Announcements

• Homework 2 due Tuesday 9/17 @ 11:59pm
Announcements

• Homework 2 due Tuesday 9/17 @ 11:59pm

• Project 2 due Thursday 9/19 @ 11:59pm
Announcements

• Homework 2 due Tuesday 9/17 @ 11:59pm

• Project 2 due Thursday 9/19 @ 11:59pm

• Optional Guerrilla section next Monday for students to master higher-order functions
Announcements

• Homework 2 due Tuesday 9/17 @ 11:59pm

• Project 2 due Thursday 9/19 @ 11:59pm

• Optional Guerrilla section next Monday for students to master higher-order functions
  • Organized by Andrew Huang and the readers
Announcements

• Homework 2 due Tuesday 9/17 @ 11:59pm

• Project 2 due Thursday 9/19 @ 11:59pm

• Optional Guerrilla section next Monday for students to master higher-order functions
  - Organized by Andrew Huang and the readers
  - Work in a group on a problem until everyone in the group understands the solution
Announcements

• Homework 2 due Tuesday 9/17 @ 11:59pm

• Project 2 due Thursday 9/19 @ 11:59pm

• Optional Guerrilla section next Monday for students to master higher-order functions
  ▪ Organized by Andrew Huang and the readers
  ▪ Work in a group on a problem until everyone in the group understands the solution

• Midterm 1 on Monday 9/23 from 7pm to 9pm
Announcements

• Homework 2 due Tuesday 9/17 @ 11:59pm

• Project 2 due Thursday 9/19 @ 11:59pm

• Optional Guerrilla section next Monday for students to master higher-order functions
  ▪ Organized by Andrew Huang and the readers
  ▪ Work in a group on a problem until everyone in the group understands the solution

• Midterm 1 on Monday 9/23 from 7pm to 9pm
  ▪ Details and review materials will be posted early next week
Announcements

• Homework 2 due Tuesday 9/17 @ 11:59pm

• Project 2 due Thursday 9/19 @ 11:59pm

• Optional Guerrilla section next Monday for students to master higher-order functions
  ▪ Organized by Andrew Huang and the readers
  ▪ Work in a group on a problem until everyone in the group understands the solution

• Midterm 1 on Monday 9/23 from 7pm to 9pm
  ▪ Details and review materials will be posted early next week
  ▪ There will be a web form for students who cannot attend due to a conflict
Lambda Expressions

(Demo)
Lambda Expressions
Lambda Expressions

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

```python
>>> ten = 10

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

```python
>>> ten = 10

An expression: this one evaluates to a number

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

```python
>>> ten = 10

>>> square = x * x

An expression: this one evaluates to a number

>>> square = lambda x: 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
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
```
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
```
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
    that returns the value of "x * 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
Important: No "return" keyword!
A function
with formal parameter x
that returns the value of "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
A function with formal parameter x that returns the value of "x * x"

Important: No "return" keyword!

Must be a single expression
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
Important: No "return" keyword!
A function
with formal parameter x
that returns the value of "x * x"

>>> square(4)
16
Must be a single expression
```
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
Important: No "return" keyword!
A function with formal parameter x
that returns the value of "x * x"

>>> square(4)
16
Must be a single expression
```

Lambda expressions are not common in Python, but important in general.
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
Important: No "return" keyword!
A function
with formal parameter x
that returns the value of "x * x"

>>> square(4)
16
Must be a single expression

Lambda expressions are not common in Python, but important in general.
Lambda expressions in Python cannot contain statements at all!
Lambda Expressions Versus Def Statements

Example: http://goo.gl/XH54uE
Lambda Expressions Versus Def Statements

VS

Example:  http://goo.gl/XH54uE
Lambda Expressions Versus Def Statements

\[
square = \lambda x: x \times x
\]

VS

Example: [http://goo.gl/XH54uE](http://goo.gl/XH54uE)
Lambda Expressions Versus Def Statements

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

Example: [http://goo.gl/XH54yE](http://goo.gl/XH54yE)
Lambda Expressions Versus 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 domain, range, and behavior.

Example: http://goo.gl/XH54uE
Lambda Expressions Versus Def Statements

\[\text{square} = \lambda x: x \times x\]  \hspace{1cm} \text{VS} \hspace{1cm} \text{def square}(x):\]
\[\text{return } x \times x\]

• Both create a function with the same domain, range, and behavior.
• Both functions have as their parent the environment in which they were defined.

Example: [http://goo.gl/XH54uE](http://goo.gl/XH54uE)
Lambda Expressions Versus Def Statements

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name \text{square}.

Example: http://goo.gl/XH54uE
Lambda Expressions Versus Def Statements

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name \textit{square}.
- Only the \texttt{def} statement gives the function an intrinsic name.

Example: \url{http://goo.gl/XH54uE}
Lambda Expressions Versus Def Statements

\[
square = \text{lambda } x: x \times x
\]

VS

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name `square`.
- Only the `def` statement gives the function an intrinsic name.

Example: [http://goo.gl/XH54uE](http://goo.gl/XH54uE)
Lambda Expressions Versus Def Statements

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

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name `square`.
- Only the `def` statement gives the function an intrinsic name.

Example:  http://goo.gl/XH54uE
Lambda Expressions Versus Def Statements

\[
square = \lambda x: x \times x
\]

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name `square`.
- Only the `def` statement gives the function an intrinsic name.

Example: [http://goo.gl/XH54uE](http://goo.gl/XH54uE)
Lambda Expressions Versus Def Statements

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name \textit{square}.
- Only the \texttt{def} statement gives the function an intrinsic name.

Example: \url{http://goo.gl/XH54uE}
Currying
Function Currying
Function Currying

def make_adder(n):
    return lambda k: n + k
Function Currying

```python
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

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

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

There's a general relationship between these functions.
Function Currying

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

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

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

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

```python
>>> 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.

Currying was discovered by Moses Schönfinkel and re-discovered by Haskell Curry.
Function Currying

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

```python
>>> 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.

Currying was discovered by Moses Schönfinkel and re-discovered by Haskell Curry.

Schönfinkeling?
Newton's Method
Newton's Method Background

Quickly finds accurate approximations to zeroes of differentiable functions!
Newton's Method Background

Quickly finds accurate approximations to zeroes of differentiable functions!

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

Quickly finds accurate approximations to zeroes of differentiable functions!

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

Quickly finds accurate approximations to zeroes of differentiable functions!

A "zero" of a function $f$ is an input $x$ such that $f(x) = 0$.
Newton's Method Background

Quickly finds accurate approximations to zeroes of differentiable functions!

f(x) = x^2 - 2

A "zero" of a function f is an input x such that f(x)=0

x=1.414213562373095
Newton's Method Background

Quickly finds accurate approximations to zeroes of differentiable functions!

A "zero" of a function $f$ is an input $x$ such that $f(x) = 0$.

Application: a method for computing square roots, cube roots, etc.
Newton's Method Background

Quickly finds accurate approximations to zeroes of differentiable functions!

Application: a method for computing square roots, cube roots, etc.

The positive zero of $f(x) = x^2 - a$ is $\sqrt{a}$. (We're solving the equation $x^2 = a$.)
Newton's Method

Given a function $f$ and initial guess $x$, 

Newton's Method

Given a function $f$ and initial guess $x$,

Repeatedly improve $x$:

Newton's Method

Given a function $f$ and initial guess $x$,

Repeatedly improve $x$:
Newton's Method

Given a function $f$ and initial guess $x$,

Repeatedly improve $x$:

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

Newton's Method

Given a function $f$ and initial guess $x$,

Repeatedly improve $x$:

1. Compute the value of $f$ at the guess: $f(x)$
2. Compute the derivative of $f$ at the guess: $f'(x)$
Newton's Method

Given a function $f$ and initial guess $x$,

Repeatedly improve $x$:

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

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

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

---

Newton's Method

Given a function $f$ and initial guess $x$,

Repeatedly improve $x$:

1. Compute the value of $f$ at the guess: $f(x)$
2. Compute the derivative of $f$ at the guess: $f'(x)$
3. Update guess $x$ to be:
   \[ x = x - \frac{f(x)}{f'(x)} \]
Newton's Method

Given a function $f$ and initial guess $x$,

Repeatedly improve $x$:

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

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

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

Newton's Method

Given a function f and initial guess x,

Repeatedly improve x:

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

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

3. Update guess x to be:
   \[ x - \frac{f(x)}{f'(x)} \]
Newton's Method

Given a function $f$ and initial guess $x$,

Repeatedly improve $x$:

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

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

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

Given a function $f$ and initial guess $x$,

Repeatedly improve $x$:

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

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

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

Newton's Method

Given a function $f$ and initial guess $x$,

Repeatedly improve $x$:

1. Compute the value of $f$ at the guess: $f(x)$
2. Compute the derivative of $f$ at the guess: $f'(x)$
3. Update guess $x$ to be:
   $$x - \frac{f(x)}{f'(x)}$$

Newton's Method

Given a function $f$ and initial guess $x$,

Repeatedly improve $x$:

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

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

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

Finish when $f(x) = 0$ (or close enough)
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**2 - 2
>>> df = lambda x: 2*x
>>> find_zero(f, df)
1.4142135623730951
```
Using Newton's Method

How to find the square root of 2?

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

How to find the square root of 2?

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

\[ f(x) = x^2 - 2 \]
\[ f'(x) = 2x \]
Using Newton's Method

How to find the square root of 2?

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

Applies Newton's method until $|f(x)| < 10^{-15}$, starting at 1
Using Newton's Method

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

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

How to find the **cube root** of 729?

```python
Applies Newton's method until |f(x)| < 10^{-15}, starting at 1
```
Using Newton's Method

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

\[
\begin{align*}
\text{f}(x) &= x^2 - 2 \\
\text{f}'(x) &= 2x
\end{align*}
\]

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

How to find the **cube root** of 729?

```
Applies Newton's method until |f(x)| < 10^{-15}, starting at 1
```
Using Newton's Method

How to find the square root of 2?

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

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

Applies Newton's method until $|f(x)| < 10^{-15}$, starting at 1

How to find the cube root of 729?

```python
>>> g = lambda x: x**3 - 729
>>> dg = lambda x: 3*x**2
>>> find_zero(g, dg)
9.0
```

$pV = nRT$
Using Newton's Method

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

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

How to find the **cube root** of 729?

```python
>>> g = lambda x: x**3 - 729
>>> dg = lambda x: 3*x**2
>>> find_zero(g, dg)
9.0
```
Iterative Improvement
Special Case: Square Roots
Special Case: Square Roots

How to compute square_root(a)

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

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

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

\textbf{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 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 $a$

**Update:**

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

**Babylonian Method**

**Implementation questions:**
Special Case: Square Roots

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

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

**Update:**
\[
x = \frac{x + \frac{a}{x}}{2}
\]

**Babylonian Method**

**Implementation questions:**

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

How to compute \texttt{square_root}(a)

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

Update:

\[
\text{\( x = \frac{x + \frac{a}{x}}{2} \)}
\]

\textbf{Babylonian Method}

\textbf{Implementation questions:}

What \textit{guess} should start the computation?

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

How to compute cube_root(a)

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

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

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

\textbf{Update:}
Special Case: Cube Roots

How to compute cube_root(a)

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

**Update:**

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

How to compute \( \text{cube}_\text{root}(a) \)

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

\[
\text{Update:} \quad x = \frac{2 \cdot x + \frac{a}{x^2}}{3}
\]

**Implementation questions:**

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

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

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

**Update:**

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

**Implementation questions:**

What *guess* should start the computation?

How do we know when we are finished?
Implementing Newton's Method