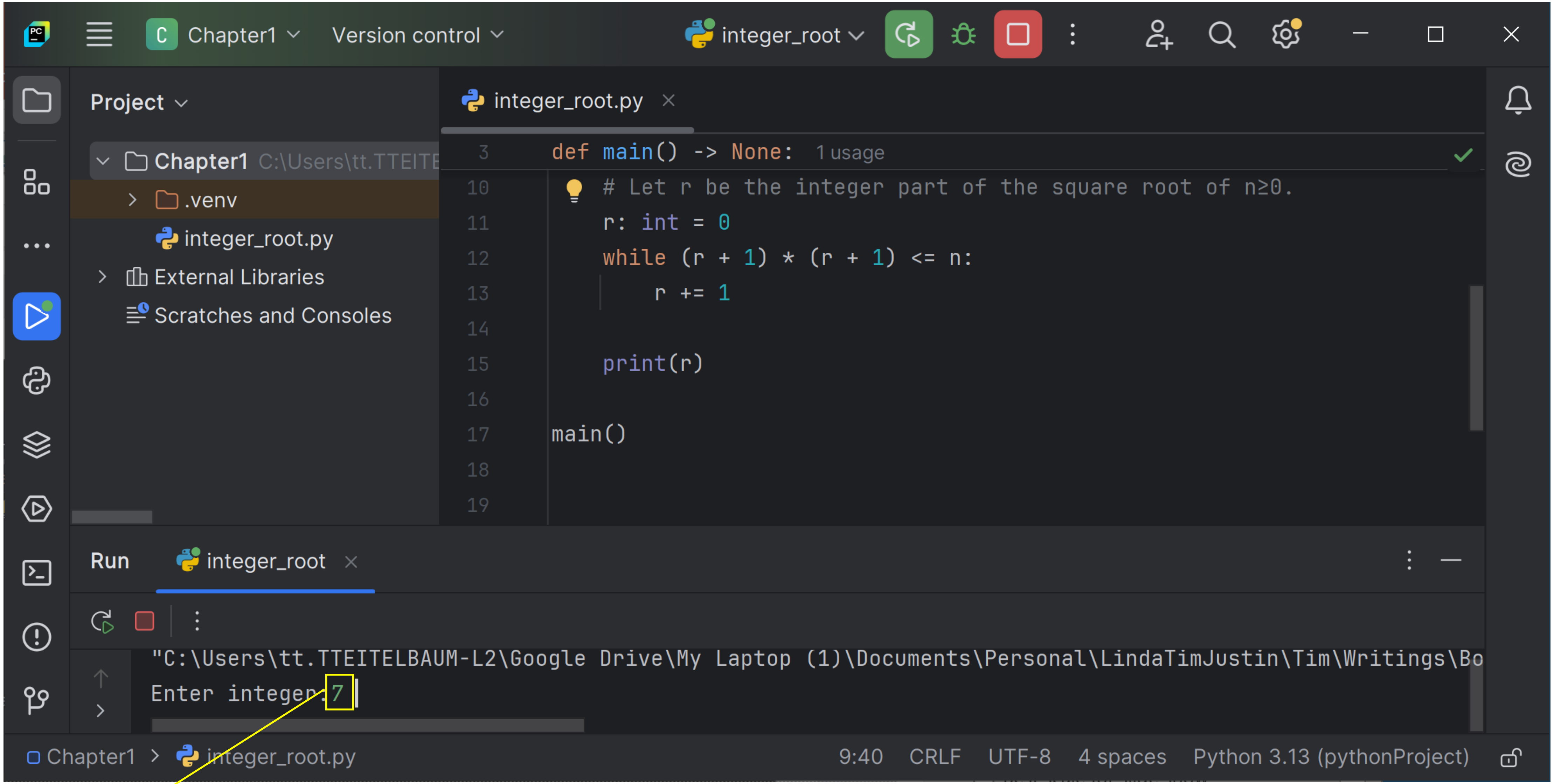


We click the icon requesting program execution.

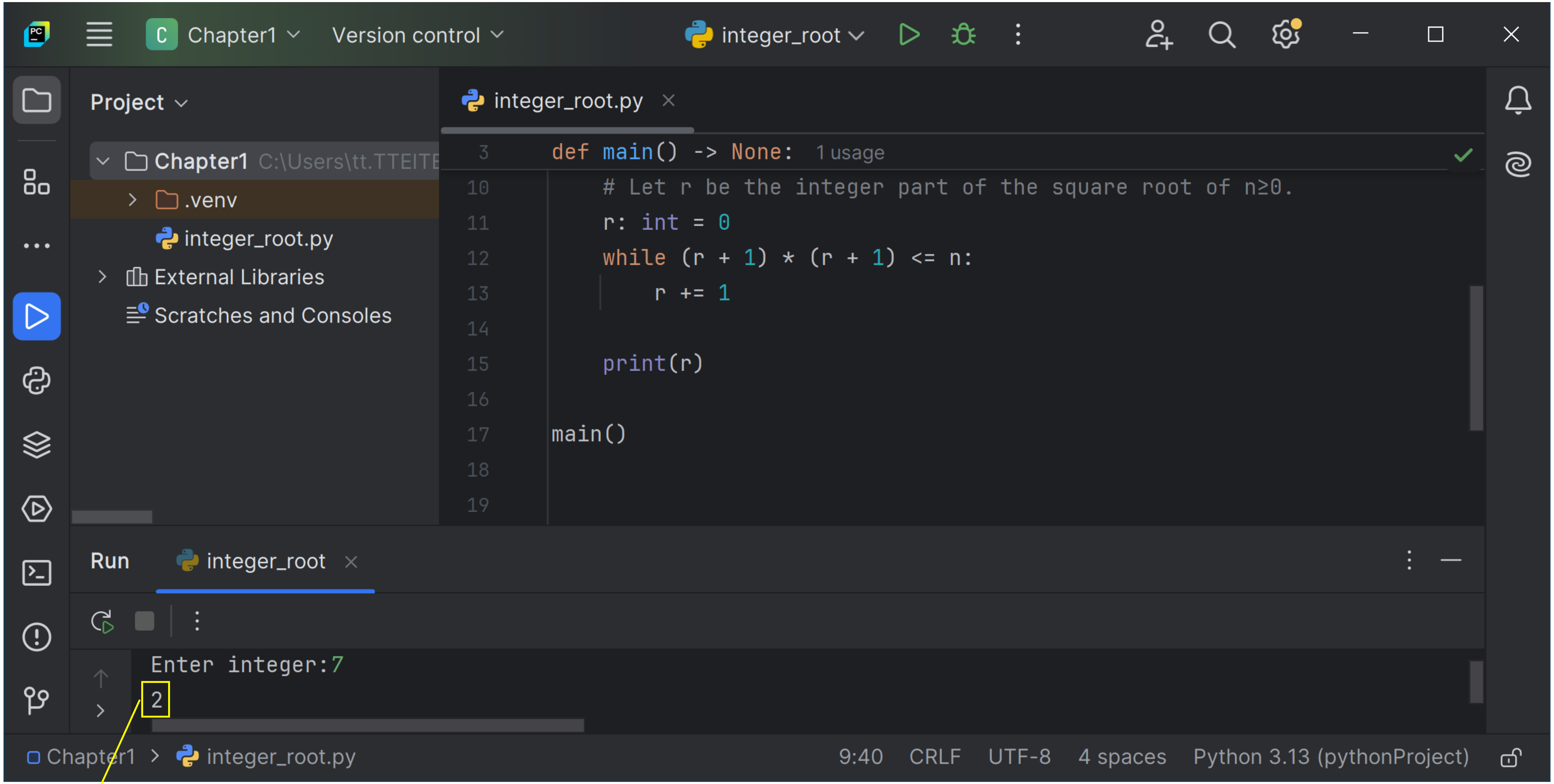


The prompt requesting an integer input appears in the terminal window.

Pragmatics



We enter “7”.



The program responds with “2”, and then completes its execution.

Summary

Goals

Elements of methodology

- Precepts, Patterns, Analysis, Process

Core programming-language constructs

- (almost all that we will need)

Illustrated the approach with a complete example