Announcements

- HW3 out, due Tuesday at 7pm
- Midterm next Wednesday at 7pm
  - Keep an eye out for your assigned location
  - Old exams posted soon
- Review sessions
  - Saturday 2-4pm in TBA
  - Extend office hours Sunday 11-3pm in TBA
  - HKN review session Sunday 3-6pm in 145 Dwinelle
- Environment diagram handout on website
- Code review system online
  - See Piazza post for details
How to Draw an Environment Diagram
How to Draw an Environment Diagram

When defining a function:
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature

\(<\text{name}>\)(\text{<formal parameters>})
When defining a function:

Create a function value with signature
\(<\text{name}>\)(\text{<formal parameters>})

For nested definitions, label the parent as the first frame of the current environment
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
<name>(<formal parameters>)

For nested definitions, label the parent as the first frame of the current environment

Bind <name> to the function value in the first frame of the current environment
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
\texttt{<name> (<formal parameters>)}

For nested definitions, label the parent as the first frame of the current environment

Bind \texttt{<name>} to the function value in the first frame of the current environment

When calling a function:
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
\(<\text{name}\>(\text{<formal parameters>})\)

For nested definitions, label the parent as the first frame of the current environment

Bind \(<\text{name}\>) to the function value in the first frame of the current environment

When calling a function:

1. Add a local frame labeled with the \(<\text{name}\>) of the function
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
<name>(<formal parameters>)

For nested definitions, label the parent as the first frame of the current environment

Bind <name> to the function value in the first frame of the current environment

When calling a function:

1. Add a local frame labeled with the <name> of the function
2. If the function has a parent label, copy it to this frame
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
<name>(<formal parameters>)

For nested definitions, label the parent as the first frame of the current environment

Bind <name> to the function value in the first frame of the current environment

When calling a function:

1. Add a local frame labeled with the <name> of the function
2. If the function has a parent label, copy it to this frame
3. Bind the <formal parameters> to the arguments in this frame
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
<name>(<formal parameters>)

For nested definitions, label the parent as the first frame of the current environment

Bind <name> to the function value in the first frame of the current environment

When calling a function:

1. Add a local frame labeled with the <name> of the function
2. If the function has a parent label, copy it to this frame
3. Bind the <formal parameters> to the arguments in this frame
4. Execute the body of the function in the environment that starts with this frame
Environment for Function Composition

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

```python
1   def square(x):
2       return x * x
3
4   def make_adder(n):
5       def adder(k):
6           return n + k
7       return adder
8
9   def compose1(f, g):
10      def h(x):
11          return f(g(x))
12      return h
13
14   compose1(square, make_adder(2))(3)
```

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

```python
1 def square(x):
2    return x * x
3
4 def make_adder(n):
5    def adder(k):
6        return n + k
7    return adder
8
9 def compose1(f, g):
10   def h(x):
11      return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example:

```python
1. def square(x):
   2.     return x * x
3. def make_adder(n):
4.     def adder(k):
5.         return n + k
6.     return adder
7. def compose1(f, g):
8.     def h(x):
9.         return f(g(x))
10. return h
11. compose1(square, make_adder(2))(3)
```

Return value of `make_adder` is an argument to `compose1`

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

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

def make_adder(n):
    def adder(k):
        return n + k
    return adder

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)

Return value of `make_adder` is an argument to `compose1`
Environment for Function Composition

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

def make_adder(n):
    def adder(k):
        return n + k
    return adder

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```

Return value of `make_adder` is an argument to `compose1`

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

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

def make_adder(n):
    def adder(k):
        return n + k
    return adder

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```

Return value of make_adder is an argument to compose1

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example: [link](http://goo.gl/5zcug)
Environment for Function Composition

Example:
```
1. def square(x):
   return x * x
2. def make_adder(n):
   def adder(k):
       return n + k
   return adder
3. def compose1(f, g):
   def h(x):
       return f(g(x))
   return h
4. compose1(square, make_adder(2))(3)
```

Return value of make_adder is an argument to compose1

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Return value of make_adder is an argument to compose1

Example: [link](http://goo.gl/5zcug)
Lambda Expressions
Lambda Expressions

```python
>>> ten = 10
```
Lambda Expressions

```python
>>> ten = 10

>>> square = x * x
```
Lambda Expressions

```python
>>> ten = 10

>>> square = x * x
```

An expression: this one evaluates to a number
>>> ten = 10

>>> square = \(x \times x\)

>>> square = lambda x: x * x

An expression: this one evaluates to a number
Lambda Expressions

```python
>>> ten = 10

>>> square = x * x

An expression: this one evaluates to a number

>>> square = lambda x: x * x

Also an expression: evaluates to a function
```
Lambda Expressions

```python
>>> 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

A function
Lambda Expressions

```python
>>> 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

A function with formal parameter x
Lambda Expressions

```python
>>> 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

```python
>>> 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

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

Notice: no "return"
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

A function

with formal parameter x

and body "return x * x"

Notice: no "return"

Must be a single expression
Lambda Expressions

```python
>>> ten = 10
>>> square = x * x
>>> square = lambda x: x * x
>>> square(4)
16
```

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

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 are rare in Python, but important in general
Evaluation of Lambda vs. Def
Evaluation of Lambda vs. Def

\textbf{lambda } x: x * x
Evaluation of Lambda vs. Def

\[ \text{lambda } x: x \ast x \quad \text{VS} \]
Evaluation of Lambda vs. Def

\[
\text{lambda } x: x \times x \quad \text{VS} \quad \text{def } \text{square}(x): \\
\quad \text{return } x \times x
\]
Evaluation of Lambda vs. Def

\[
\text{lambda } x: \ x * x \quad \text{VS} \quad \text{def } \text{square}(x): \\
\text{return } x * x
\]

Execution procedure for def statements:
Evaluation of Lambda vs. Def

\[ \texttt{lambda } x : x * x \quad \text{VS} \quad \texttt{def } \texttt{square}(x) : \texttt{return } x * x \]

Execution procedure for def statements:
1. Create a function value with signature \(<\text{name}>(<\text{formal parameters}>)>\) and the current frame as parent.
Evaluation of Lambda vs. Def

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

Execution procedure for def statements:
1. Create a function value with signature `<name>(<formal parameters>)` and the current frame as parent
2. Bind `<name>` to that value in the current frame
Evaluation of Lambda vs. Def

```
lambda x: x * x
```

VS

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

Execution procedure for def **statements**:

1. Create a function value with signature 
   `<name>`(<formal parameters>)
   and the current frame as parent

2. Bind `<name>` to that value in the current frame

Evaluation procedure for lambda **expressions**:
Evaluation of Lambda vs. Def

\[ \text{lambda } x: x \times x \quad \text{VS} \quad \text{def } \text{square}(x): \text{return } x \times x \]

Execution procedure for def statements:
1. Create a function value with signature \(<\text{name}>(\text{<formal parameters>})\)
   and the current frame as parent
2. Bind \(<\text{name}>\) to that value in the current frame

Evaluation procedure for lambda expressions:
1. Create a function value with signature \(\lambda(\text{<formal parameters>})\)
   and the current frame as parent
Evaluation of Lambda vs. Def

\[
\text{lambda } x: x \times x \quad \text{VS} \quad \text{def } \text{square}(x): \text{return } x \times x
\]

Execution procedure for def statements:
1. Create a function value with signature \( <\text{name}>(<\text{formal parameters}>) \)
   and the current frame as parent
2. Bind \( <\text{name}> \) to that value in the current frame

Evaluation procedure for lambda expressions:
1. Create a function value with signature \( \lambda(<\text{formal parameters}>) \)
   and the current frame as parent
Evaluation of Lambda vs. Def

\[ \text{lambda } x: x \times x \quad \text{VS} \quad \text{def } \text{square}(x): \text{return } x \times x \]

**Execution procedure for def statements:**
1. Create a function value with signature \(<\text{name}>(<\text{formal parameters}>)\) and the current frame as parent
2. Bind \(<\text{name}>\) to that value in the current frame

**Evaluation procedure for lambda expressions:**
1. Create a function value with signature \(\lambda(<\text{formal parameters}>)\) and the current frame as parent
2. Evaluate to that value
Lambda vs. Def Statements
Lambda vs. Def Statements

```python
square = lambda x: x * x
```
Lambda vs. Def Statements

\[ \text{square} = \lambda x: x \times x \quad \text{VS} \]
Lambda vs. Def Statements

```
square = lambda x: x * x

def square(x):
    return x * x
```
Lambda vs. Def Statements

\[
\text{square} = \text{lambda } x: x \ast x \quad \text{VS} \quad \text{def } \text{square}(x): \text{return } x \ast x
\]

Both create a function with the same arguments & behavior
Lambda vs. Def Statements

\[\text{square} = \text{lambda } x: x \ast x \quad \text{VS} \quad \text{def} \text{ square}(x): \text{ return } x \ast x\]

Both create a function with the same arguments & behavior

Both of those functions are associated with the environment in which they are defined
Lambda vs. Def Statements

\[
\text{square} = \text{lambda } x: x \times x \quad \text{VS} \quad \text{def } \text{square}(x) : \\
\text{return } x \times x
\]

Both create a function with the same arguments & behavior

Both of those functions are associated with the environment in which they are defined

Both bind that function to the name "square"
Lambda vs. Def Statements

\[
square = \text{lambda } x: \ x \times x \quad \text{VS} \quad \text{def } \text{square}(x): \ \\
\quad \quad \text{return } x \times x
\]

Both create a function with the same arguments & behavior.

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 vs. Def Statements

\[
\text{square} = \text{lambda } x: x \times x \quad \text{VS} \quad \text{def } \text{square}(x): \text{return } x \times x
\]

Both create a function with the same arguments & behavior

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 vs. Def Statements

\[
\text{square} = \lambda x: x \times x \quad \text{VS} \quad \text{def} \ square(x): \\
\quad \text{return} \ x \times x
\]

Both create a function with the same arguments & behavior

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 vs. Def Statements

```python
square = lambda x: x * x  # VS
```
```python
def square(x):
    return x * x
```

Both create a function with the same arguments & behavior

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 vs. Def Statements

\[
\text{square} = \lambda x: x \times x \quad \text{VS} \quad \text{def square}(x): \quad \text{return} \; x \times x
\]

Both create a function with the same arguments & behavior.

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.

The Greek letter lambda
Newton’s Method Background

Finds approximations to zeroes of differentiable functions
Newton’s Method Background

Finds approximations to zeroes of differentiable functions

\[ f(x) = x^2 - 2 \]
Newton’s Method Background

Finds approximations to zeroes of differentiable functions

\[ f(x) = x^2 - 2 \]
Newton’s Method Background

Finds approximations to zeroes of differentiable functions

\[ f(x) = x^2 - 2 \]
Newton’s Method Background

Finds approximations to zeroes of differentiable functions

\[ f(x) = x^2 - 2 \]

A “zero”

\[ x = 1.414213562373095 \]
Newton’s Method Background

Finds approximations to zeroes of differentiable functions

\[ f(x) = x^2 - 2 \]

Application: a method for (approximately) computing square roots, using only basic arithmetic.
Newton’s Method Background

Finds approximations to zeroes of differentiable functions

\[ f(x) = x^2 - 2 \]

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

The positive zero of \( f(x) = x^2 - a \) is \( \sqrt{a} \)
Newton’s Method

Begin with a function $f$ and an initial guess $x$

Newton’s Method

Begin with a function $f$ and an initial guess $x$

Newton’s Method

Begin with a function $f$ and an initial guess $x$

Compute the value of $f$ at the guess: $f(x)$

Newton’s Method

Begin with a function $f$ and an initial guess $x$

Compute the value of $f$ at the guess: $f(x)$
Compute the derivative of $f$ at the guess: $f'(x)$

Newton’s Method

Begin with a function $f$ and an initial guess $x$

Compute the value of $f$ at the guess: $f(x)$

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

Update guess to be:

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

Newton’s Method

Begin with a function $f$ and an initial guess $x$

Compute the value of $f$ at the guess: $f(x)$

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

Update guess to be:

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

Newton’s Method

Begin with a function $f$ and an initial guess $x$

Compute the value of $f$ at the guess: $f(x)$

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

Update guess to be:

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

Newton’s Method

Begin with a function $f$ and an initial guess $x$.

Compute the value of $f$ at the guess: $f(x)$

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

Update guess to be:

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

Newton’s Method

Begin with a function $f$ and an initial guess $x$

Compute the value of $f$ at the guess: $f(x)$

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

Update guess to be:

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

Using Newton’s Method
Using Newton’s Method

How to find the square root of 2?
Using Newton’s Method

How to find the square root of 2?

```python
>>> f = lambda x: x*x - 2
>>> find_zero(f)
1.4142135623730951
```
Using Newton’s Method

How to find the square root of 2?

```python
>>> f = lambda x: x**2 - 2
>>> find_zero(f)
1.4142135623730951
```

\( f(x) = x^2 - 2 \)
Using Newton’s Method

How to find the square root of 2?

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

```python
>>> f = lambda x: x**2 - 2
>>> find_zero(f)
1.4142135623730951
```
Using Newton’s Method

How to find the **square root** of 2?

```python
>>> f = lambda x: x*x - 2
>>> find_zero(f)
1.4142135623730951
```

How to find the **log base 2** of 1024?

```python
f(x) = x^2 - 2
```
Using Newton’s Method

How to find the square root of 2?

\[ f(x) = x^2 - 2 \]

```python
>>> f = lambda x: x*x - 2
>>> find_zero(f)
1.4142135623730951
```

How to find the log base 2 of 1024?

```python
>>> g = lambda x: pow(2, x) - 1024
>>> find_zero(g)
10.0
```
Using Newton’s Method

How to find the square root of 2?

```python
>>> f = lambda x: x*x - 2
>>> find_zero(f)
1.4142135623730951
```

\[ f(x) = x^2 - 2 \]

How to find the log base 2 of 1024?

```python
>>> g = lambda x: pow(2, x) - 1024
>>> find_zero(g)
10.0
```

\[ g(x) = 2^x - 1024 \]
Using Newton’s Method

How to find the **square root** of 2?

```python
>>> f = lambda x: x*x - 2
>>> find_zero(f)
1.4142135623730951
```

\[ f(x) = x^2 - 2 \]

How to find the **log base 2** of 1024?

```python
>>> g = lambda x: pow(2, x) - 1024
>>> find_zero(g)
10.0
```

\[ g(x) = 2^x - 1024 \]
Special Case: Square Roots
Special Case: Square Roots

How to compute $\text{square_root}(a)$

Idea: Iteratively refine a guess $x$ about the square root of $a$
Special Case: Square Roots

How to compute $\text{square\_root}(a)$

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

Update:
Special Case: Square Roots

How to compute $\text{square_root}(a)$

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

Update:

$$x = \frac{x + \frac{a}{x}}{2}$$
Special Case: Square Roots

How to compute \texttt{square_root}(a)

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

Update:

\[
x = \frac{x + \frac{a}{x}}{2}
\]

\[x - f(x)/f'(x)\]
Special Case: Square Roots

How to compute \texttt{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 $\text{square\_root}(a)$

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

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

Implementation questions:
Special Case: Square Roots

How to compute \texttt{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?
Special Case: Square Roots

How to compute \texttt{square_root(a)}

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

Update:

\[ x = \frac{x + \frac{a}{x}}{2} \]

Implementation questions:

What guess should start the computation?

How do we know when we are finished?
Special Case: Cube Roots
Special Case: Cube Roots

How to compute $\text{cube\_root}(a)$

Idea: Iteratively refine a guess $x$ about the cube root of $a$
Special Case: Cube Roots

How to compute $\text{cube\_root}(a)$

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

Update:
Special Case: Cube Roots

How to compute $\text{cube\_root}(a)$

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

Update:

$$x = \frac{2x + \frac{a}{x^2}}{3}$$
Special Case: Cube Roots

How to compute \texttt{cube_root(a)}

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

Update:

\[ x = \frac{2x + \frac{a}{x^2}}{3} \]

\[ x - f(x)/f'(x) \]
Special Case: Cube Roots

How to compute \texttt{cube\_root}(a)

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

Update:

\[
x = \frac{2x + \frac{a}{x^2}}{3}
\]

Implementation questions:
Special Case: Cube Roots

How to compute $\text{cube\_root}(a)$

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

Update:

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

Implementation questions:

What guess should start the computation?
Special Case: Cube Roots

How to compute \texttt{cube_root}(a)

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

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

Implementation questions:

What guess should start the computation?

How do we know when we are finished?
Iterative Improvement
Iterative Improvement

First, identify common structure.
Iterative Improvement

First, identify common structure.
Then define a function that generalizes the procedure.
Iterative Improvement

First, identify common structure.
Then define a function that generalizes the procedure.

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

    >>> 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
Newton’s Method for nth Roots
Newton’s Method for nth Roots

```python
def nth_root_func_and_derivative(n, a):
    def root_func(x):
        return pow(x, n) - a
    def derivative(x):
        return n * pow(x, n-1)
    return root_func, derivative

def nth_root_newton(a, n):
    """Return the nth root of a."
    root_func, deriv = nth_root_func_and_derivative(n, a)
    def update(x):
        return x - root_func(x) / deriv(x)
    def done(x):
        return root_func(x) == 0
    return iter_improve(update, done)
```

```python
>>> nth_root_newton(8, 3)
2.0
"""
```
Newton’s Method for nth Roots

```python
def nth_root_func_and_derivative(n, a):
    def root_func(x):
        return pow(x, n) - a
    def derivative(x):
        return n * pow(x, n-1)
    return root_func, derivative

def nth_root_newton(a, n):
    """Return the nth root of a."
    root_func, deriv = nth_root_func_and_derivative(n, a)
    def update(x):
        return x - root_func(x) / deriv(x)
    def done(x):
        return root_func(x) == 0
    return iter_improve(update, done)
```

```python
>>> nth_root_newton(8, 3)
2.0
"""
```
Newton’s Method for nth Roots

```python
def nth_root_func_and_derivative(n, a):
    def root_func(x):
        return pow(x, n) - a
    def derivative(x):
        return n * pow(x, n-1)
    return root_func, derivative

def nth_root_newton(a, n):
    """Return the nth root of a."

    root_func, deriv = nth_root_func_and_derivative(n, a)
    def update(x):
        return x - root_func(x) / deriv(x)
    def done(x):
        return root_func(x) == 0
    return iter_improve(update, done)
```

$\textit{Exact derivative}$
def nth_root_func_and_derivative(n, a):
    def root_func(x):
        return pow(x, n) - a
    def derivative(x):
        return n * pow(x, n-1)
    return root_func, derivative

def nth_root_newton(a, n):
    """Return the nth root of a."

    >>> nth_root_newton(8, 3)
    2.0
    """

    root_func, deriv = nth_root_func_and_derivative(n, a)
    def update(x):
        return x - root_func(x) / deriv(x)
    def done(x):
        return root_func(x) == 0
    return iter_improve(update, done)