61A Lecture 6

Friday, September 9

Lambda Expressions

Lambda Expressions

```
>>> ten = 10
```

Lambda Expressions

```
>>> ten = 10

>>> square = x * x
```

Lambda Expressions

```
>>> ten = 10

>>> square = x * x
```

An expression: this one evaluates to a number

Lambda Expressions

```
>>> ten = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x

>>> square = lambda x: x * x
```

Lambda Expressions

```
>>> ten = 10

>>> square = x * x

>>> square = lambda x: x * x
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

Lambda Expressions

```
>>> ten = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

A function

Lambda Expressions

```
>>> ten = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

A function

with formal parameter x

Lambda Expressions

```
>>> ten = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

A function

with formal parameter x

and body "return x * x"

Lambda Expressions

```
>>> ten = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

Notice: no "return"

A function

with formal parameter x

and body "return x * x"

Lambda Expressions

```
>>> ten = 10

>>> square = x * x

>>> square = lambda x: x * x
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

Notice: no "return"

A function
with formal parameter x
and body "return x * x"

Must be a single expression

Lambda Expressions

```
>>> ten = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

Notice: no "return"

A function

with formal parameter x

and body "return x * x"

Must be a single expression

```
>>> square(4)
16
```

Lambda Expressions

```
>>> ten = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

Notice: no "return"

A function

with formal parameter x

and body "return x * x"

Must be a single expression

```
>>> square(4)
16
```

Lambda expressions are rare in Python, but important in general

Lambda Expressions Versus Def Statements

Lambda Expressions Versus Def Statements

VS

Lambda Expressions Versus Def Statements

```
square = lambda x: x * x
```

VS

Lambda Expressions Versus Def Statements

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

Lambda Expressions Versus Def Statements

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

- Both create a function with the same arguments & body

Lambda Expressions Versus Def Statements

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

- Both create a function with the same arguments & body
- Both of those functions are associated with the environment in which they are defined

Lambda Expressions Versus Def Statements

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

- Both create a function with the same arguments & body
- Both of those functions are associated with the environment in which they are defined
- Both bind that function to the name "square"

Lambda Expressions Versus Def Statements

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

- Both create a function with the same arguments & body
- Both of those functions are associated with the environment in which they are defined
- Both bind that function to the name "square"
- Only the def statement gives the function an intrinsic name

Lambda Expressions Versus Def Statements

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

- Both create a function with the same arguments & body
- Both of those functions are associated with the environment in which they are defined
- Both bind that function to the name "square"
- Only the def statement gives the function an intrinsic name

```
<lambda>(x):
    return x * x
```

Lambda Expressions Versus Def Statements

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

- Both create a function with the same arguments & body
- Both of those functions are associated with the environment in which they are defined
- Both bind that function to the name "square"
- Only the def statement gives the function an intrinsic name

```
<lambda>(x):
    return x * x
```

```
square(x):
    return x * x
```

Function Currying

Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions

Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions

Currying: Transforming a multi-argument function into a single-argument, higher-order function.

Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions

Currying: Transforming a multi-argument function into a single-argument, higher-order function.

Fun Fact: Currying was discovered by Moses Schönfinkel and later re-discovered by Haskell Curry.

Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions

Currying: Transforming a multi-argument function into a single-argument, higher-order function.

Fun Fact: Currying was discovered by Moses Schönfinkel and later re-discovered by Haskell Curry.

Schönfinkeling?

Newton's Method Background

Finds approximations to zeroes of differentiable functions

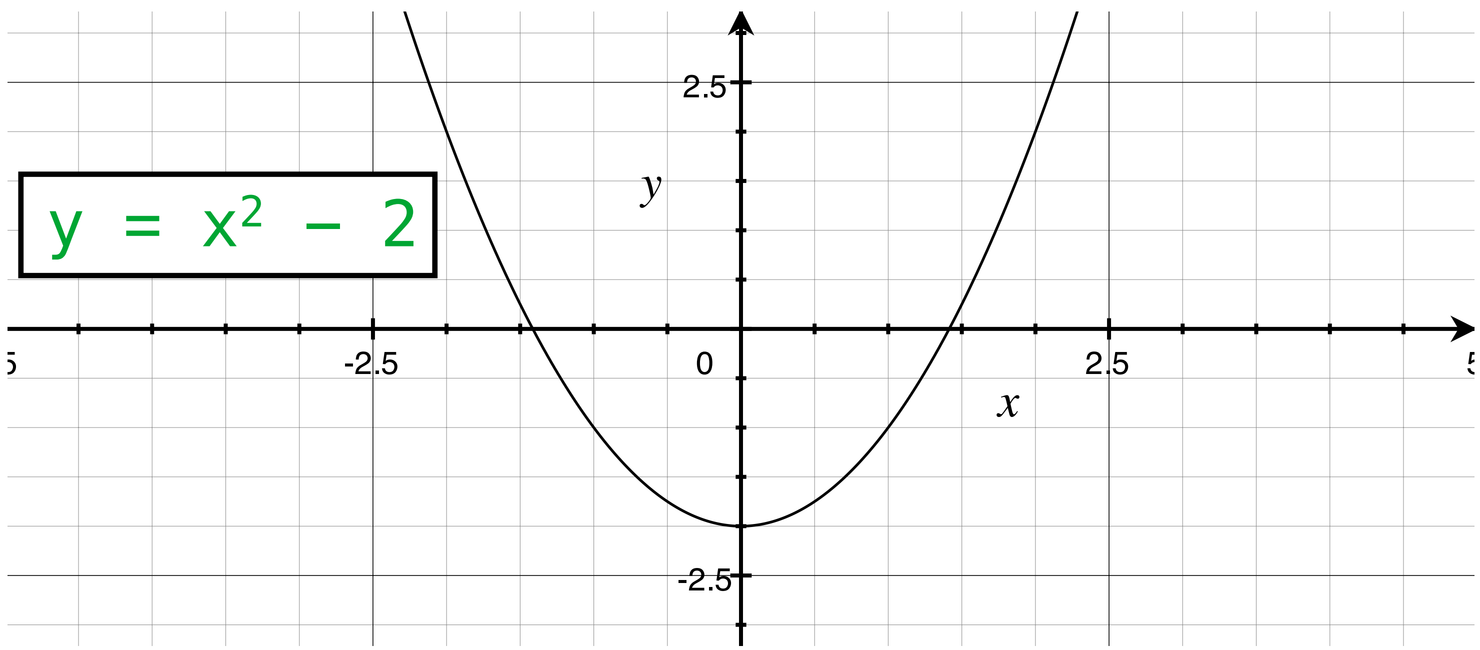
Newton's Method Background

Finds approximations to zeroes of differentiable functions

$$\boxed{y = x^2 - 2}$$

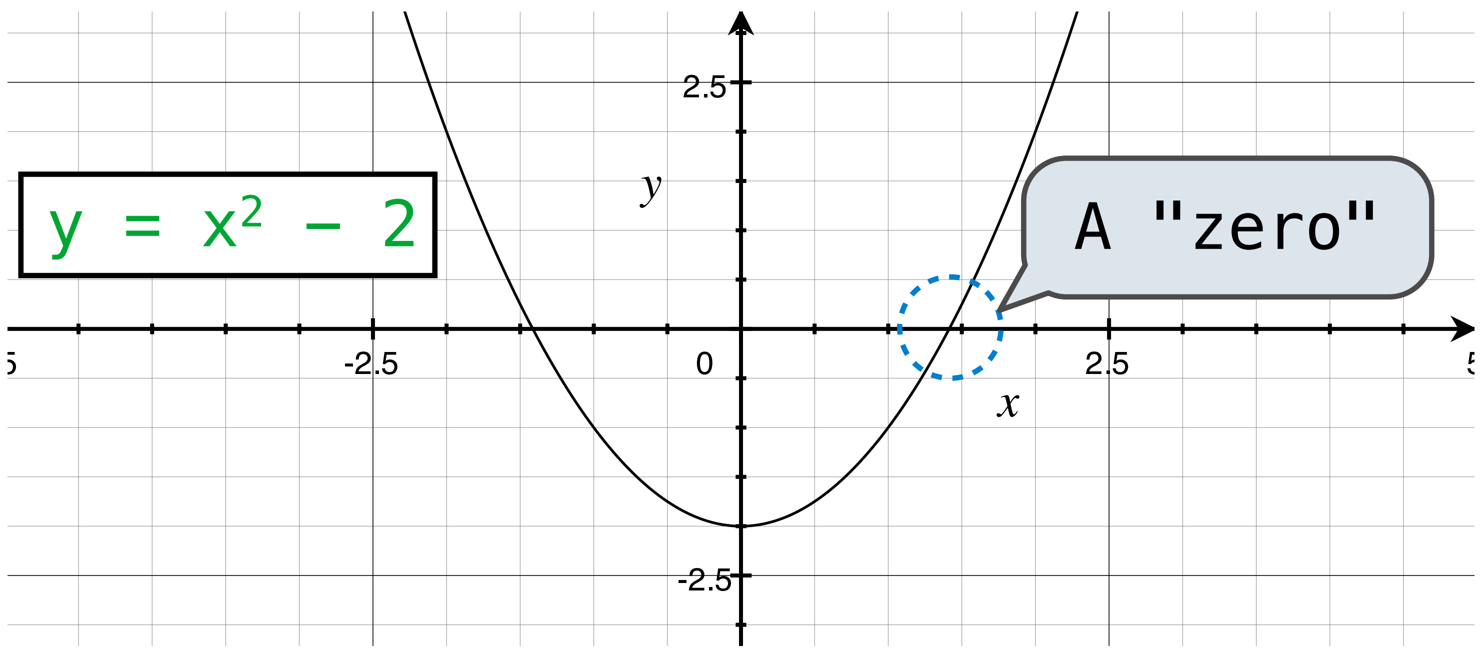
Newton's Method Background

Finds approximations to zeroes of differentiable functions



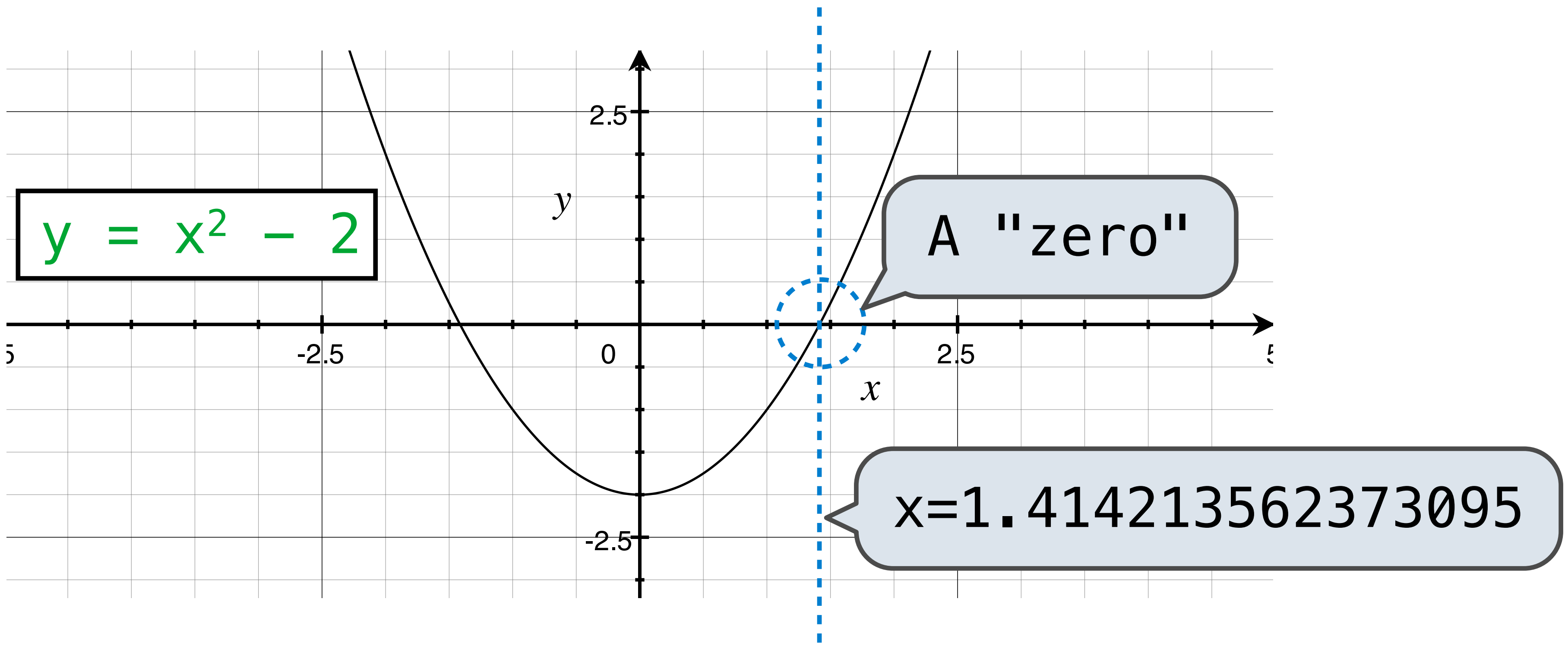
Newton's Method Background

Finds approximations to zeroes of differentiable functions



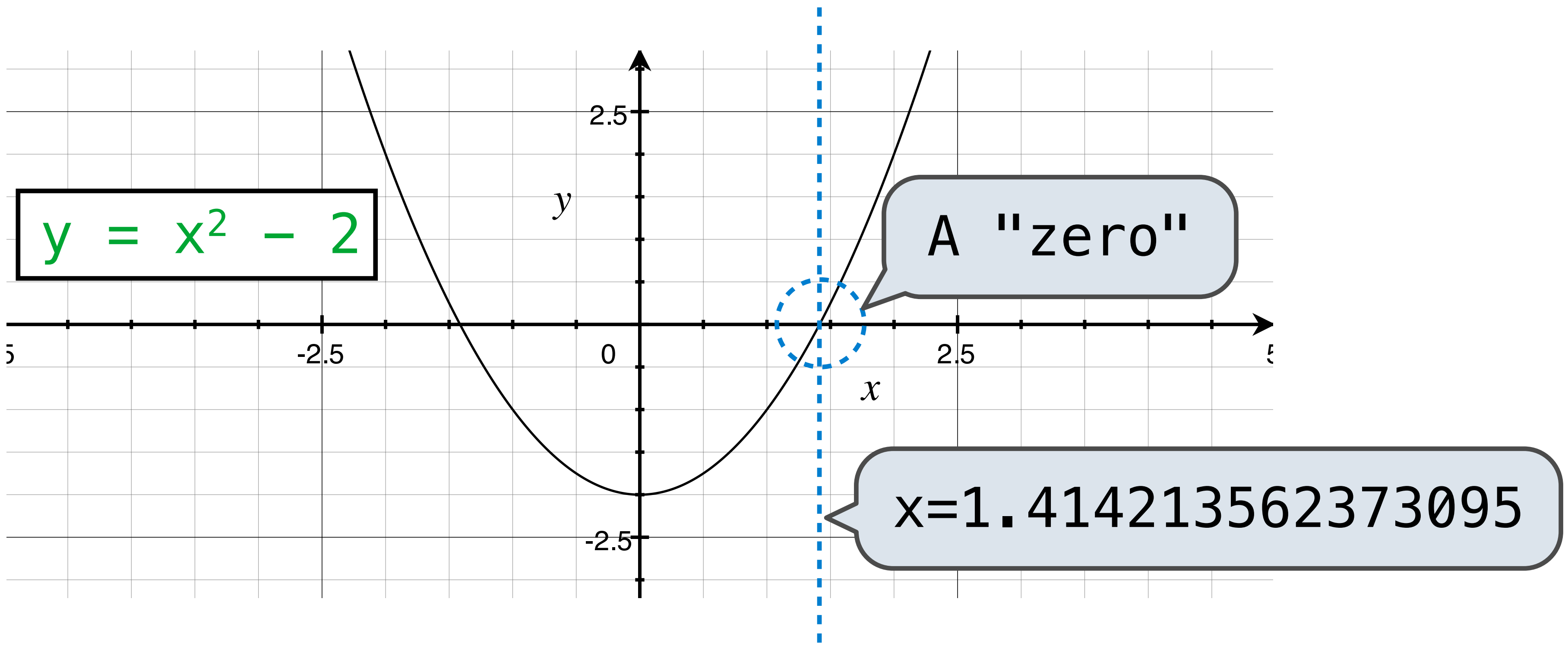
Newton's Method Background

Finds approximations to zeroes of differentiable functions



Newton's Method Background

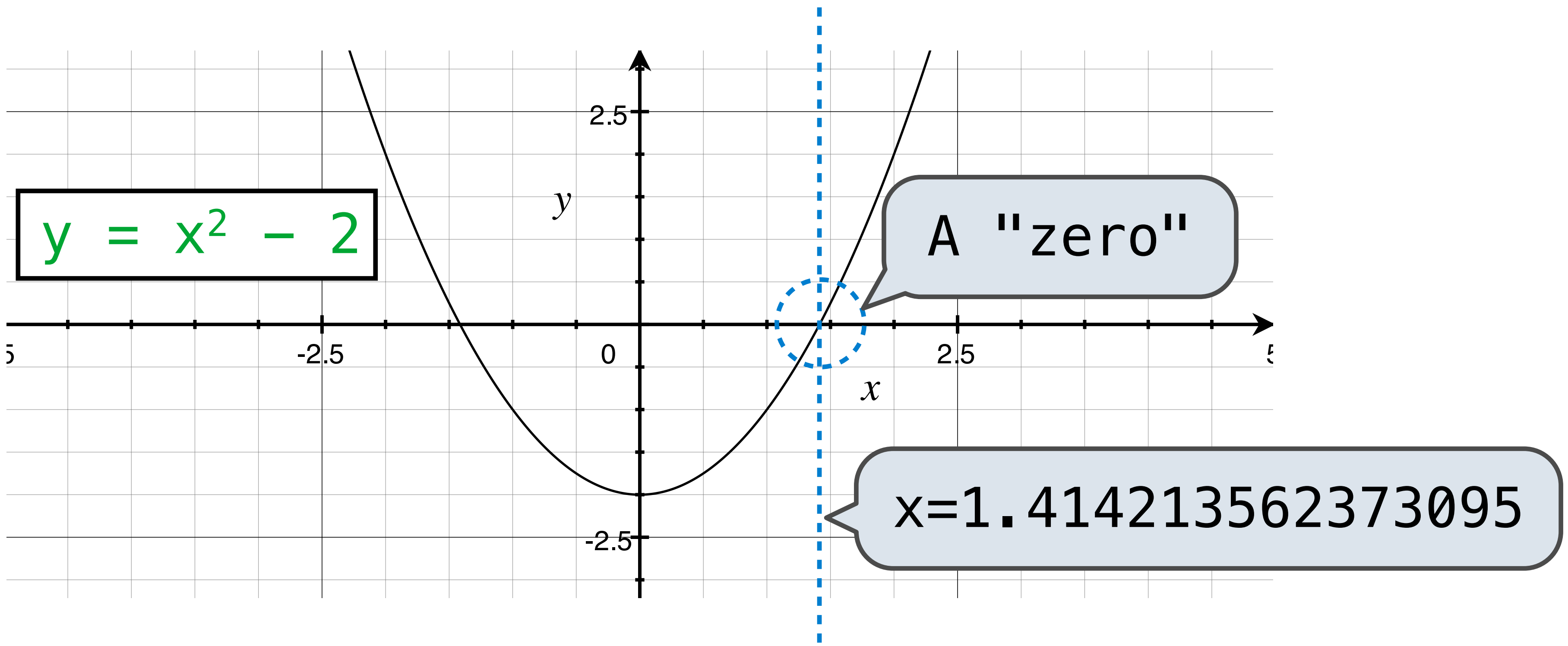
Finds approximations to zeroes of differentiable functions



Application: a method for (approximately) computing square roots, using only basic arithmetic.

Newton's Method Background

Finds approximations to zeroes of differentiable functions

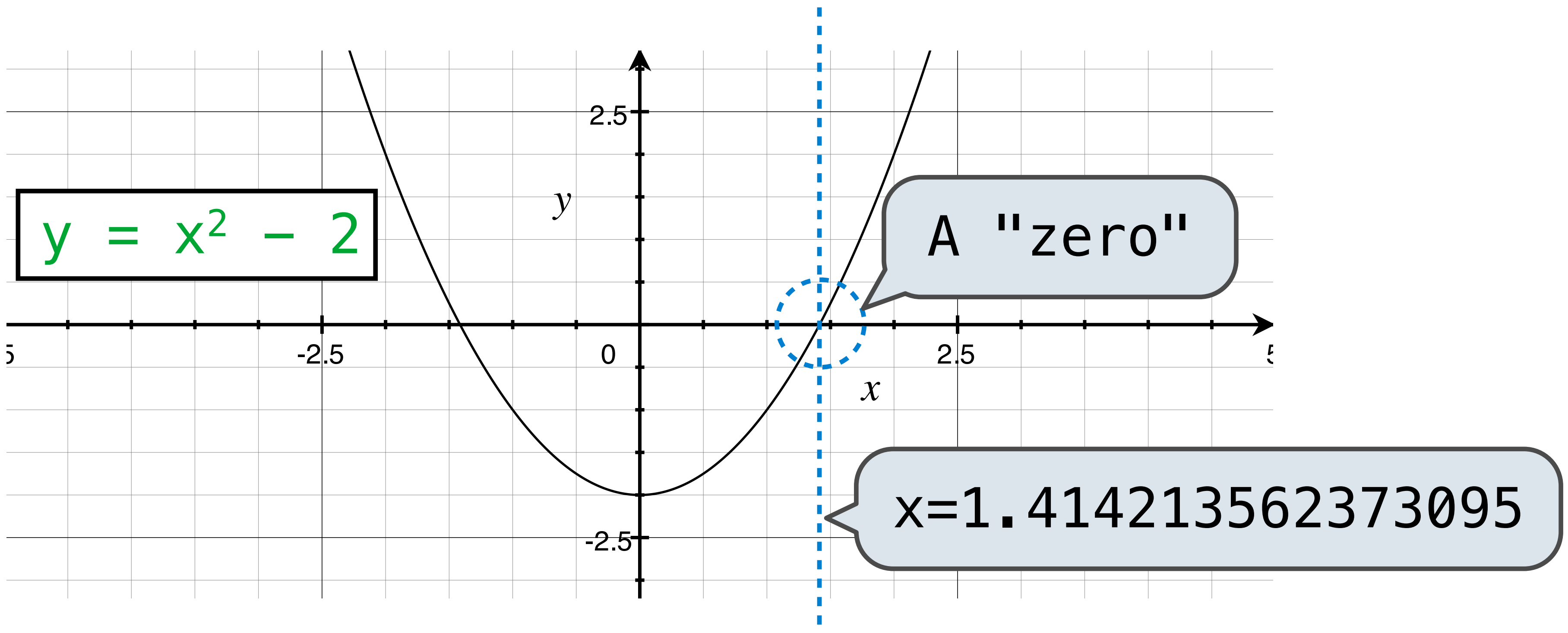


Application: a method for (approximately) computing square roots, using only basic arithmetic.

The positive zero of $y = x^2 - a$ is

Newton's Method Background

Finds approximations to zeroes of differentiable functions



Application: a method for (approximately) computing square roots, using only basic arithmetic.

The positive zero of $y = x^2 - a$ is $\sqrt{a}$

Newton's Method

`Begin with a function f and`

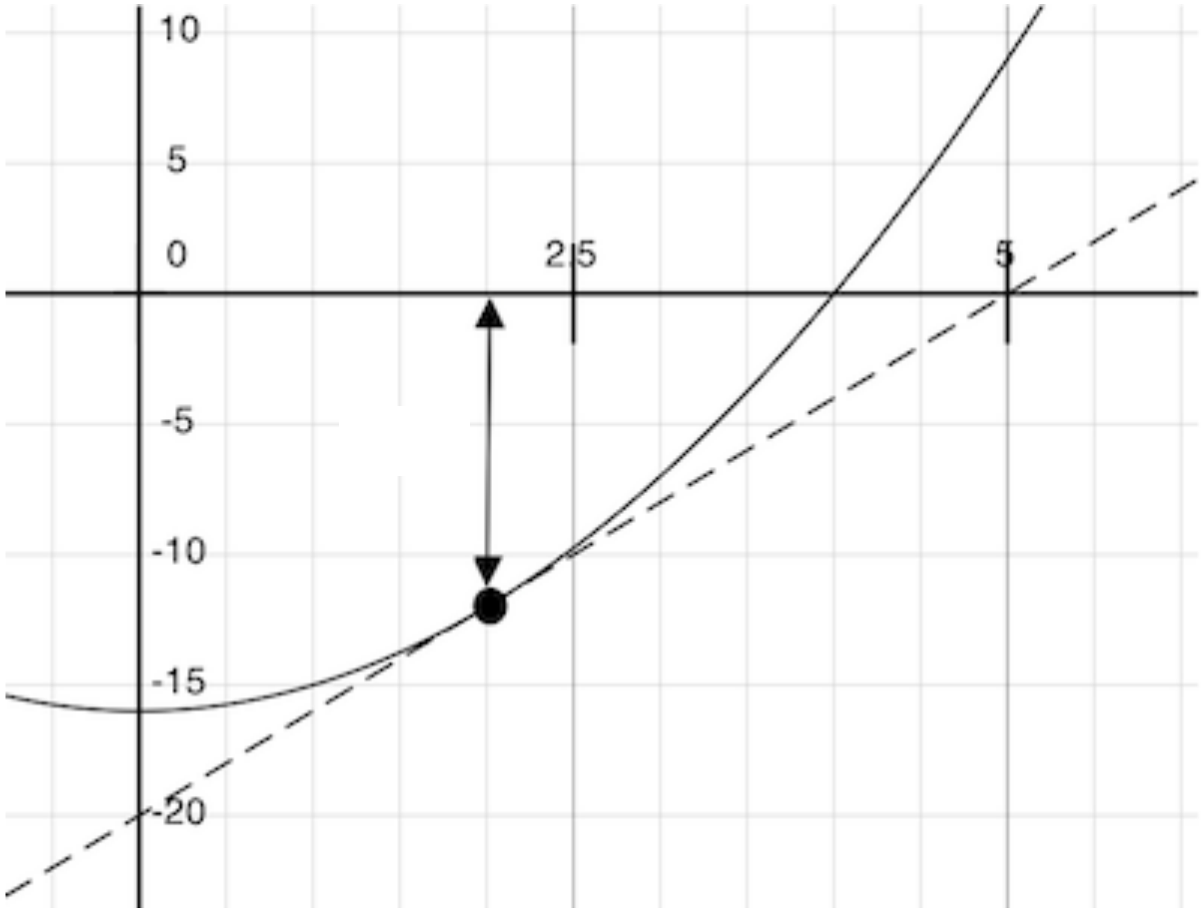
`an initial guess x`

$$x - \frac{f(x)}{f'(x)}$$

Newton's Method

Begin with a function f and

an initial guess x

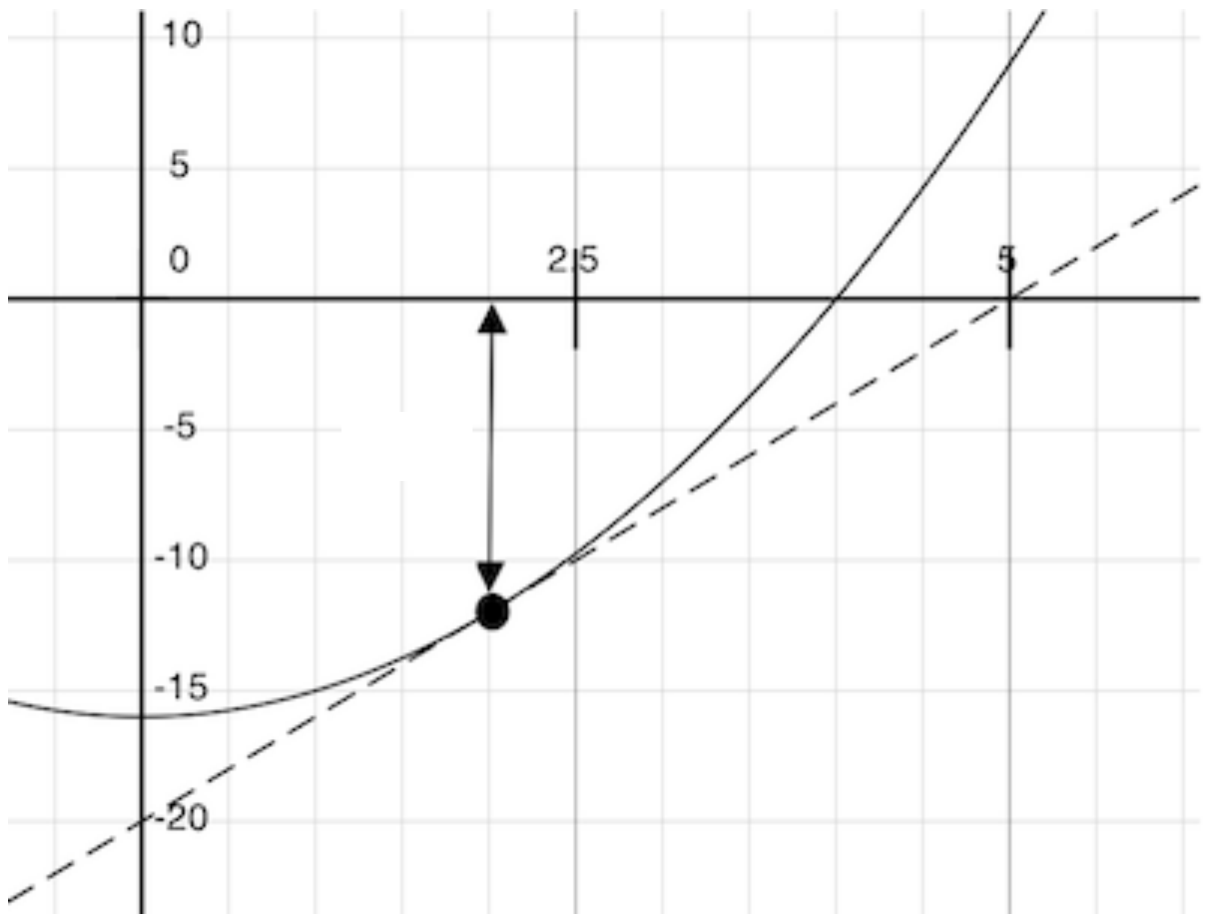


$$x - \frac{f(x)}{f'(x)}$$

Newton's Method

Begin with a function f and

an initial guess x



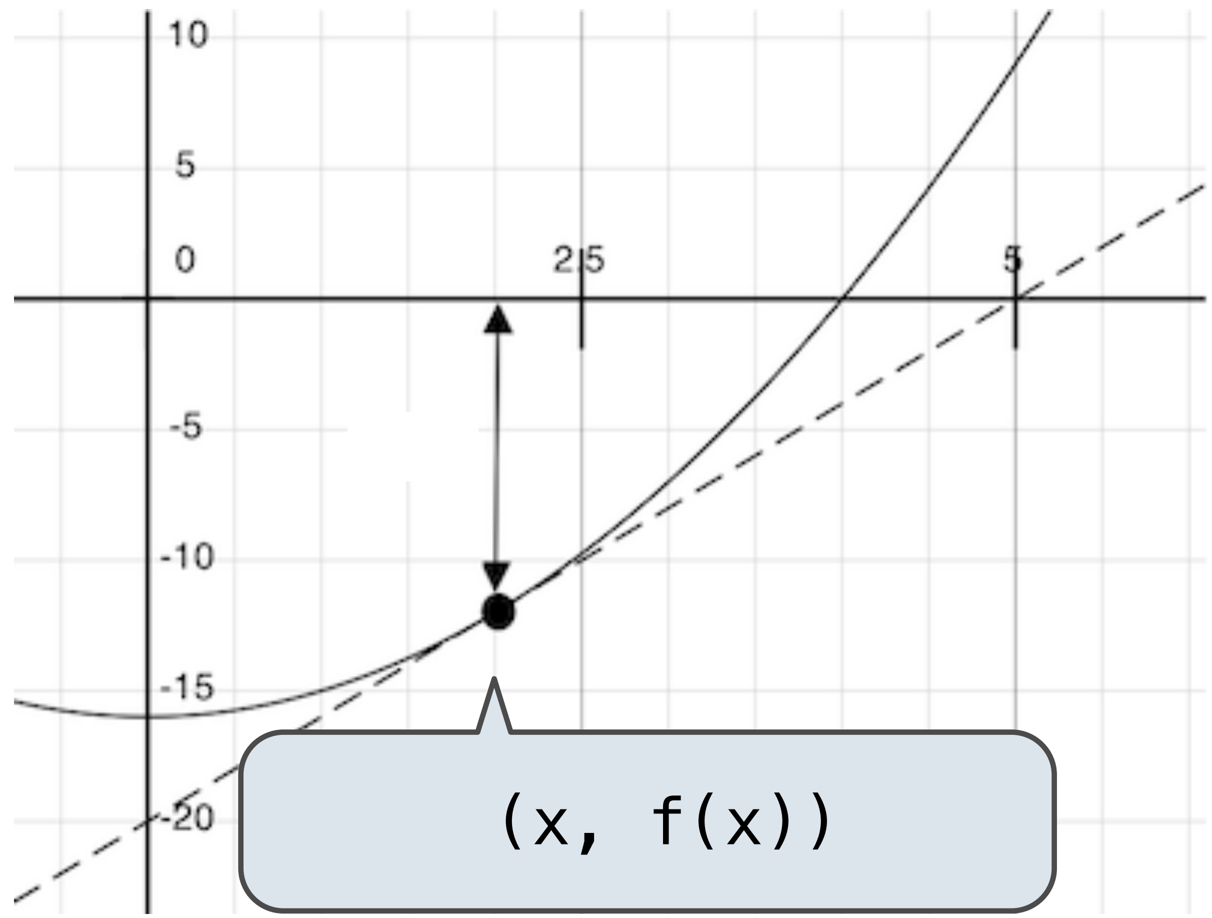
1. Compute the value of f at guess: f(x)

$$x - \frac{f(x)}{f'(x)}$$

Newton's Method

Begin with a function f and

an initial guess x



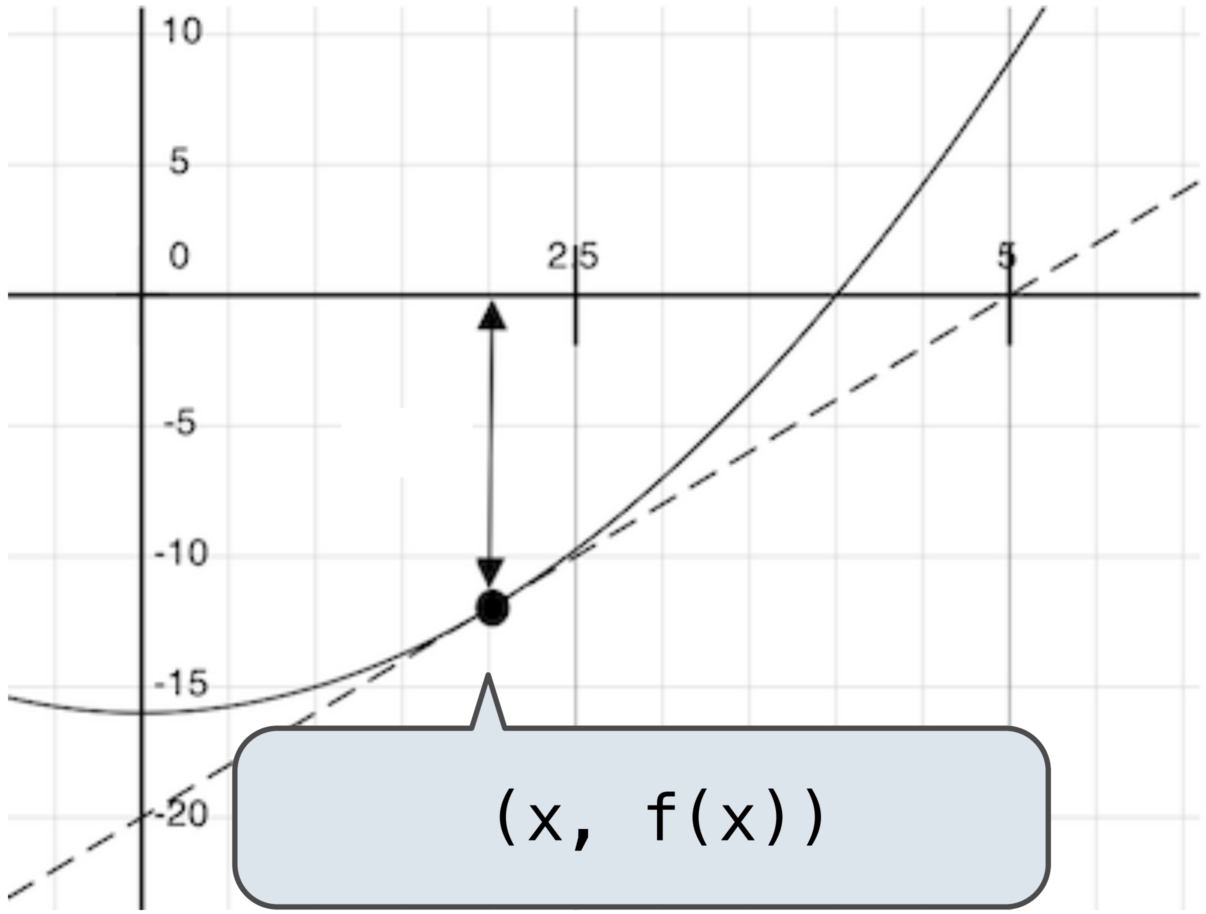
1. Compute the value of f at guess: f(x)

$$x - \frac{f(x)}{f'(x)}$$

Newton's Method

Begin with a function f and

an initial guess x



1. Compute the value of f at guess: f(x)

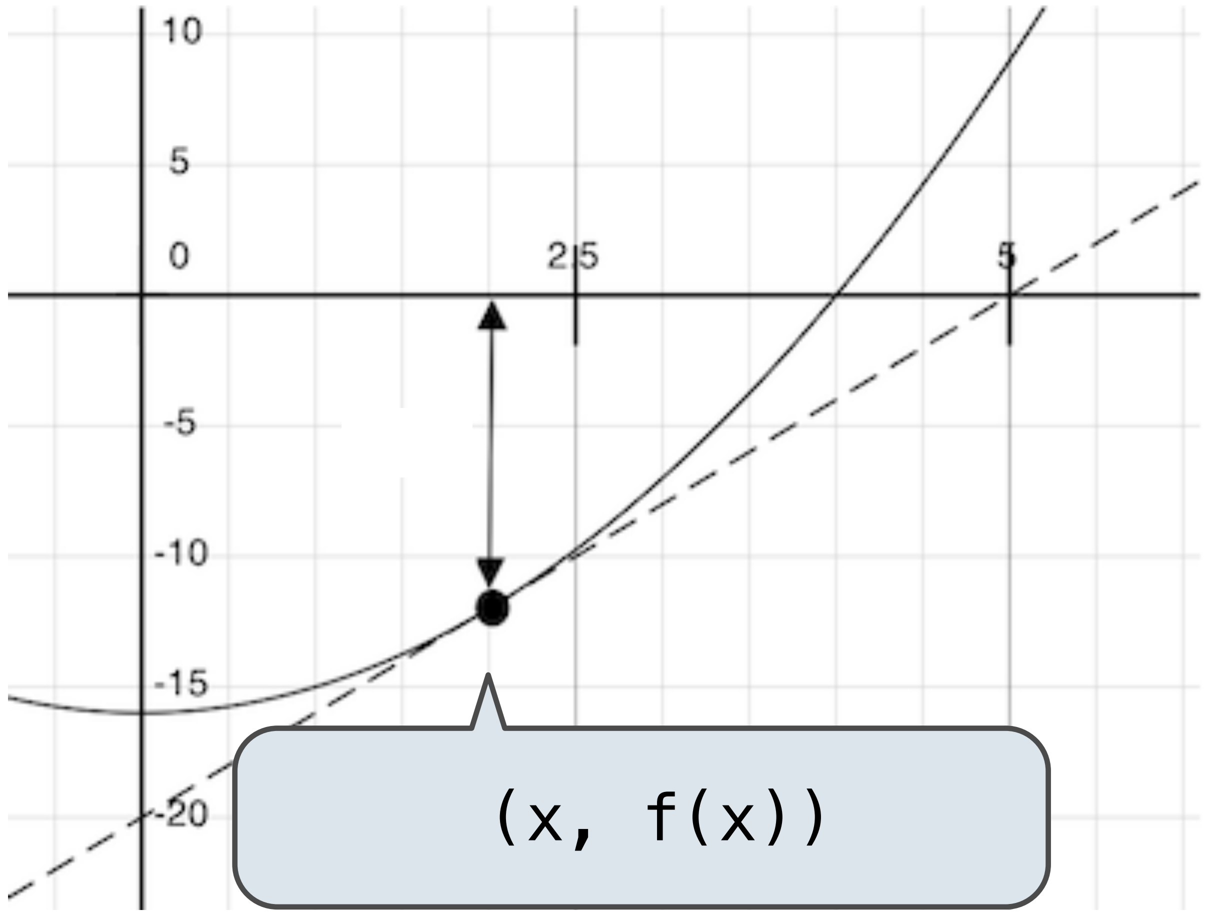
2. Compute the derivative of f at guess: f'(x)

$$x - \frac{f(x)}{f'(x)}$$

Newton's Method

Begin with a function f and

an initial guess x



1. Compute the value of f at guess: f(x)

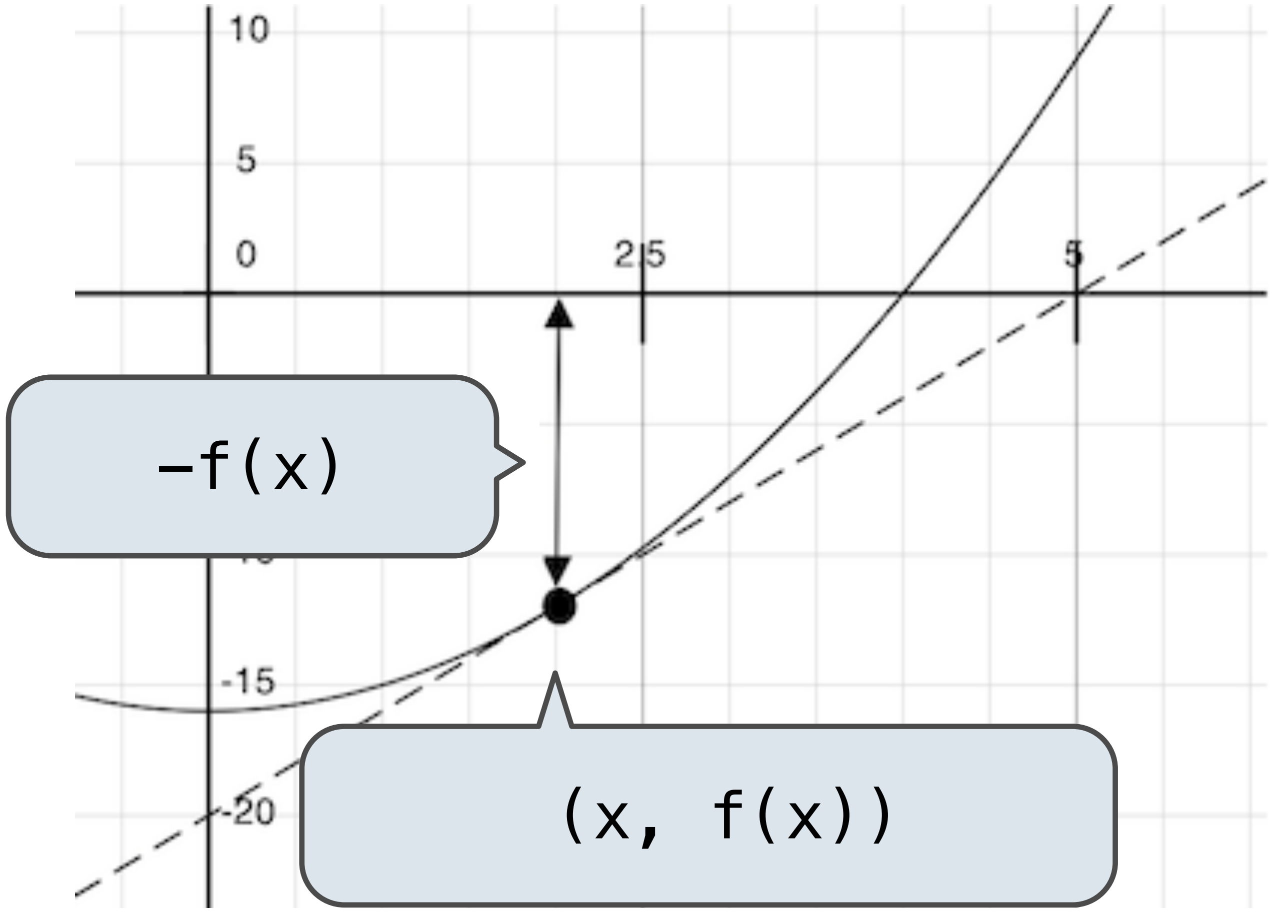
2. Compute the derivative of f at guess: f'(x)

3. Update guess x to be: $x - \frac{f(x)}{f'(x)}$

Newton's Method

Begin with a function f and

an initial guess x



1. Compute the value of f at guess: f(x)

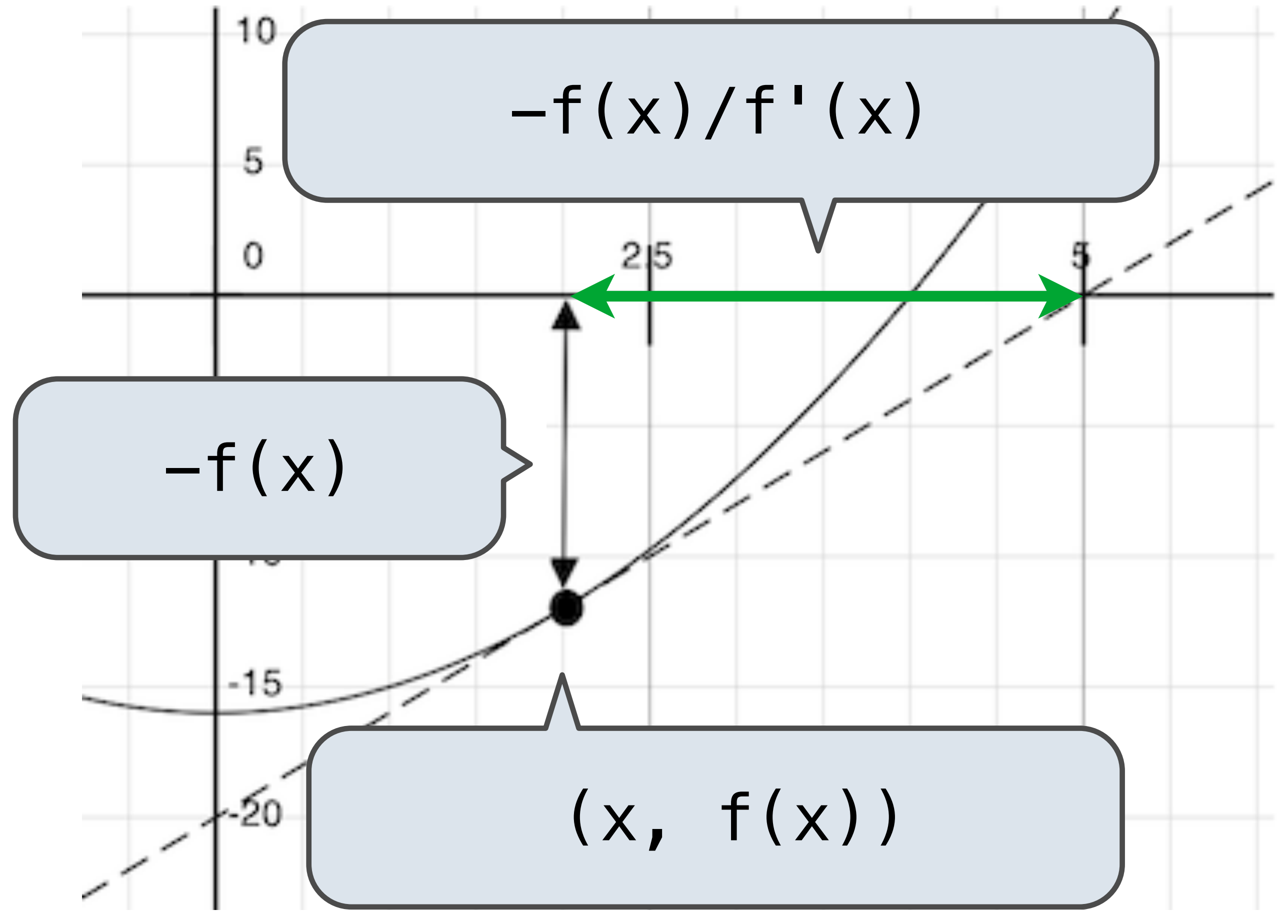
2. Compute the derivative of f at guess: f'(x)

3. Update guess x to be: $x - \frac{f(x)}{f'(x)}$

Newton's Method

Begin with a function f and

an initial guess x



1. Compute the value of f at guess: f(x)

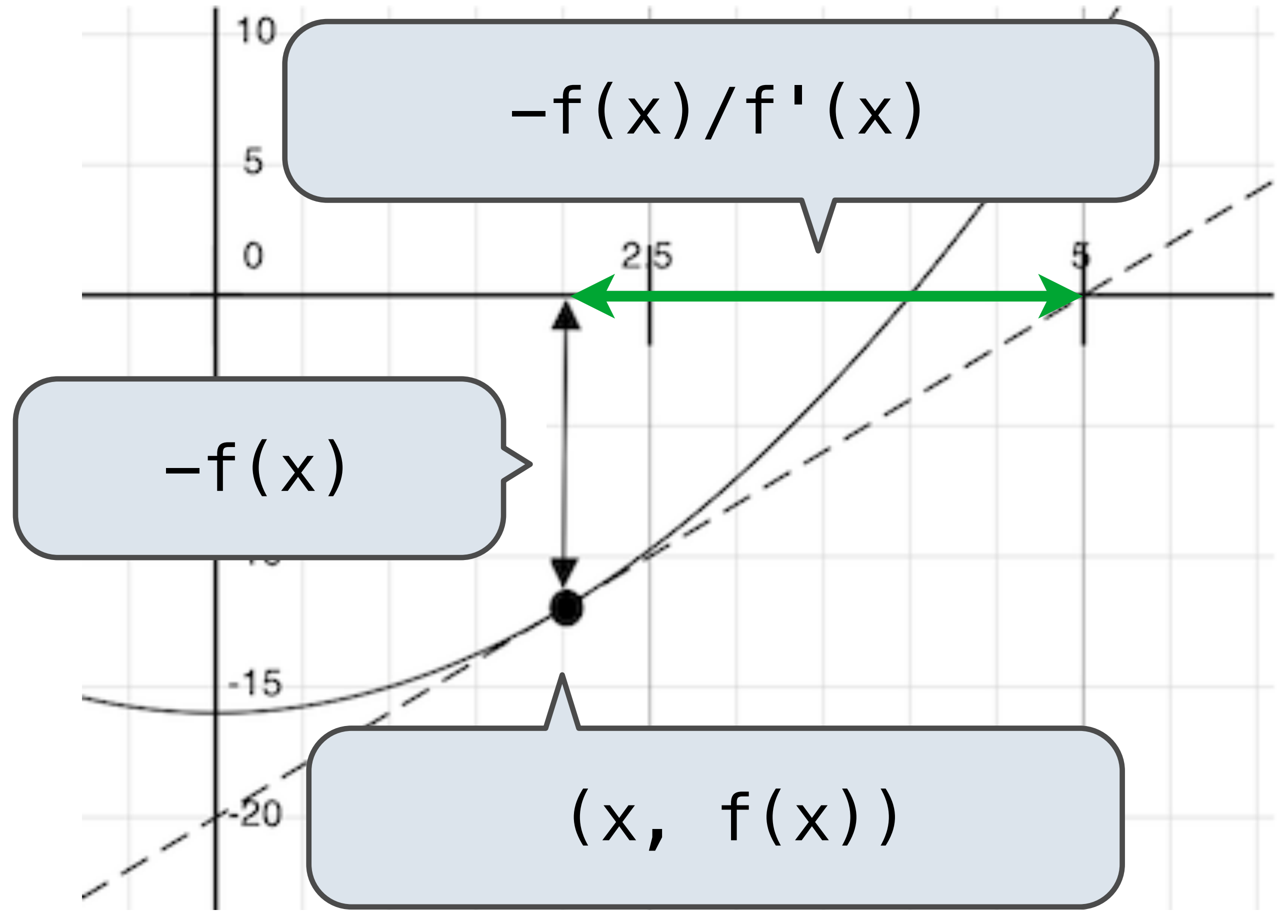
2. Compute the derivative of f at guess: f'(x)

3. Update guess x to be: $x - \frac{f(x)}{f'(x)}$

Newton's Method

Begin with a function f and

an initial guess x



1. Compute the value of f at guess: f(x)

2. Compute the derivative of f at guess: f'(x)

3. Update guess x to be: $x - \frac{f(x)}{f'(x)}$

Visualization of Newton's Method

(Demo)

http://en.wikipedia.org/wiki/File:NewtonIteration_Ani.gif

Using Newton's Method

$f(x)=x^2 - 2$

Using Newton's Method

How to find the square root of 2?

$f(x)=x^2 - 2$

Using Newton's Method

How to find the square root of 2?

```
>>> f = lambda x: x*x - 2
>>> find_root(f, 1)
```

$f(x)=x^2 - 2$

Using Newton's Method

How to find the square root of 2?

```
>>> f = lambda x: x*x - 2
>>> find_root(f, 1)
```

$f(x)=x^2 - 2$

How to find the log base 2 of 1024?

Using Newton's Method

How to find the square root of 2?

```
>>> f = lambda x: x*x - 2
>>> find_root(f, 1)
```

$f(x) = x^2 - 2$

How to find the log base 2 of 1024?

```
>>> g = lambda x: pow(2, x) - 1024
>>> find_root(g, 1)
```

Using Newton's Method

How to find the square root of 2?

```
>>> f = lambda x: x*x - 2
>>> find_root(f, 1)
```

$f(x)=x^2 - 2$

How to find the log base 2 of 1024?

```
>>> g = lambda x: pow(2, x) - 1024
>>> find_root(g, 1)
```

What number is one less than its square?

Using Newton's Method

How to find the square root of 2?

```
>>> f = lambda x: x*x - 2
>>> find_root(f, 1)
```

$f(x)=x^2 - 2$

How to find the log base 2 of 1024?

```
>>> g = lambda x: pow(2, x) - 1024
>>> find_root(g, 1)
```

What number is one less than its square?

```
>>> h = lambda x: x*x - (x+1)
>>> find_root(h, 1)
```

Special Case: Square Roots

$$x = \frac{x + \frac{a}{x}}{2}$$

Babylonian Method

Special Case: Square Roots

How to compute square_root(a)

Idea: Iteratively refine a guess x about the square root of a

$$x = \frac{x + \frac{a}{x}}{2}$$

Babylonian Method

Special Case: Square Roots

How to compute square_root(a)

Idea: Iteratively refine a guess x about the square root of a

Update: $$x = \frac{x + \frac{a}{x}}{2}$$

Babylonian Method

Special Case: Square Roots

How to compute square_root(a)

Idea: Iteratively refine a guess x about the square root of a

Update: $$x = \frac{x + \frac{a}{x}}{2}$$

Babylonian Method

What *guess* should start the computation?

Special Case: Square Roots

How to compute square_root(a)

Idea: Iteratively refine a guess x about the square root of a

Update:

$$x = \frac{x + \frac{a}{x}}{2}$$

Babylonian Method

What *guess* should start the computation?

How do we know when we are finished?

Special Case: Square Roots

How to compute square_root(a)

Idea: Iteratively refine a guess x about the square root of a

Update:

$$x = \frac{x + \frac{a}{x}}{2}$$

Babylonian Method

Implementation questions:

What *guess* should start the computation?

How do we know when we are finished?

Special Case: Square Roots

$$x = \frac{2 \cdot x + \frac{a}{x^2}}{3}$$

Special Case: Square Roots

`How to compute cube_root(a)`

Idea: Iteratively refine a guess x about the cube root of a

$$x = \frac{2 \cdot x + \frac{a}{x^2}}{3}$$

Special Case: Square Roots

How to compute `cube_root(a)`

Idea: Iterativey refine a guess x about the cube root of a

$$x = \frac{2 \cdot x + \frac{a}{x^2}}{3}$$

What *guess* should start the computation?

Special Case: Square Roots

How to compute cube_root(a)

Idea: Iteratively refine a guess x about the cube root of a

$$x = \frac{2 \cdot x + \frac{a}{x^2}}{3}$$

What *guess* should start the computation?

How do we know when we are finished?

Special Case: Square Roots

How to compute `cube_root(a)`

Idea: Iterately refine a guess x about the cube root of a

$$x = \frac{2 \cdot x + \frac{a}{x^2}}{3}$$

Implementation questions:

What *guess* should start the computation?

How do we know when we are finished?

Special Case: Square Roots

How to compute `cube_root(a)`

Idea: Iteratively refine a guess x about the cube root of a

Update:

$$x = \frac{2 \cdot x + \frac{a}{x^2}}{3}$$

Implementation questions:

What *guess* should start the computation?

How do we know when we are finished?

Iterative Improvement

(Demo)

Iterative Improvement

(Demo)

```
def iter_improve(update, done, guess=1, max_updates=1000):
    """Iteratively improve guess with update until done returns a true value.

    guess -- An initial guess
    update -- A function from guesses to guesses; updates the guess
    done -- A function from guesses to boolean values; tests if guess is good

    >>> iter_improve(golden_update, golden_test)
    1.618033988749895
    """
    k = 0
    while not done(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess
```

Iterative Improvement

(Demo)

```
def golden_update(guess):
    return 1/guess + 1
```

```
def iter_improve(update, done, guess=1, max_updates=1000):
    """Iteratively improve guess with update until done returns a true value.

    guess -- An initial guess
    update -- A function from guesses to guesses; updates the guess
    done -- A function from guesses to boolean values; tests if guess is good

    >>> iter_improve(golden_update, golden_test)
    1.618033988749895
    """
    k = 0
    while not done(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess
```

Iterative Improvement

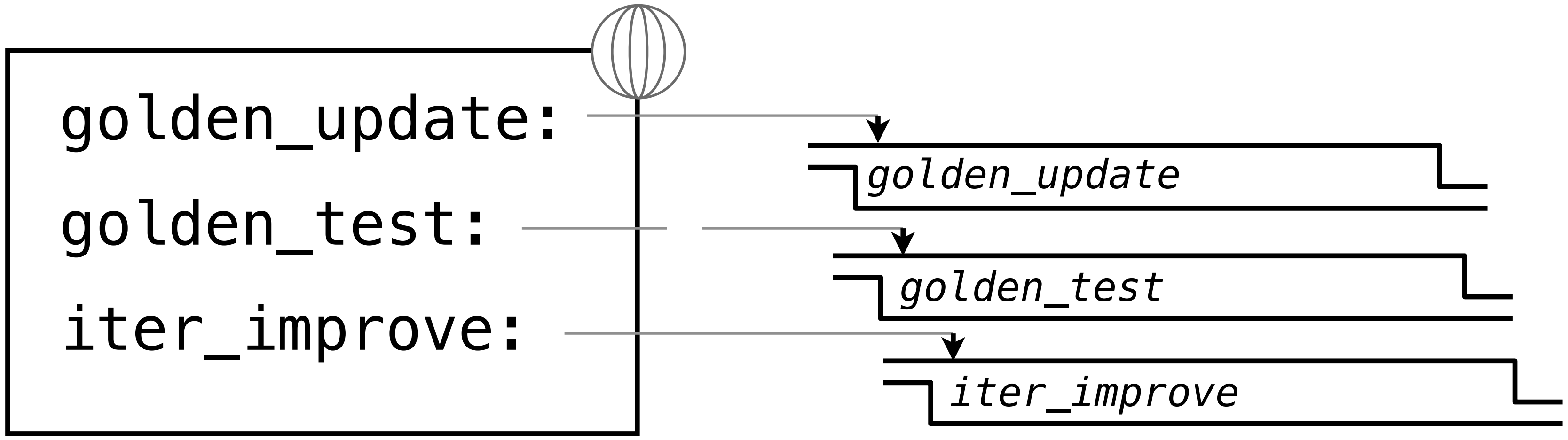
(Demo)

```
def golden_update(guess):
    return 1/guess + 1
```

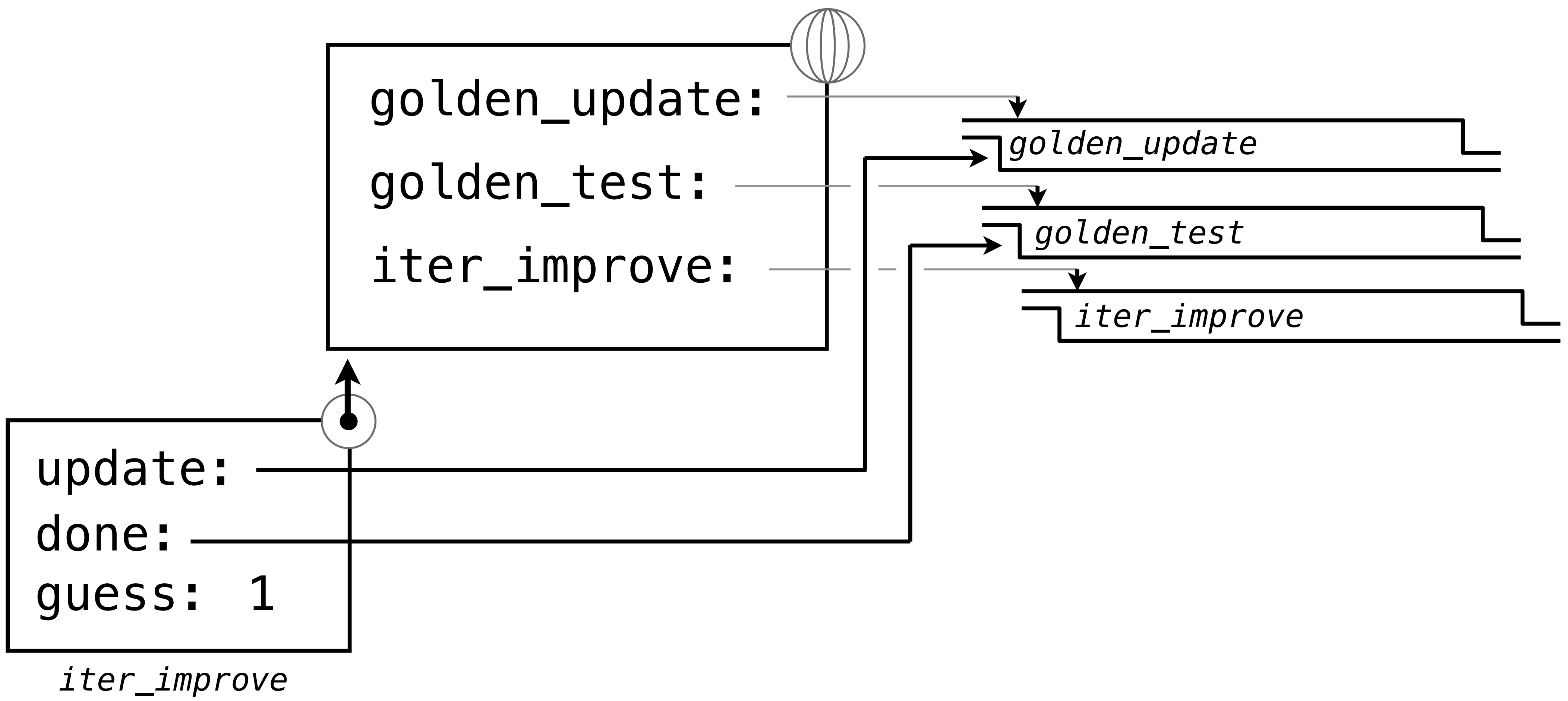
```
def golden_test(guess):
    return guess * guess == guess + 1
```

```
def iter_improve(update, done, guess=1, max_updates=1000):
    """Iteratively improve guess with update until done returns a true value.

    guess -- An initial guess
    update -- A function from guesses to guesses; updates the guess
    done -- A function from guesses to boolean values; tests if guess is good

    >>> iter_improve(golden_update, golden_test)
    1.618033988749895
    """
    k = 0
    while not done(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess
```

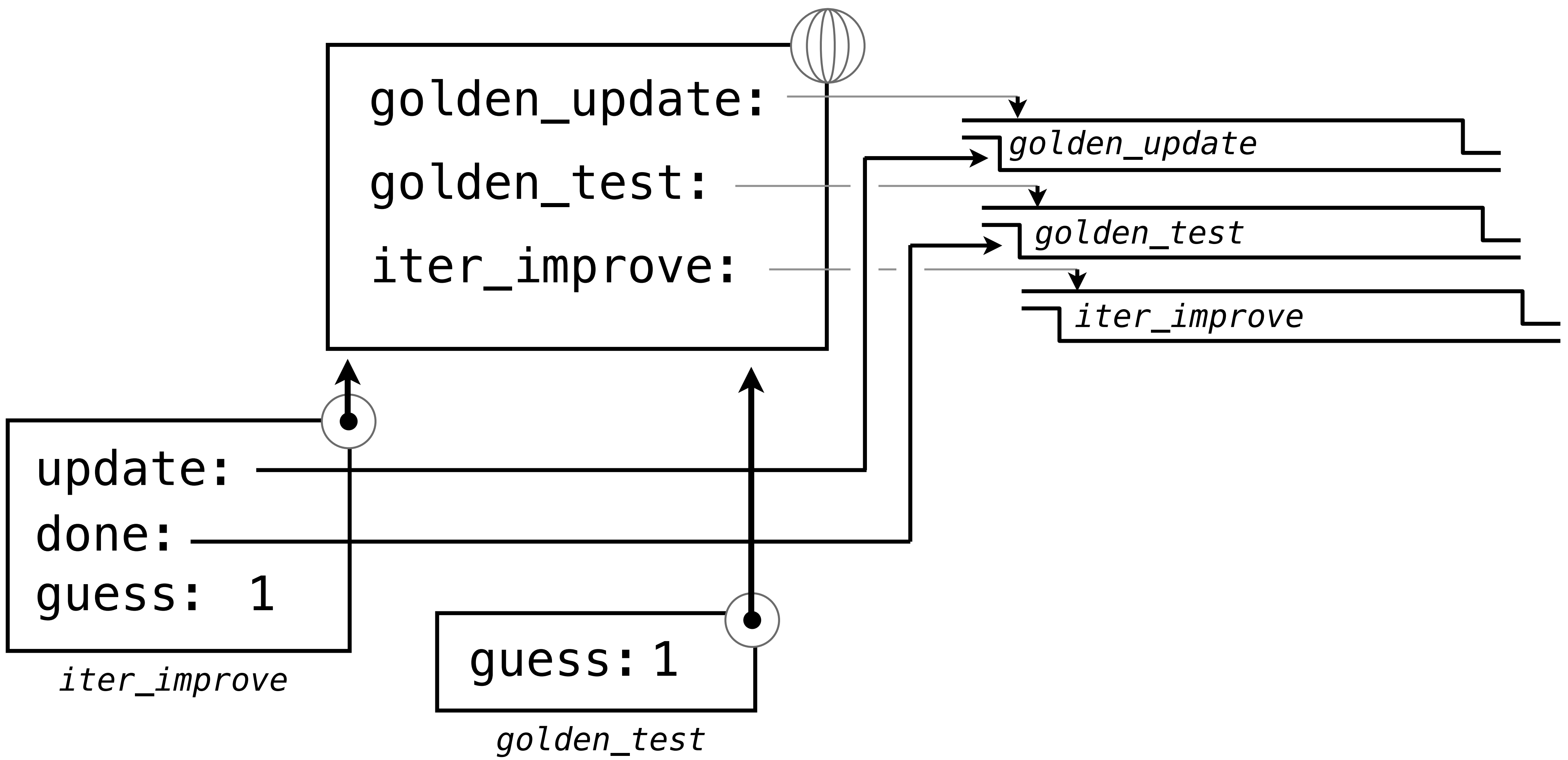
Iterative Improvement



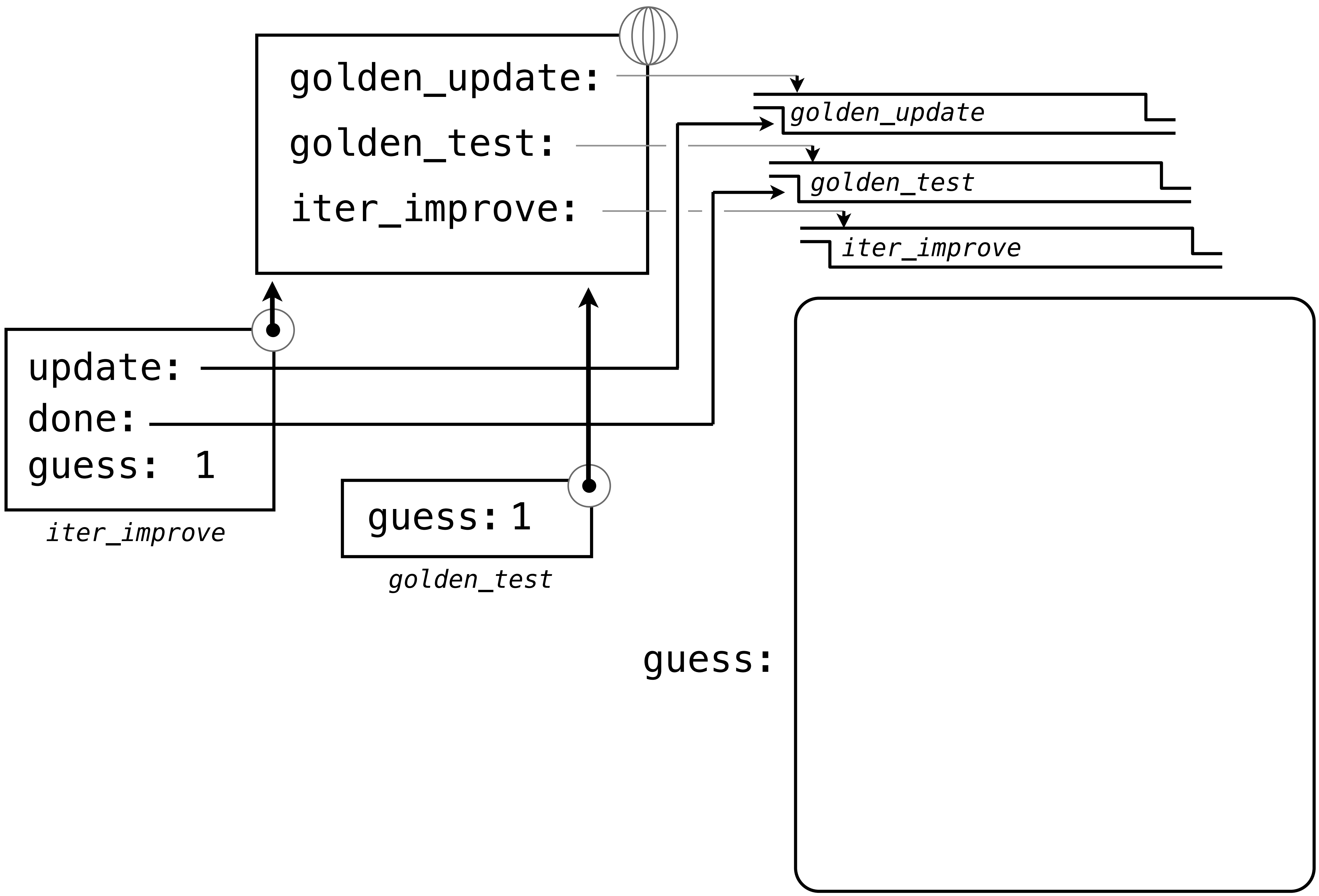
Iterative Improvement



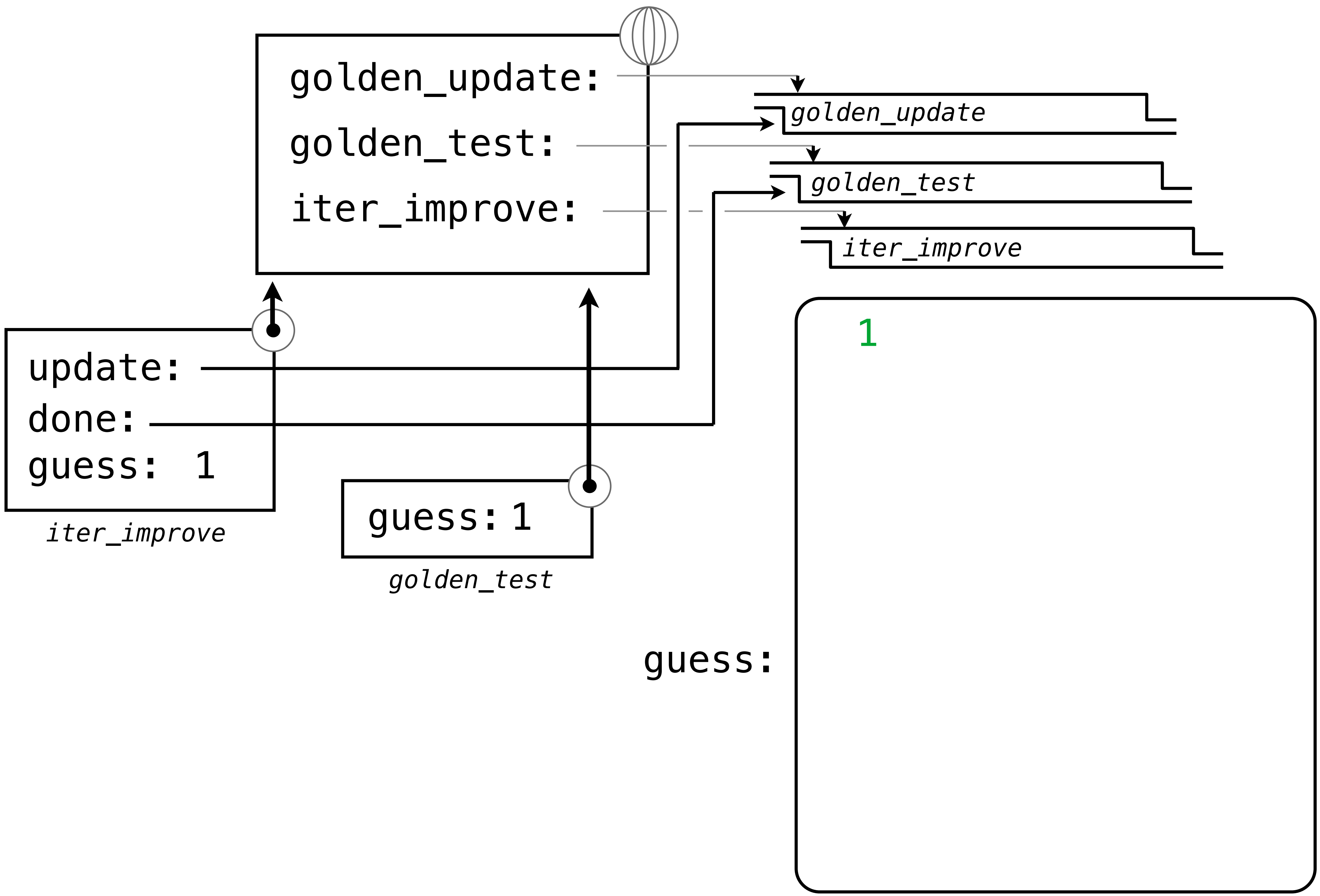
Iterative Improvement



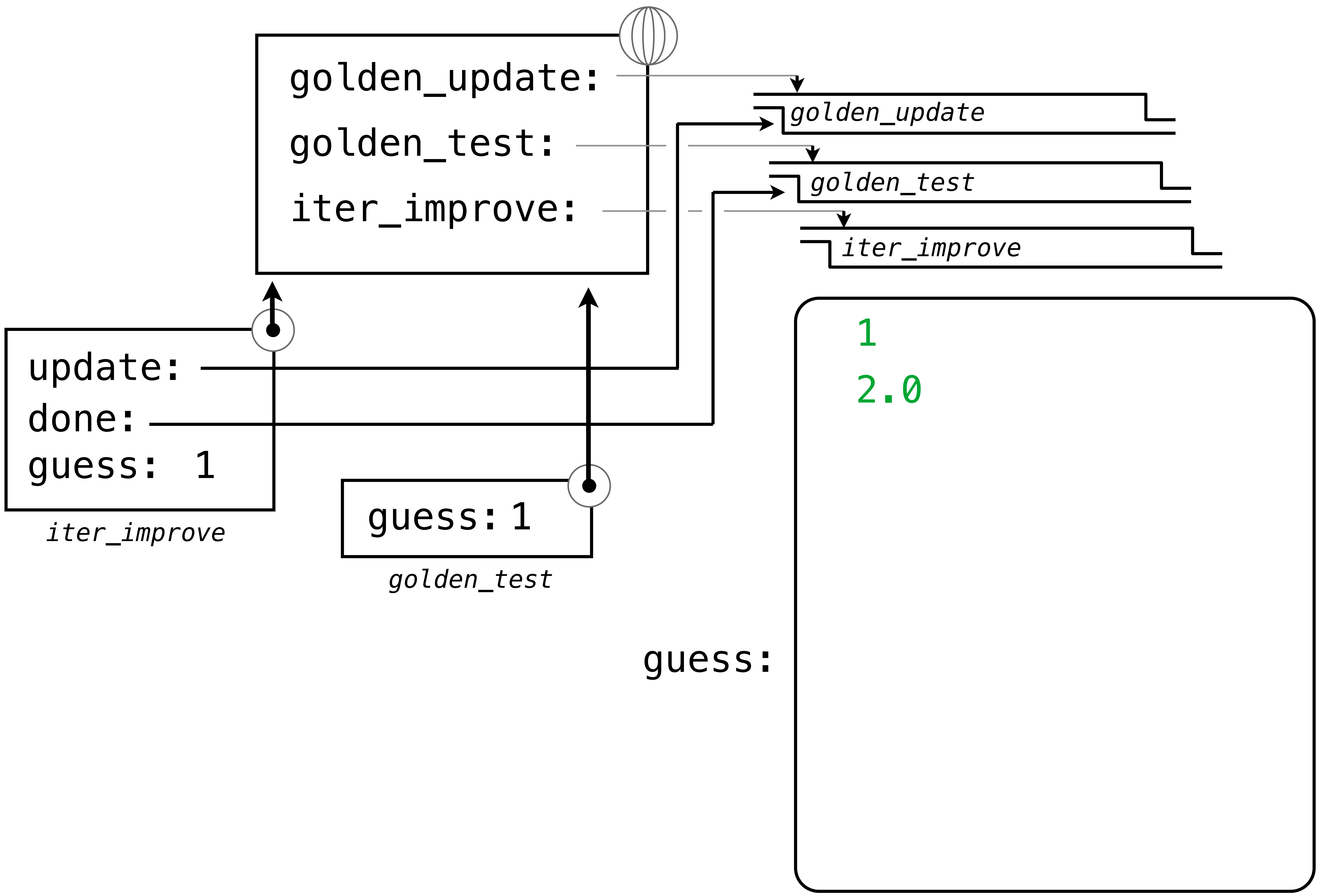
Iterative Improvement



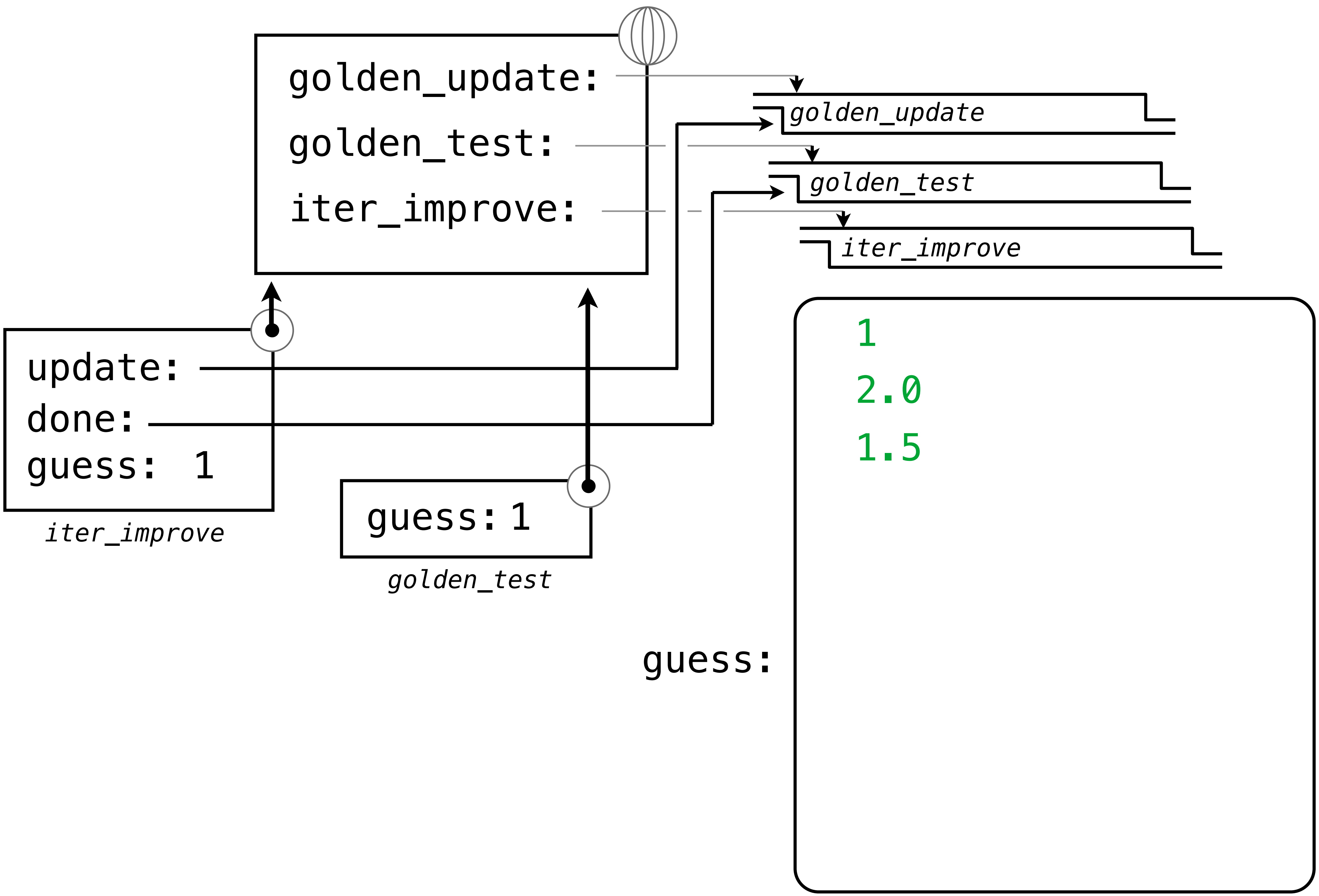
Iterative Improvement



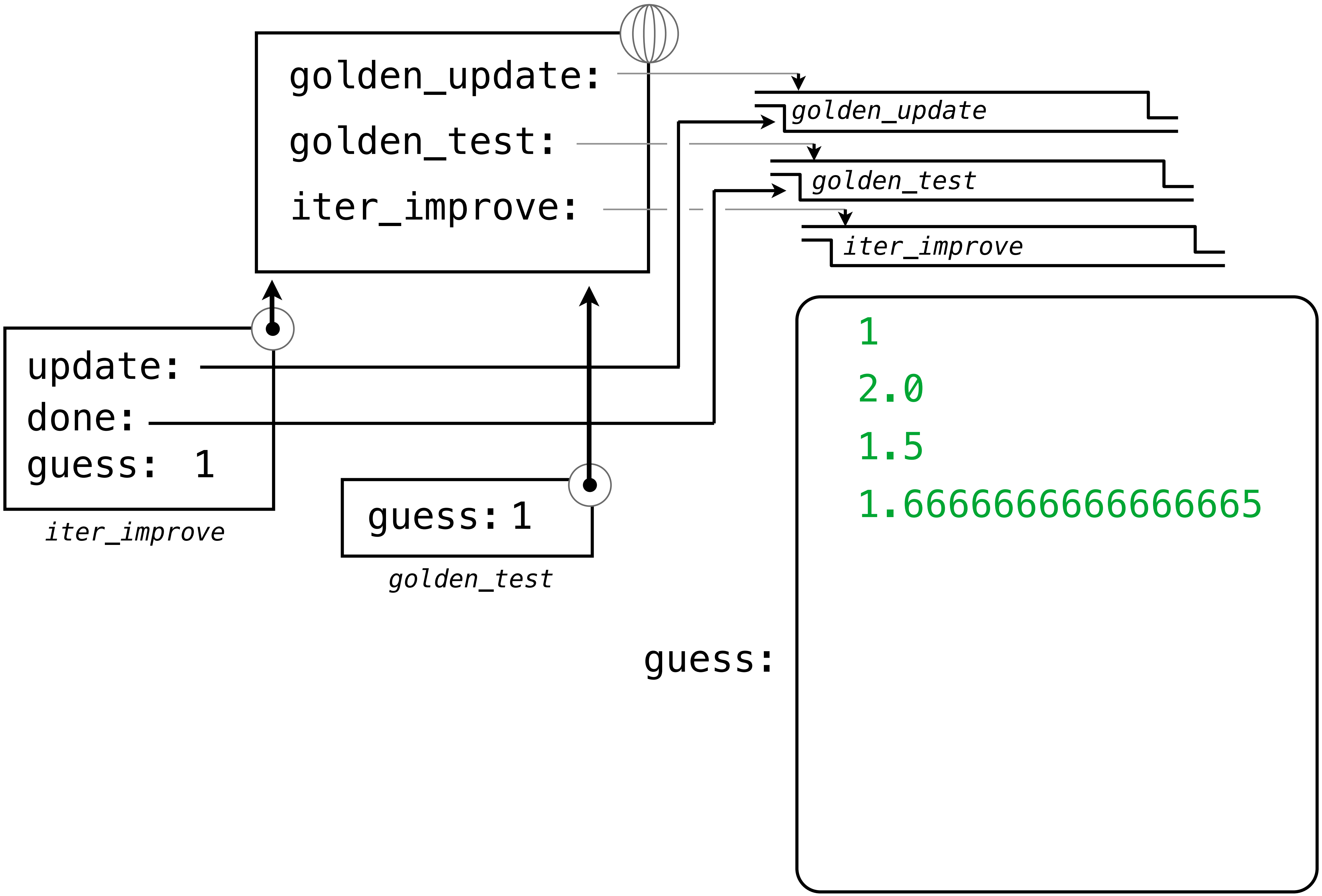
Iterative Improvement



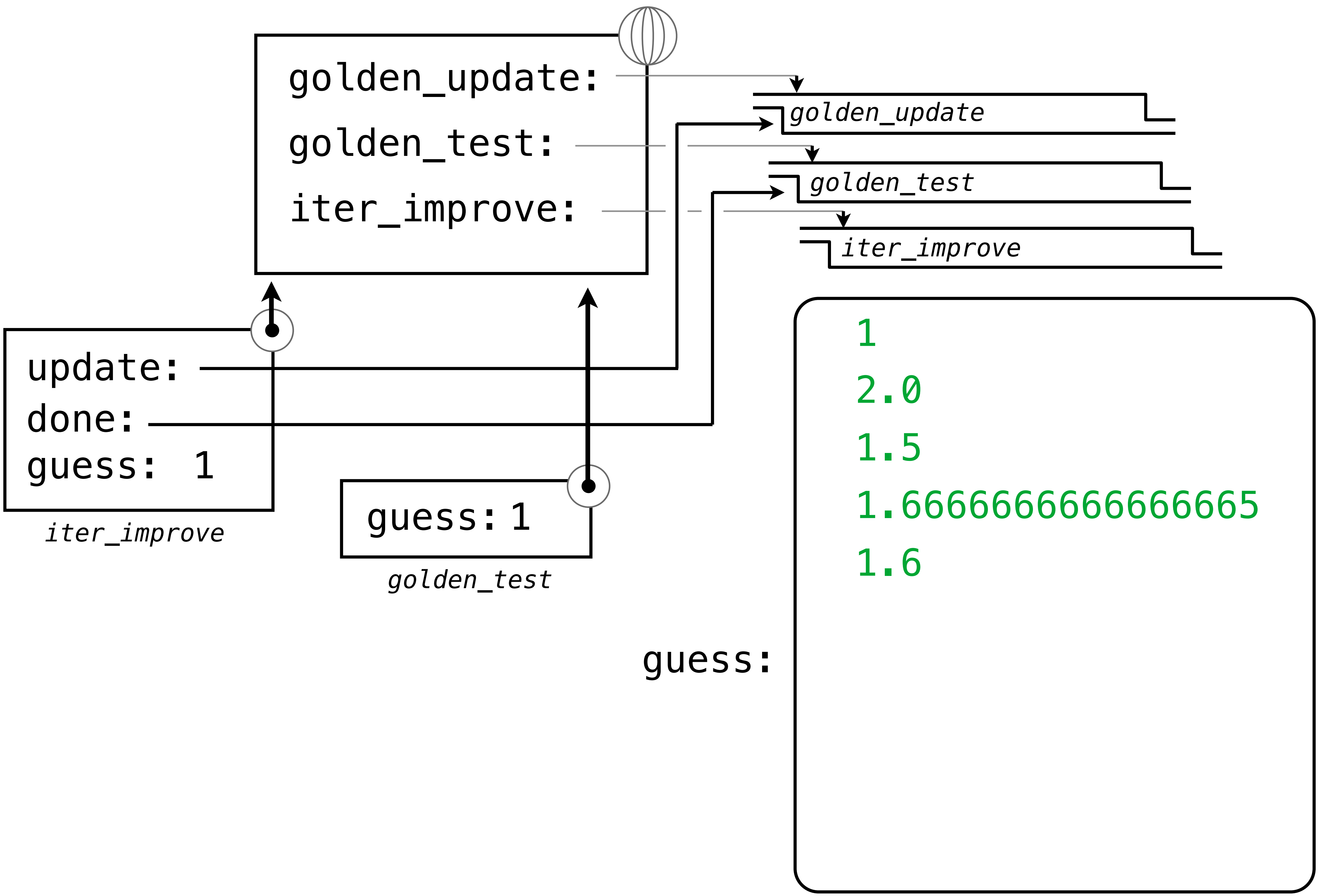
Iterative Improvement



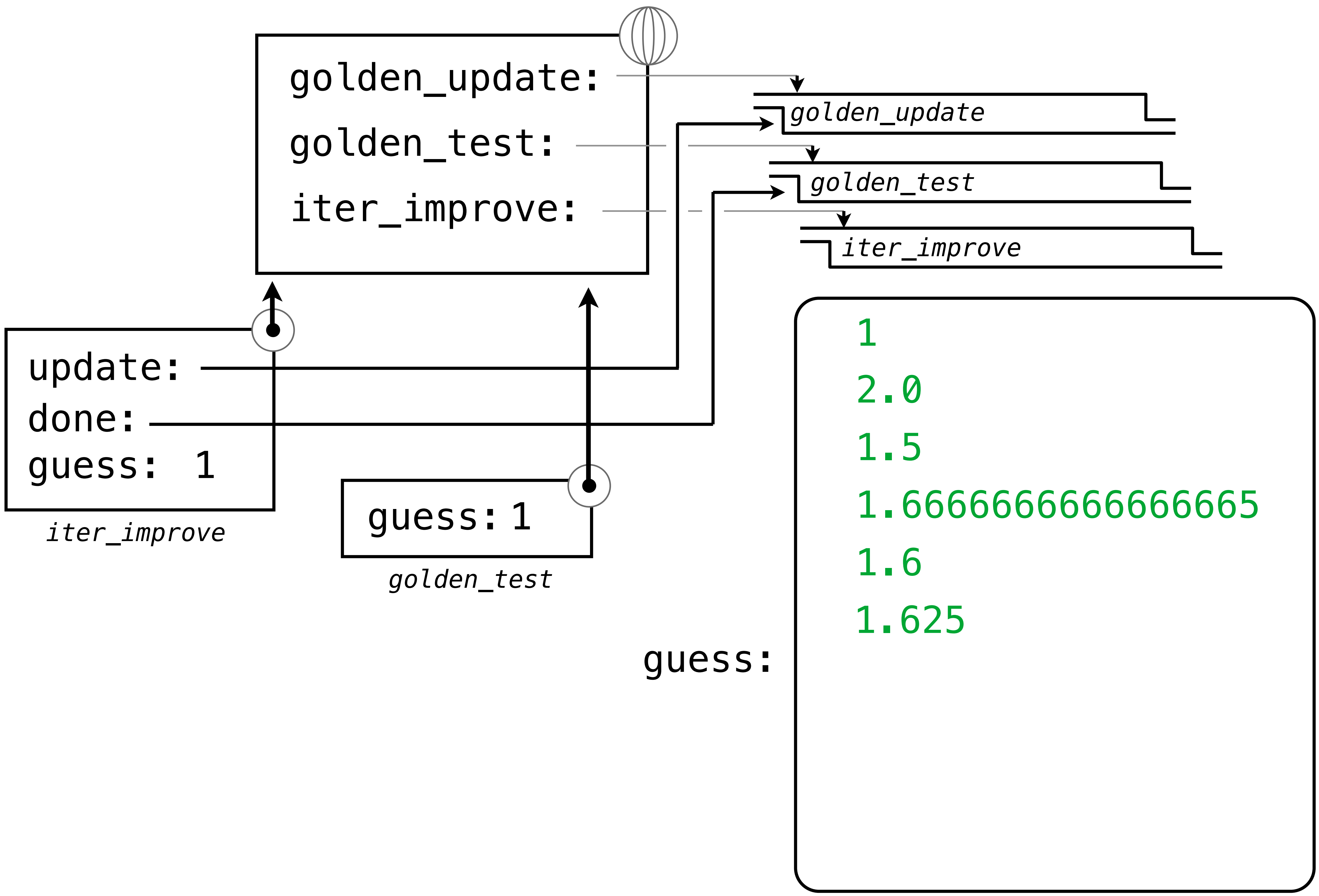
Iterative Improvement



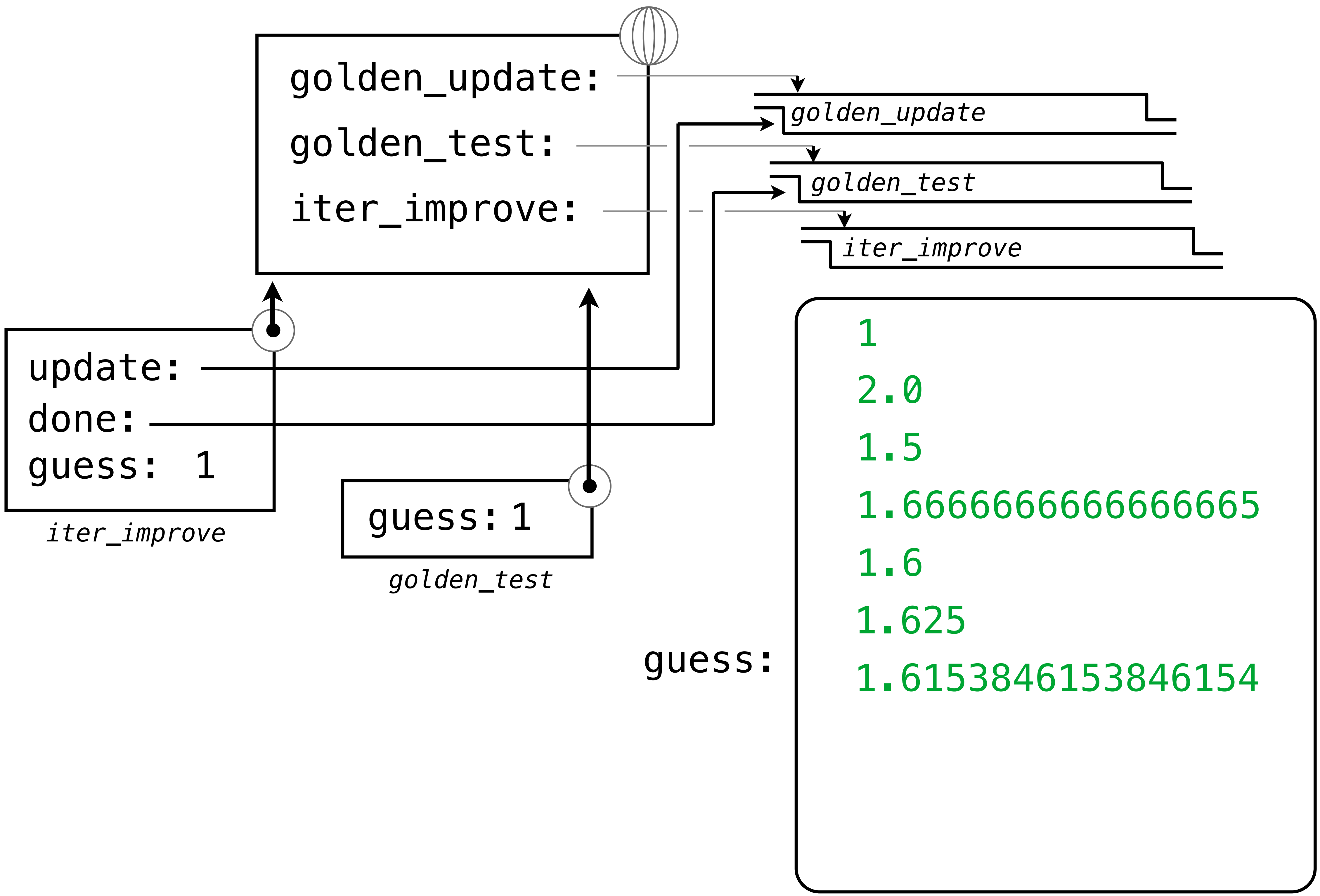
Iterative Improvement



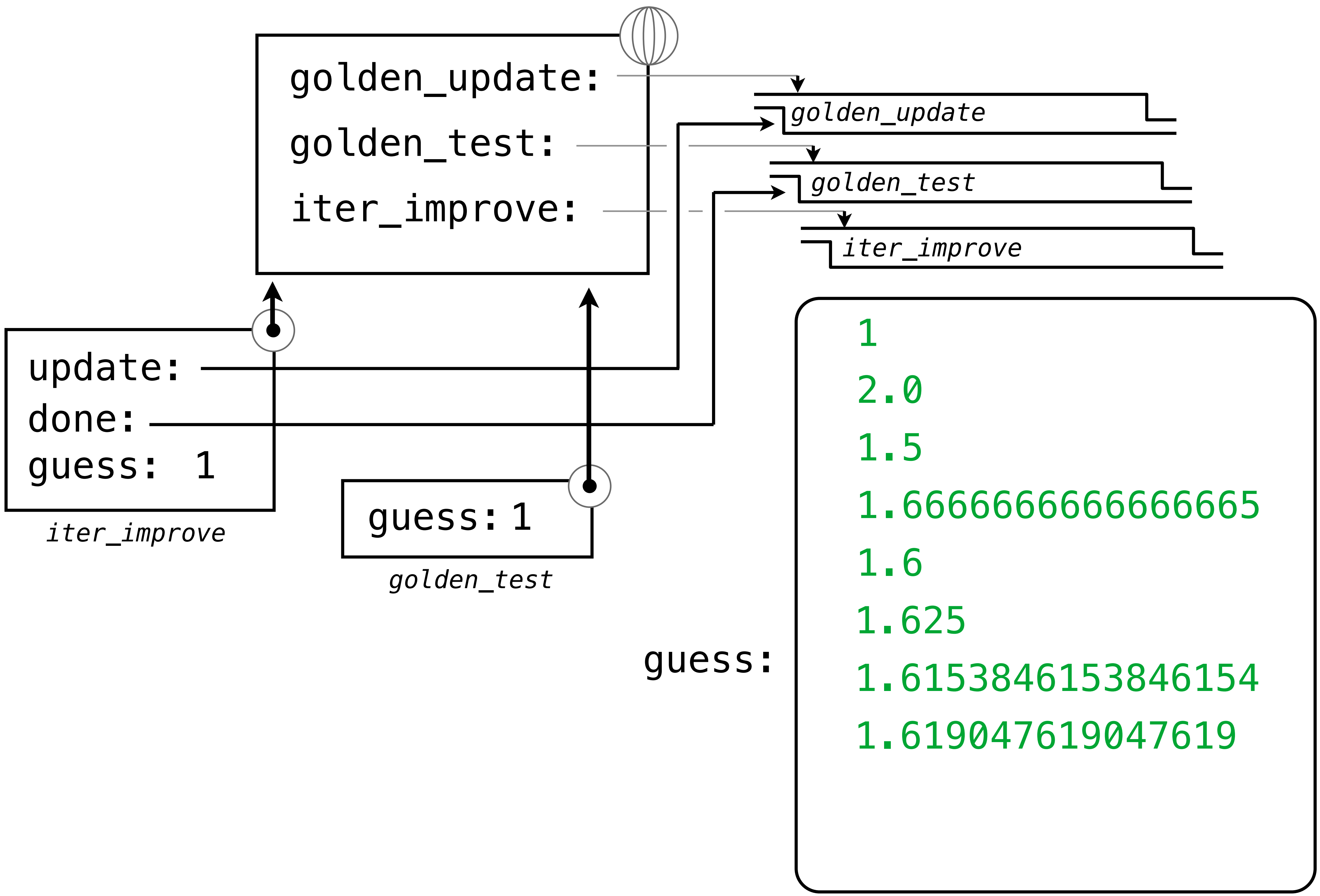
Iterative Improvement



Iterative Improvement

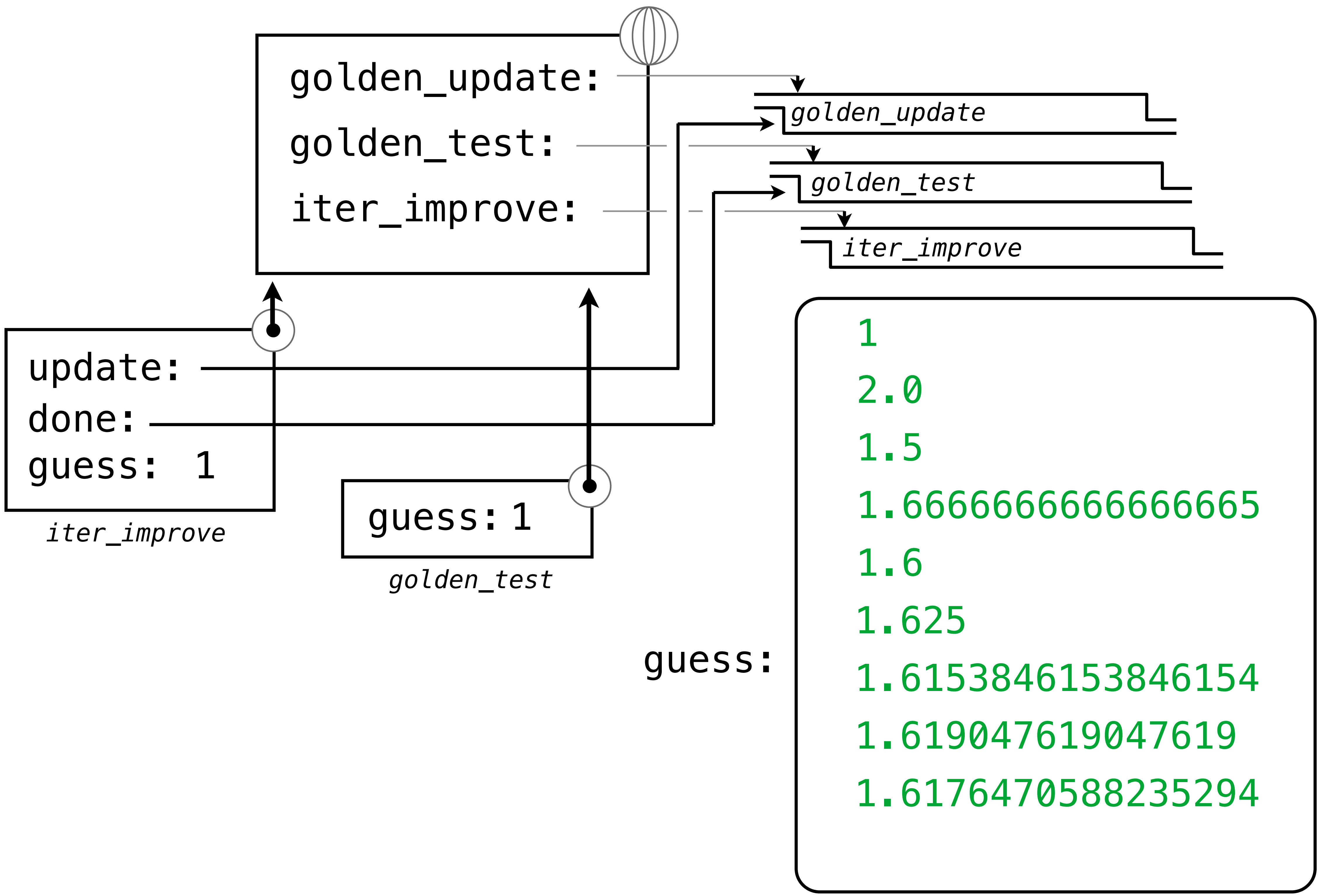


Iterative Improvement

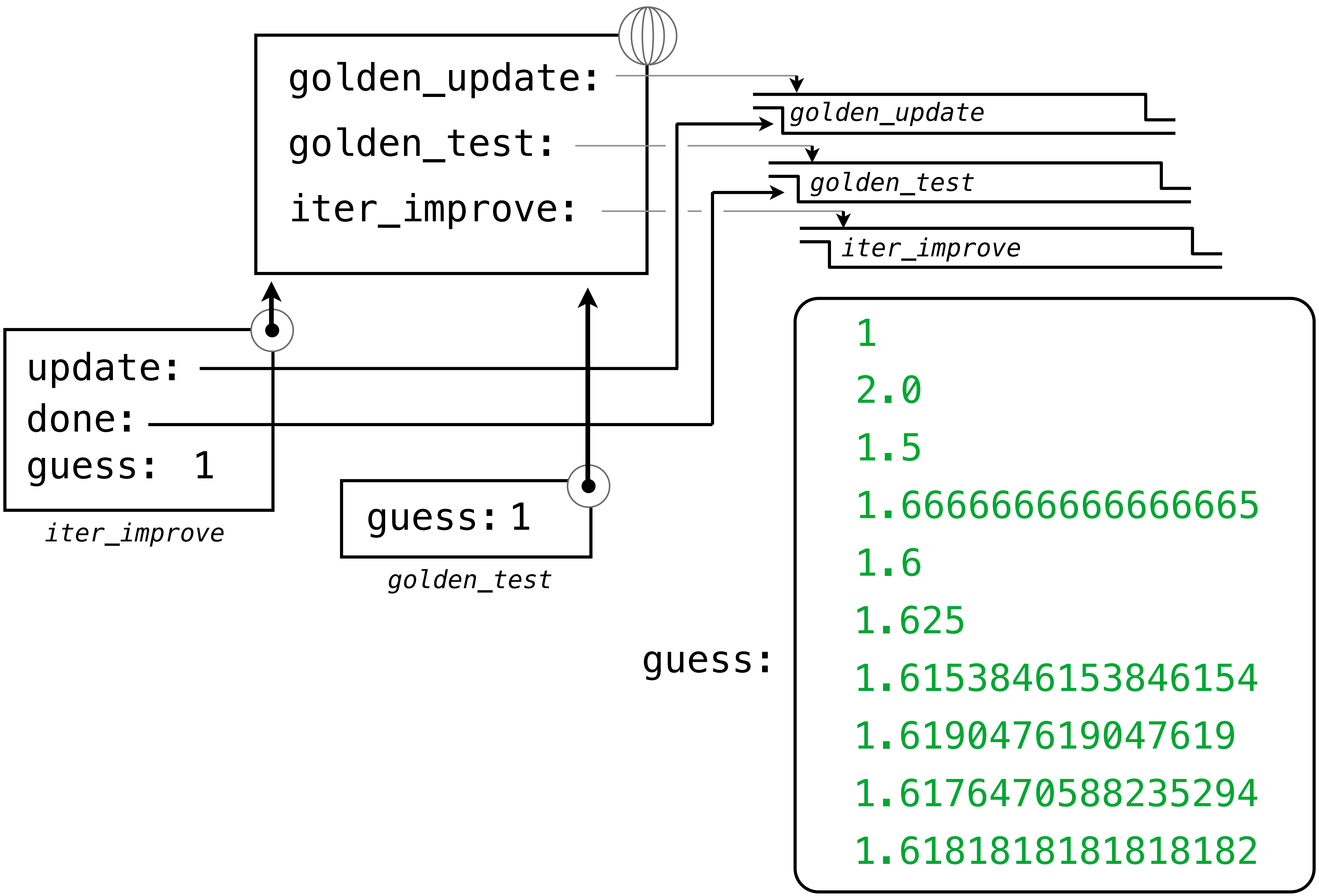


Friday, September 9, 2011

Iterative Improvement



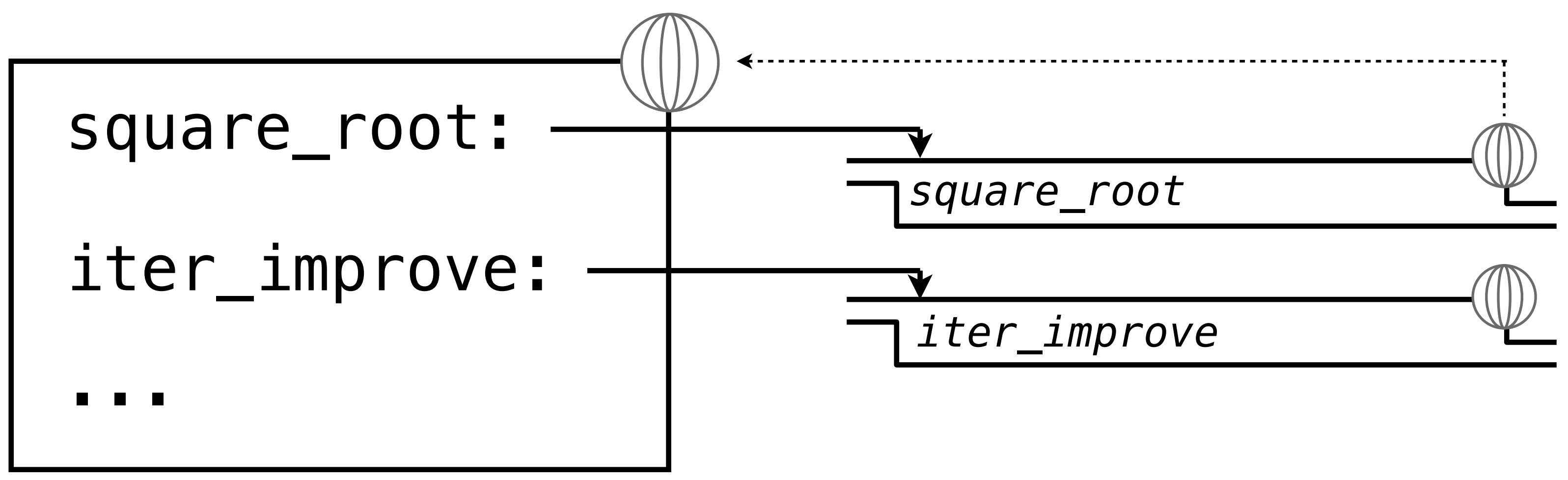
Iterative Improvement



Square Roots by Iterative Improvement

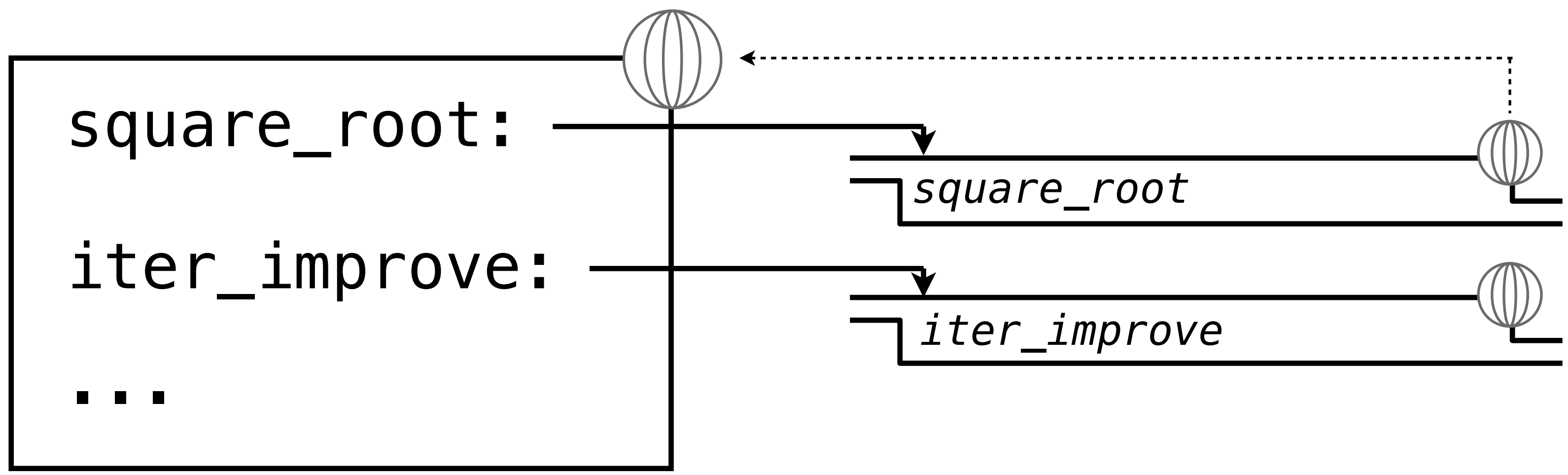
(Demo)

Square Roots by Iterative Improvement



(Demo)

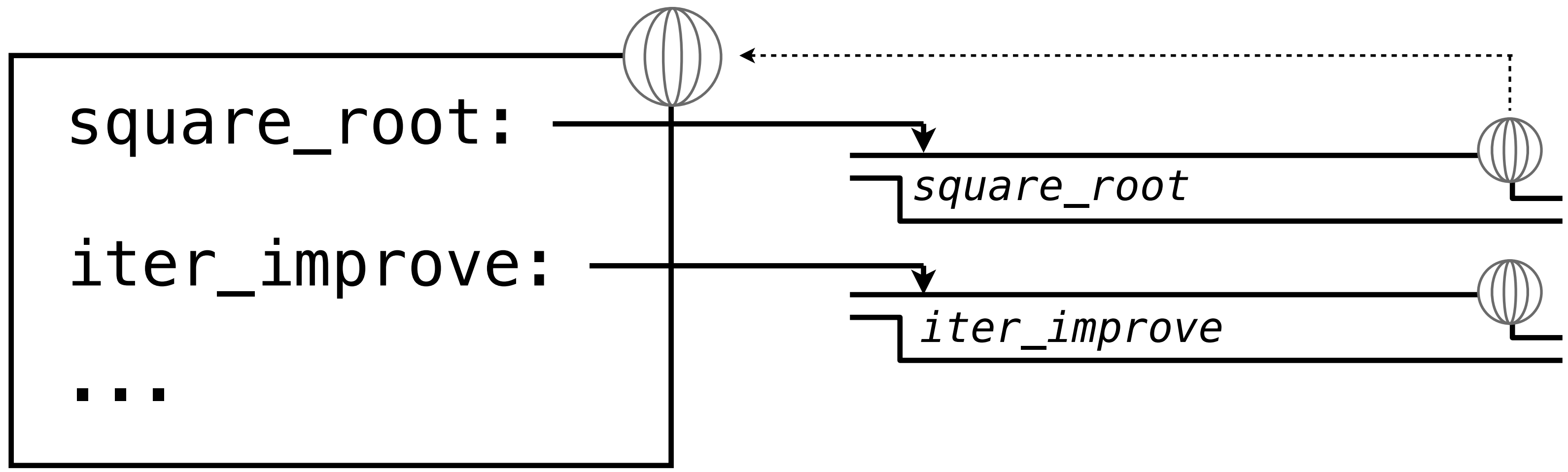
Square Roots by Iterative Improvement



(Demo)

square_root(256)

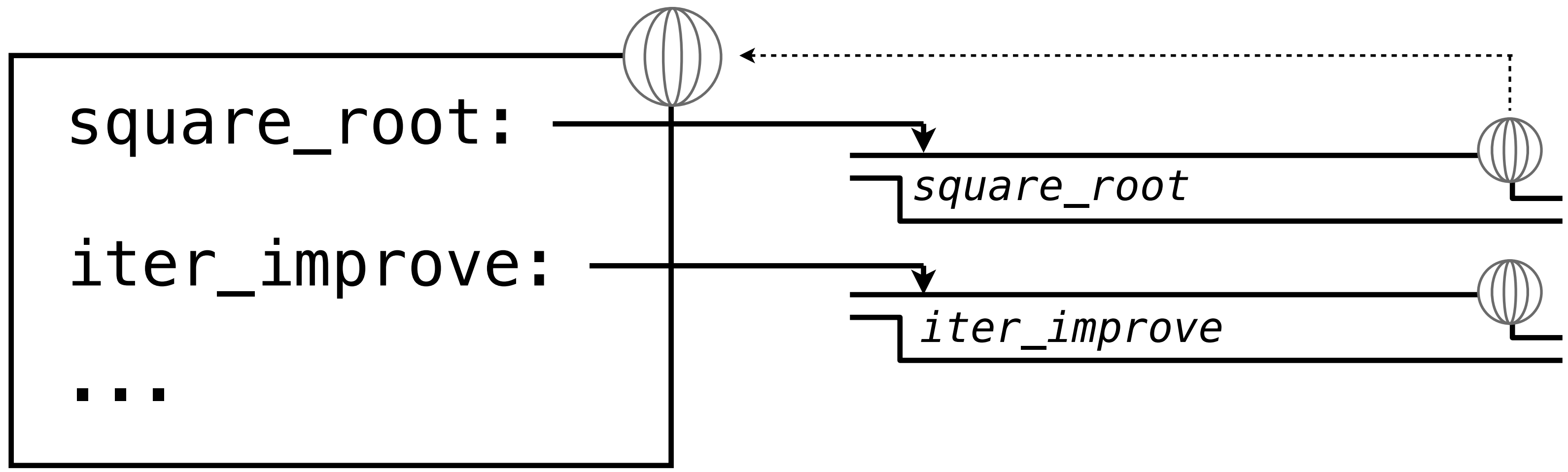
Square Roots by Iterative Improvement



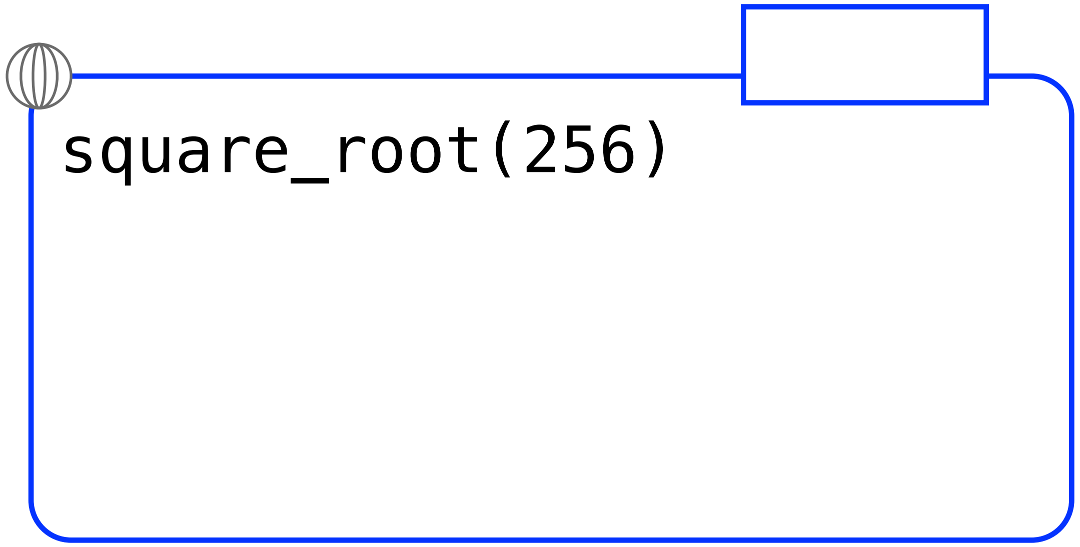
(Demo)

▶ square_root(256)

Square Roots by Iterative Improvement

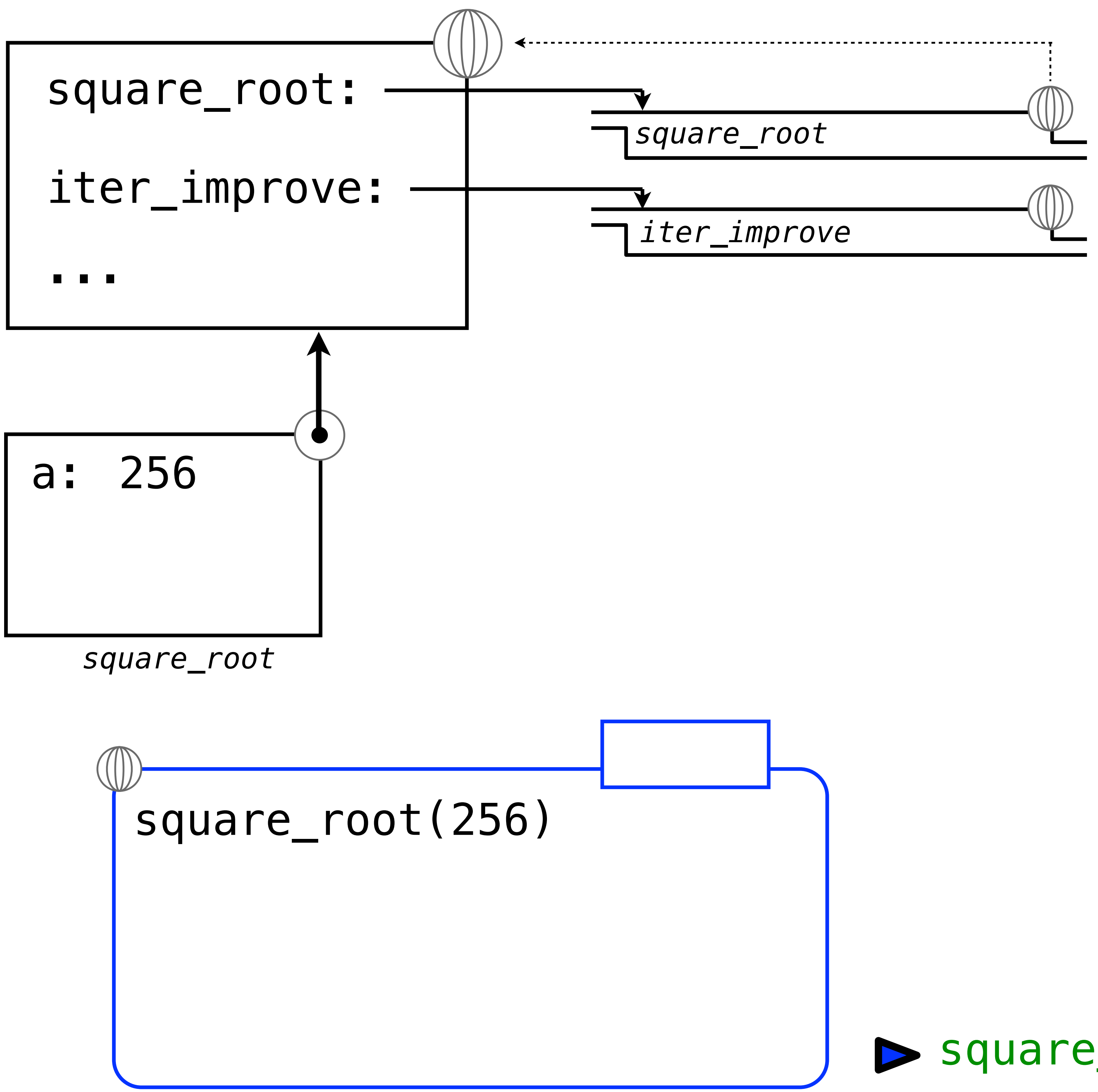


(Demo)



square_root(256)

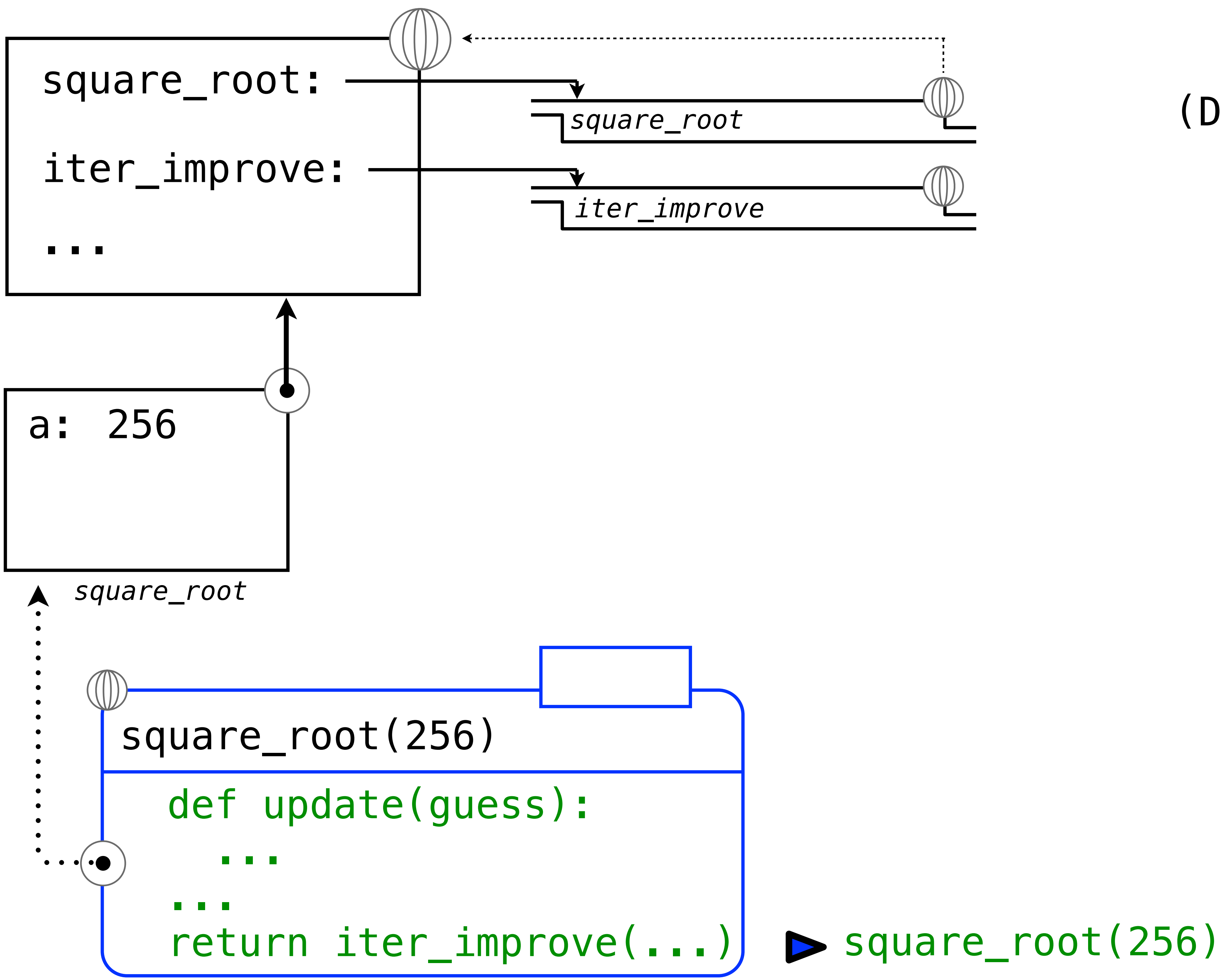
Square Roots by Iterative Improvement



(Demo)

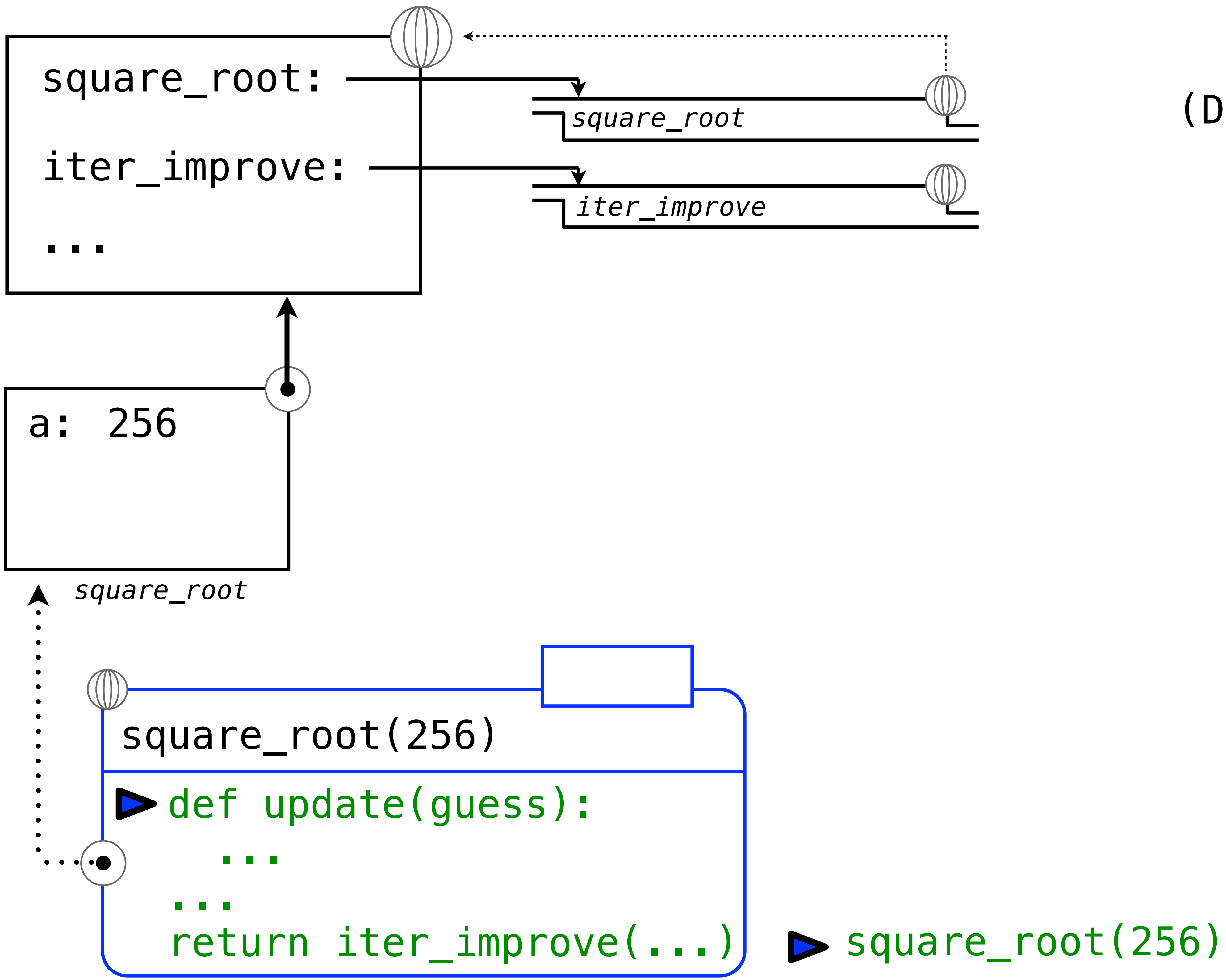
▶ square_root(256)

Square Roots by Iterative Improvement



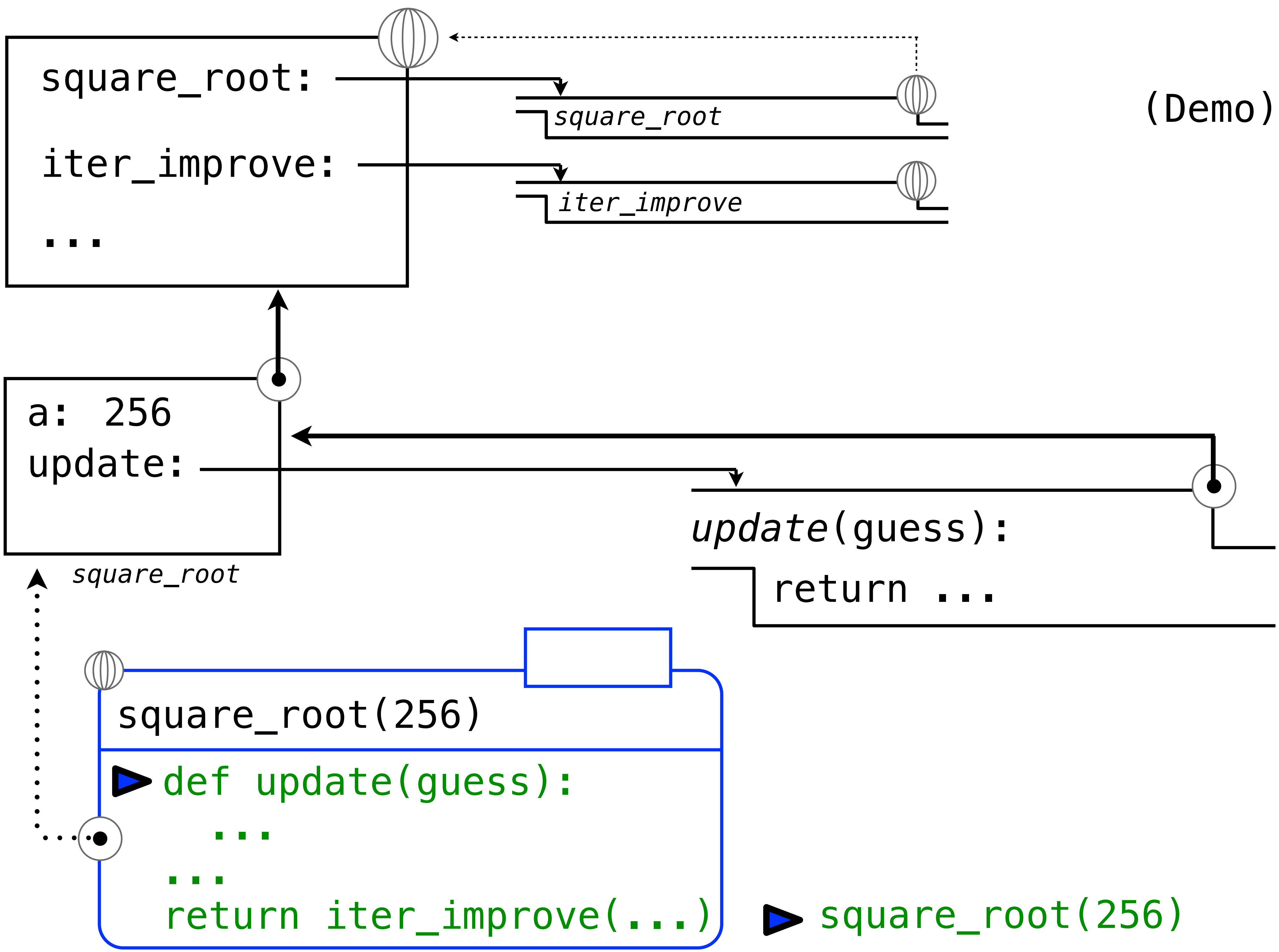
(Demo)

Square Roots by Iterative Improvement

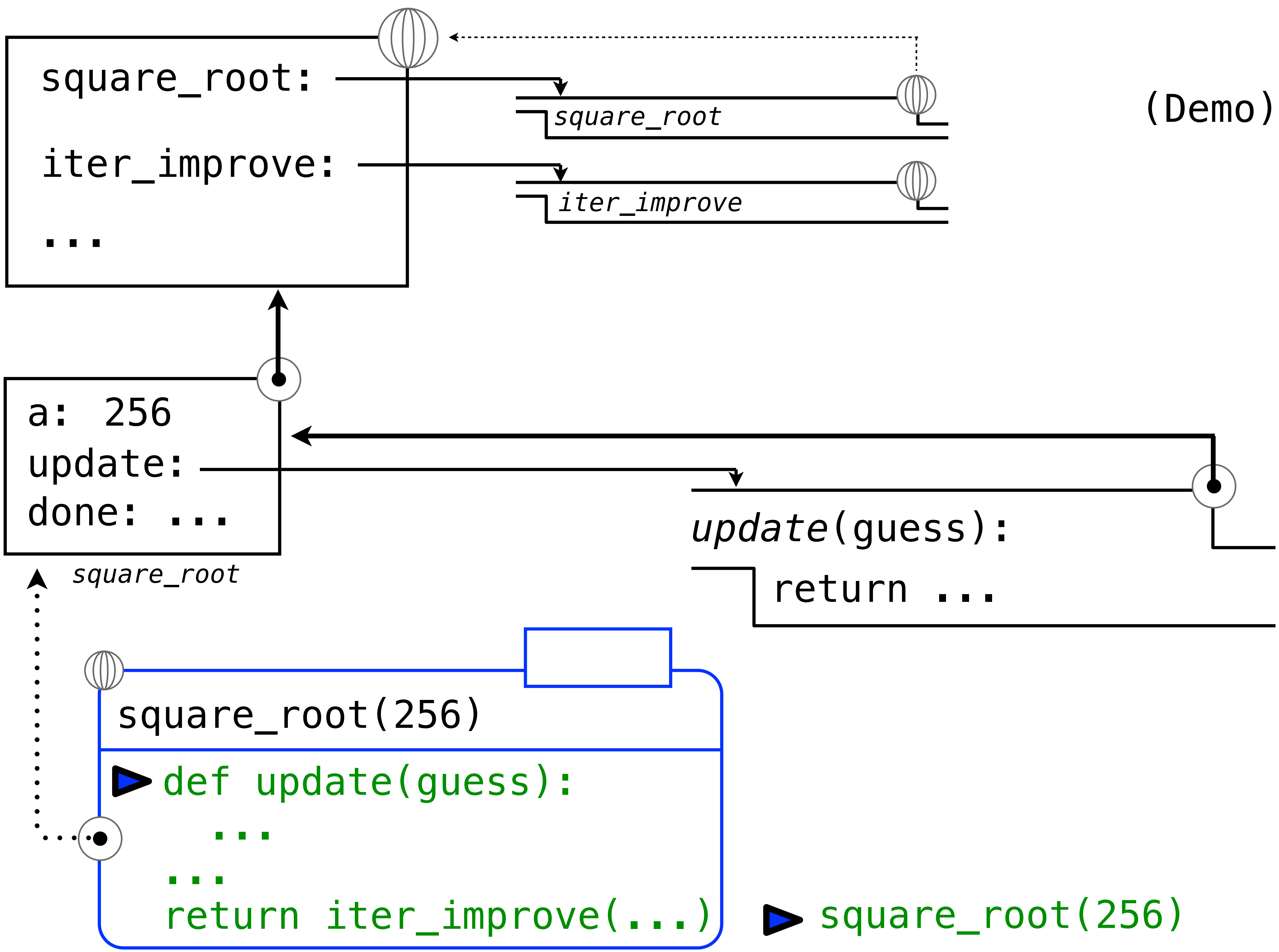


(Demo)

Square Roots by Iterative Improvement

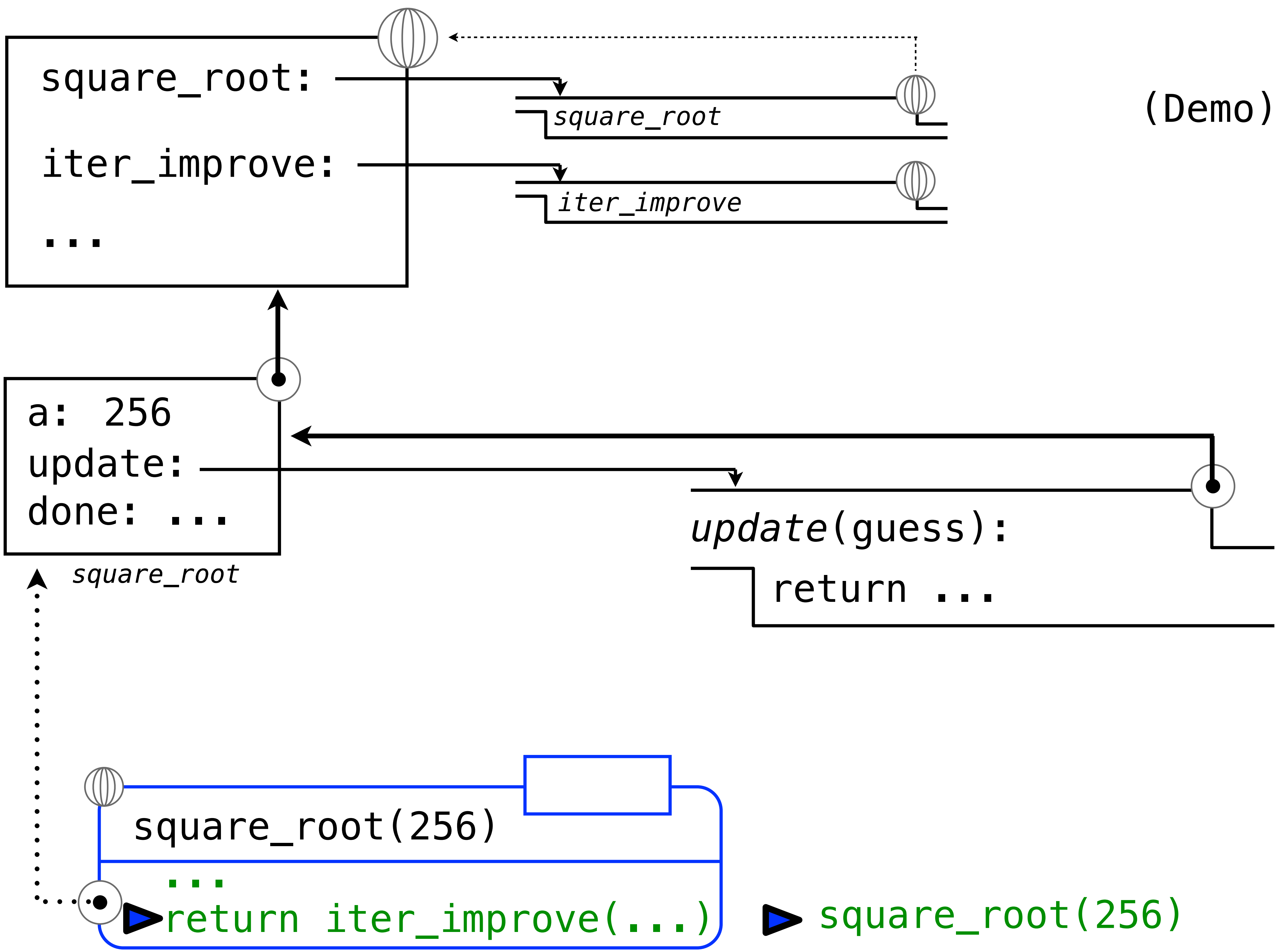


Square Roots by Iterative Improvement

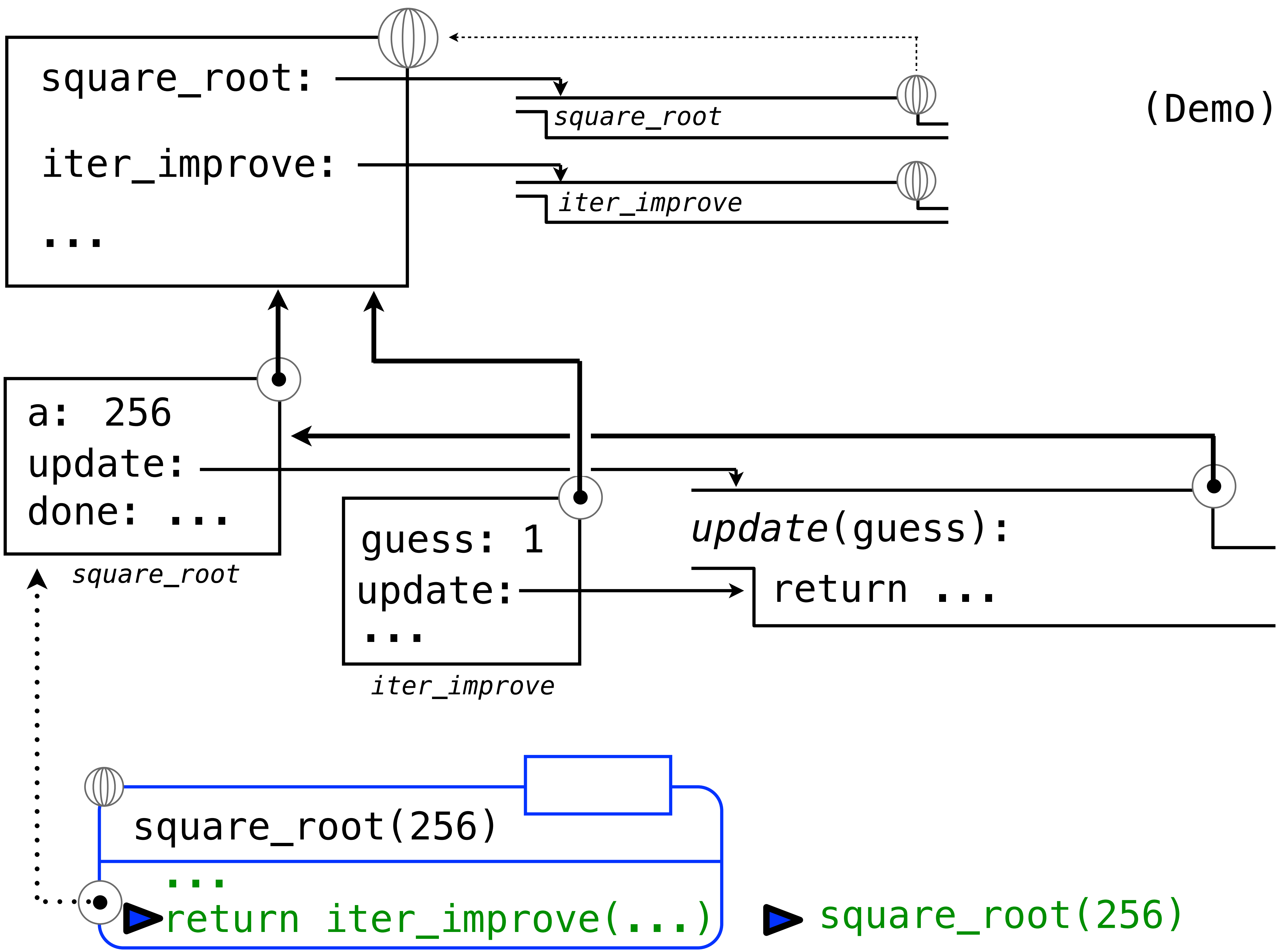


(Demo)

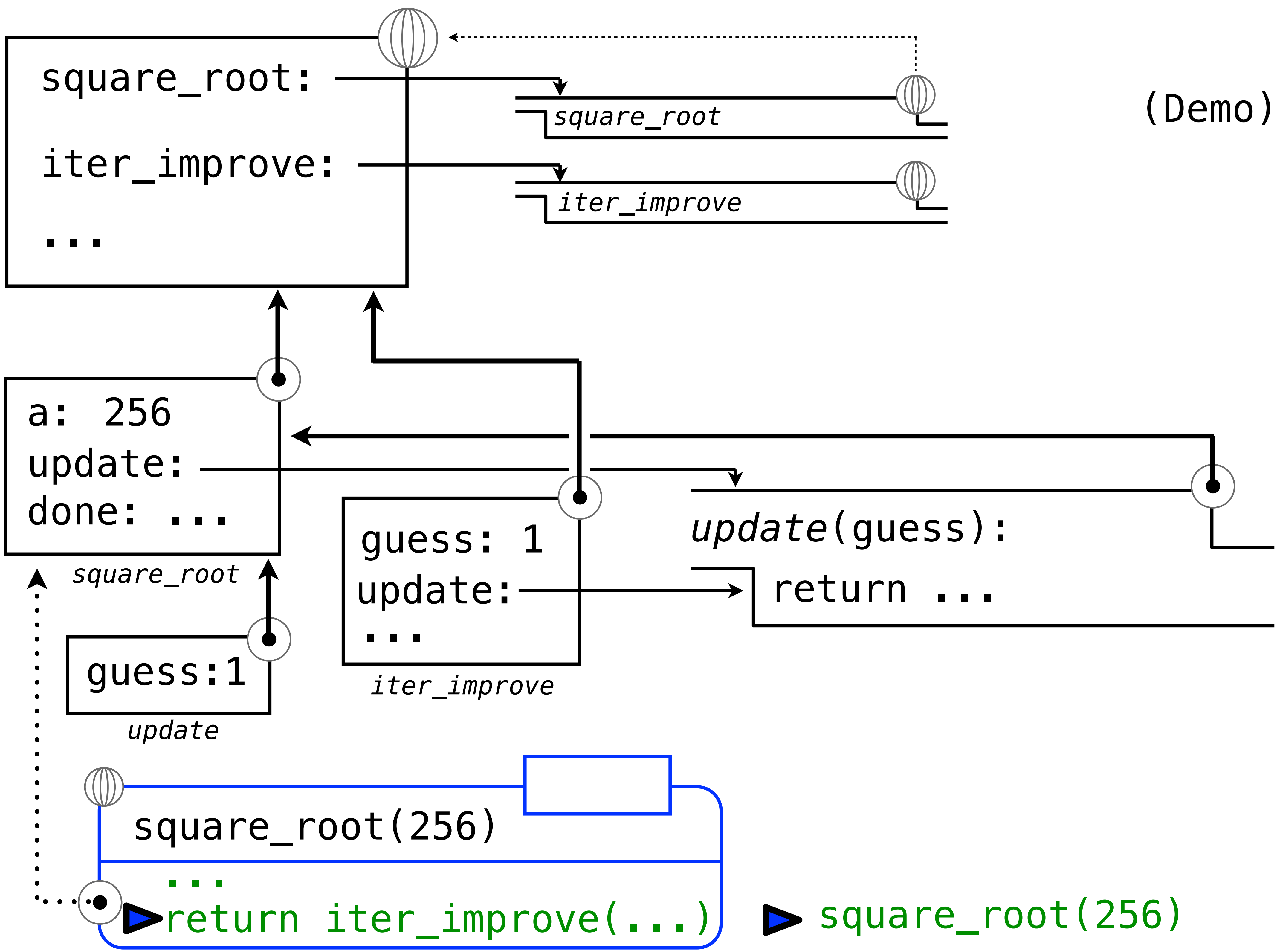
Square Roots by Iterative Improvement



Square Roots by Iterative Improvement



Square Roots by Iterative Improvement

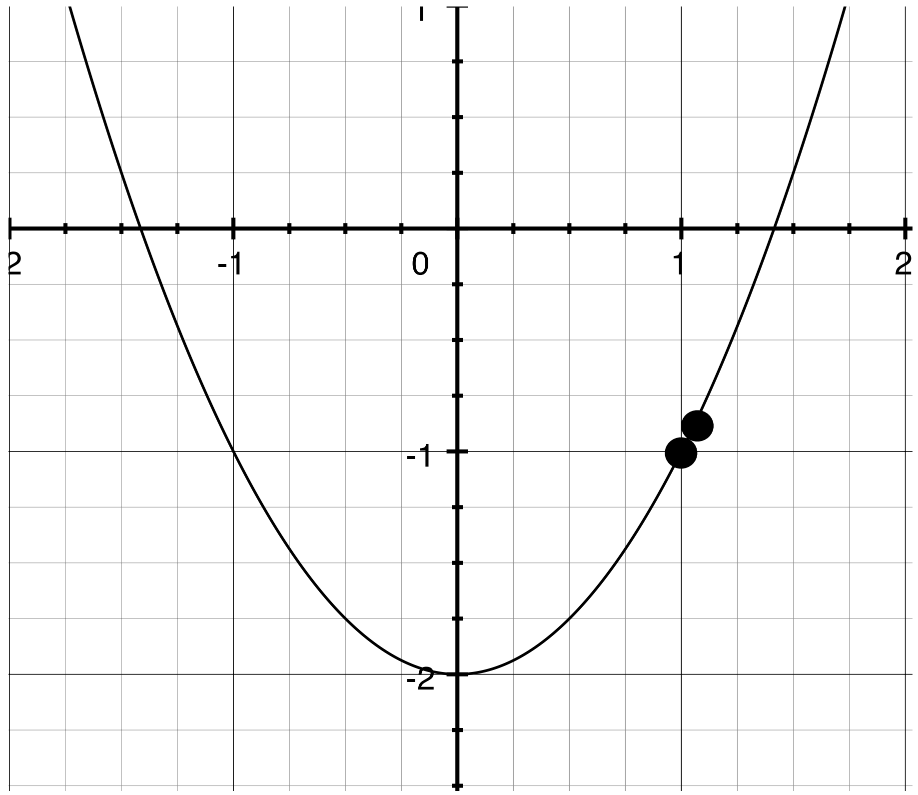


Derivatives of Single-Argument Functions

$$\texttt{f'(x)} = \lim_{\texttt{h}\to\texttt{0}} \frac{\texttt{f(x+h) - f(x)}}{\texttt{h}}$$

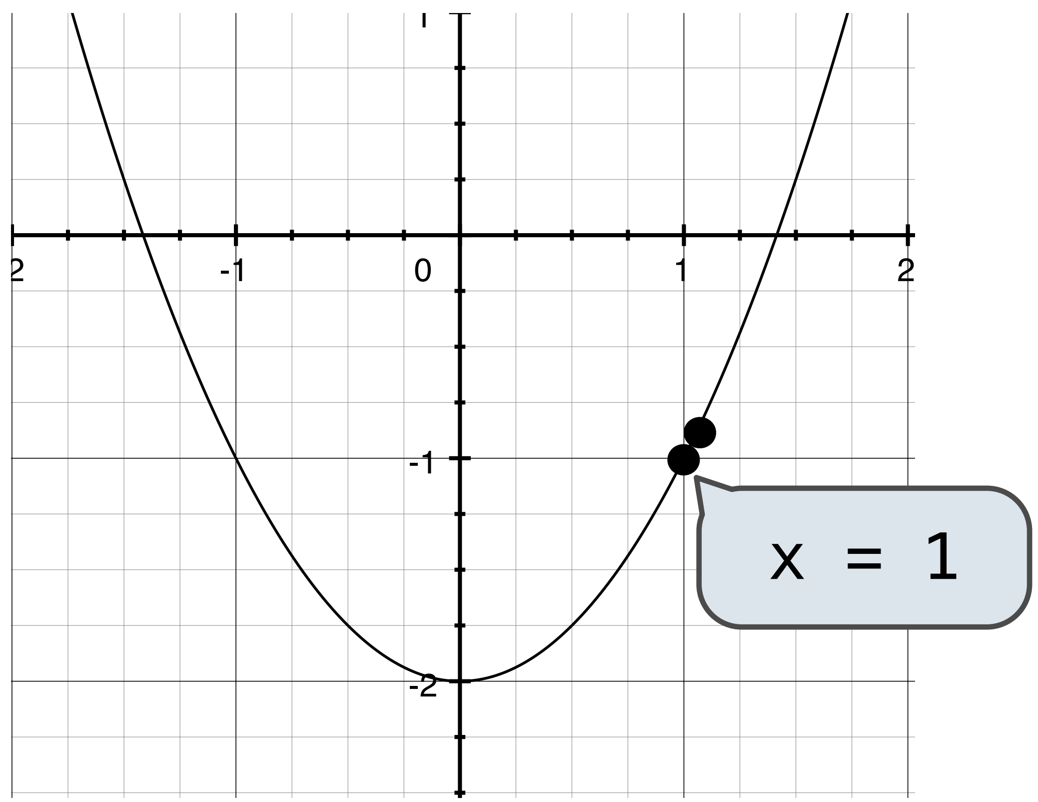
Derivatives of Single-Argument Functions

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$



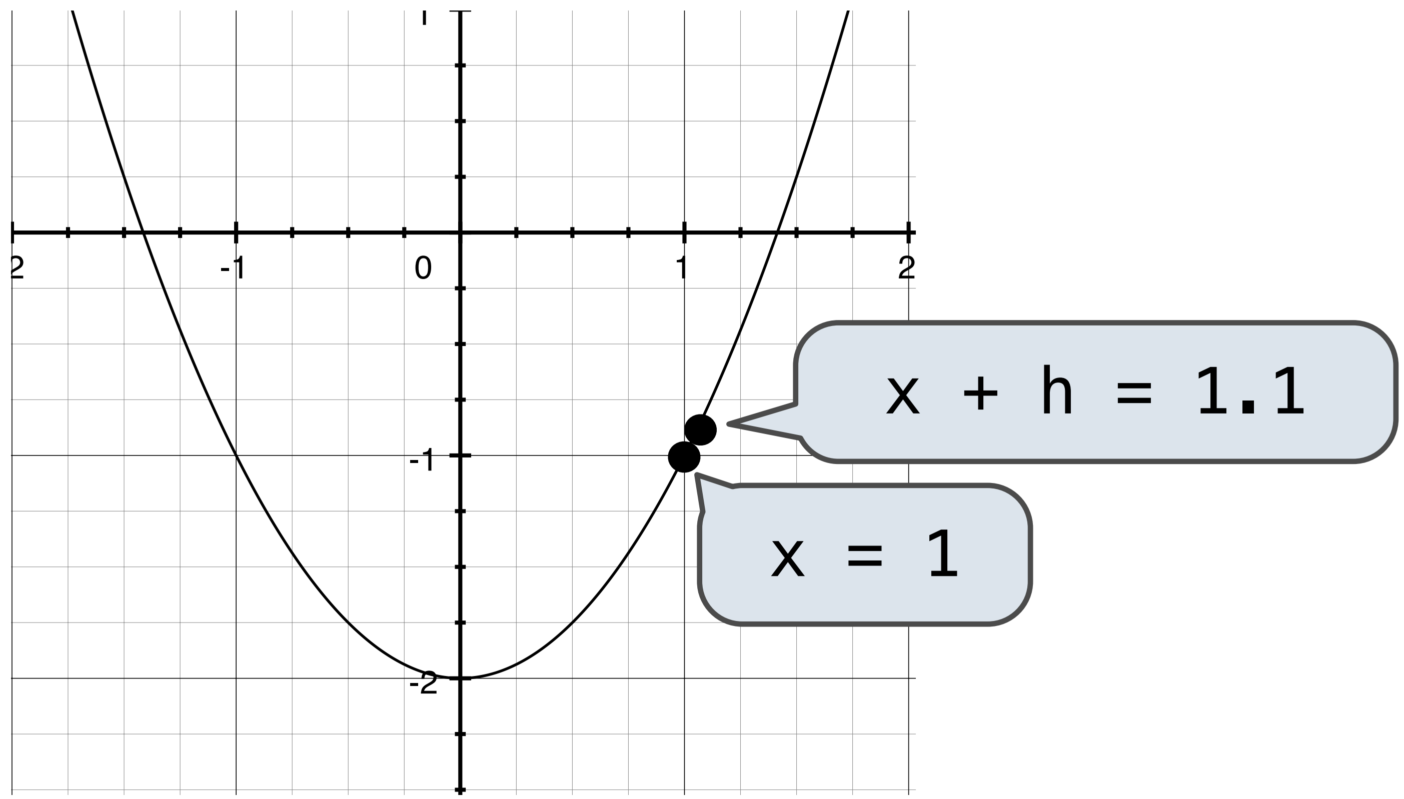
Derivatives of Single-Argument Functions

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$



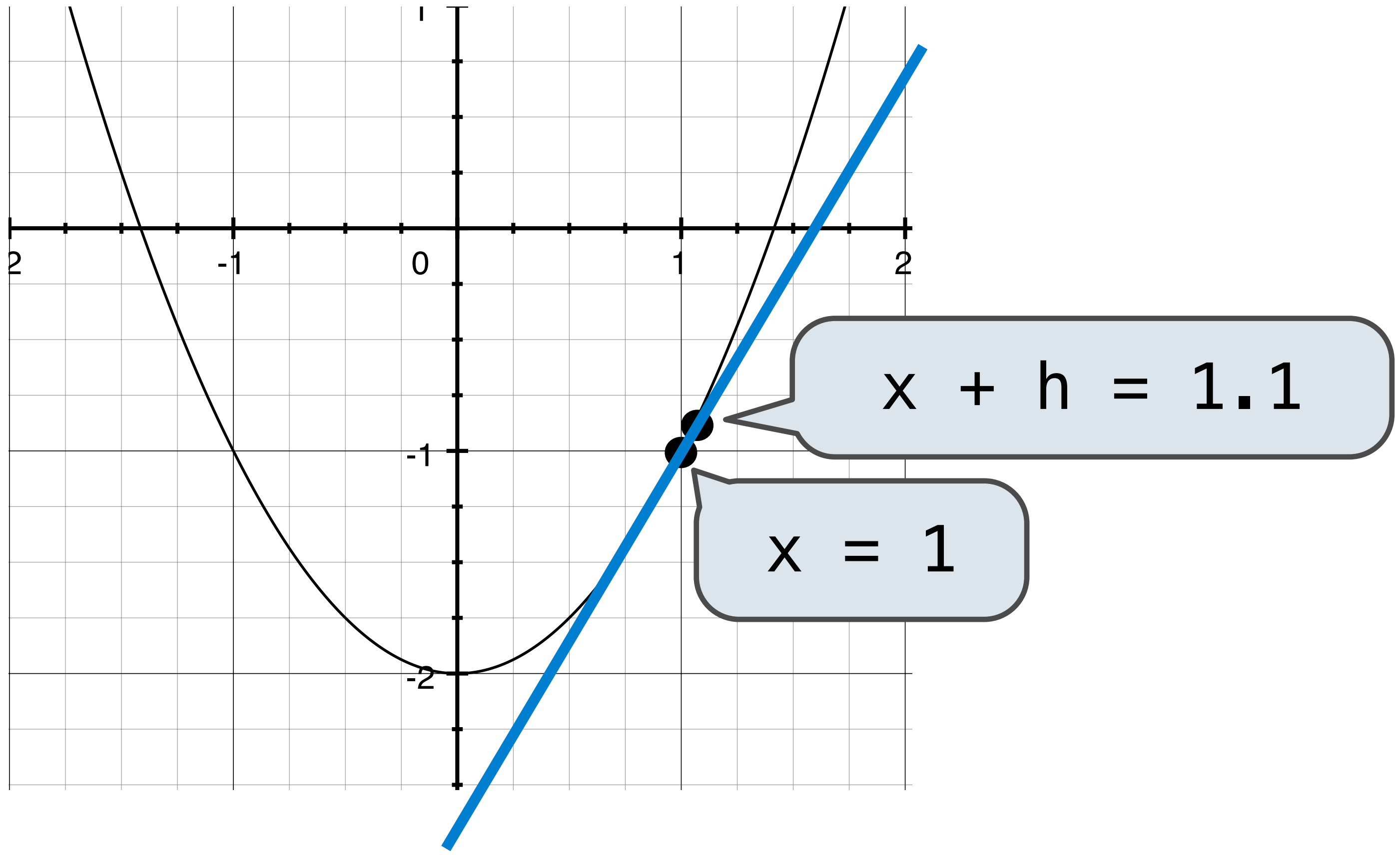
Derivatives of Single-Argument Functions

$$\text{f'(x)} = \lim_{\text{h}\to 0} \frac{\text{f(x+h) - f(x)}}{\text{h}}$$



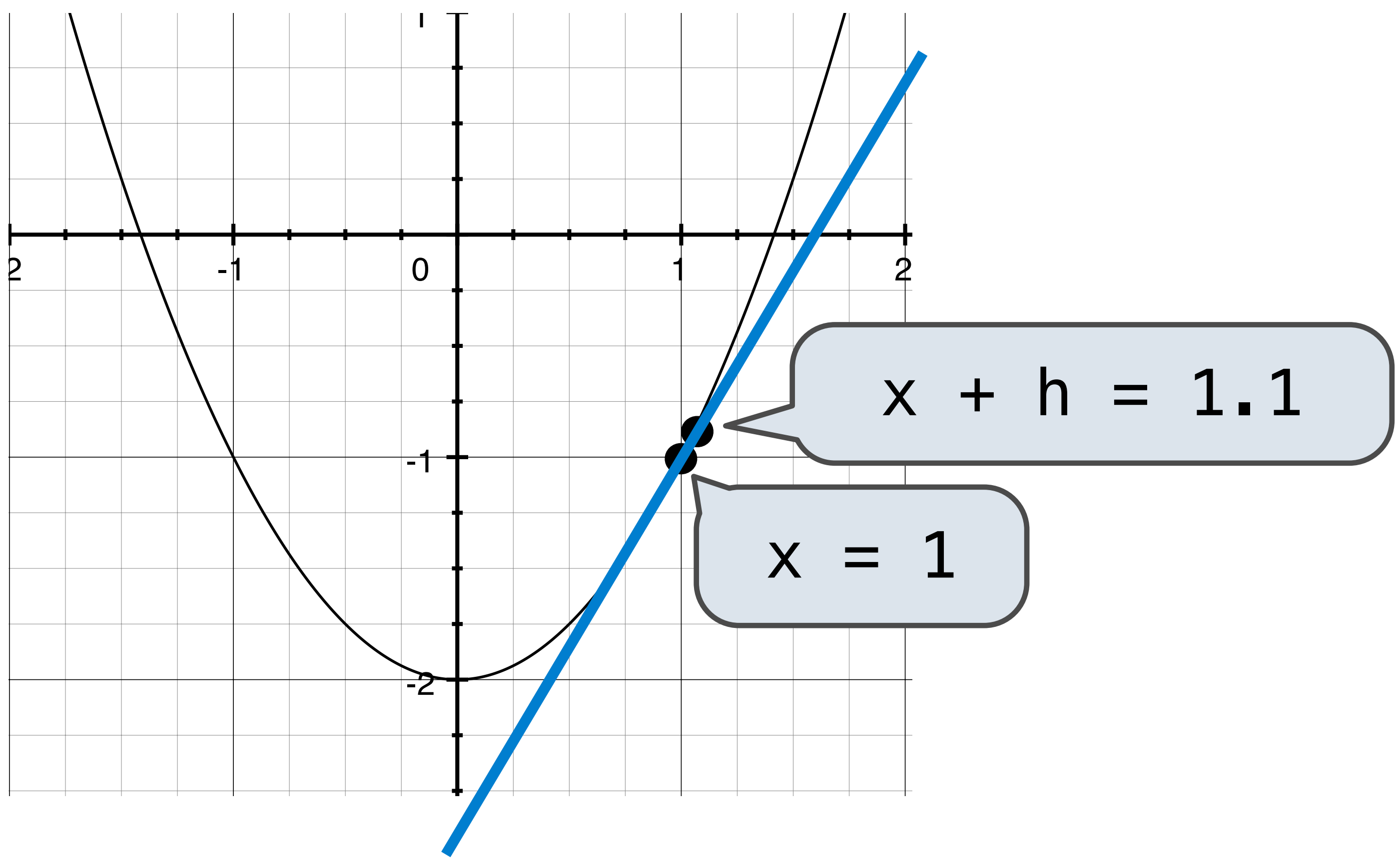
Derivatives of Single-Argument Functions

$$\texttt{f'(x)} = \lim_{\texttt{h}\to\texttt{0}} \frac{\texttt{f(x+h) - f(x)}}{\texttt{h}}$$



Derivatives of Single-Argument Functions

$$\texttt{f'(x)} = \lim_{\texttt{h}\to\texttt{0}} \frac{\texttt{f(x+h) - f(x)}}{\texttt{h}}$$



(Demo)

http://en.wikipedia.org/wiki/File:Graph_of_sliding_derivative_line.gif

Approximating Derivatives

`(Demo)`

Implementing Newton's Method

Implementing Newton's Method

```
def newton_update(f):
    """Return an update function for f using Newton's method."""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update
```

Implementing Newton's Method

```
def newton_update(f):
    """Return an update function for f using Newton's method."""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update
```

Could be replaced with the exact derivative

Implementing Newton's Method

```
def newton_update(f):
    """Return an update function for f using Newton's method."""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update
```

Could be replaced with the exact derivative

```
def approx_derivative(f, x, delta=1e-5):
    """Return an approximation to the derivative of f at x."""
    df = f(x + delta) - f(x)
    return df/delta
```

Implementing Newton's Method

```
def newton_update(f):
    """Return an update function for f using Newton's method."""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update
```

Could be replaced with the exact derivative

```
def approx_derivative(f, x, delta=1e-5):
    """Return an approximation to the derivative of f at x."""
    df = f(x + delta) - f(x)
    return df/delta
```

Limit approximated by a small value

Implementing Newton's Method

```
def newton_update(f):
    """Return an update function for f using Newton's method."""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update
```

Could be replaced with the exact derivative

```
def approx_derivative(f, x, delta=1e-5):
    """Return an approximation to the derivative of f at x."""
    df = f(x + delta) - f(x)
    return df/delta
```

Limit approximated by a small value

```
def find_root(f, guess=1):
    """Return a guess of a zero of the function f, near guess.

    >>> from math import sin
    >>> find_root(lambda y: sin(y), 3)
    3.141592653589793
    """
    return iter_improve(newton_update(f), lambda x: f(x) == 0, guess)
```

Implementing Newton's Method

```
def newton_update(f):
    """Return an update function for f using Newton's method."""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update
```

Could be replaced with the exact derivative

```
def approx_derivative(f, x, delta=1e-5):
    """Return an approximation to the derivative of f at x."""
    df = f(x + delta) - f(x)
    return df/delta
```

Limit approximated by a small value

```
def find_root(f, guess=1):
    """Return a guess of a zero of the function f, near guess.

    >>> from math import sin
    >>> find_root(lambda y: sin(y), 3)
    3.141592653589793
    """
    return iter_improve(newton_update(f), lambda x: f(x) == 0, guess)
```

Definition of a function zero