Lambda Expressions
Lambda Expressions

>>> ten = 10
Lambda Expressions

>>> ten = 10

>>> square = x * x
Lambda Expressions

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

```python
>>> square = 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

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

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

>>> ten = 10

An expression: this one evaluates to a number

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

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

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

Also an expression: evaluates to a function

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

A function
Lambda Expressions

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

>>> square = \(x \times x\)
Also an expression: evaluates to a function

>>> square = lambda x: x * x
A function with formal parameter \(x\)
```
Lambda Expressions

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

```
>>> 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
```

An expression: this one evaluates to a number

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

Also an expression: evaluates to a function

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

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

Notice: no "return"
Lambda Expressions

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

An expression: this one evaluates to a number

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

Also an expression: evaluates to a function

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

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

Notice: no "return"

Must be a single expression
Lambda Expressions

>>> ten = 10

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

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

An expression: this one evaluates to a number

Also an expression: evaluates to a function

A function with formal parameter \(x\) and body "return \(x \times x\)"

>>> square(4)
16

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

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

Notice: no "return"

Must be a single expression

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

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

\[
\text{square} = \lambda x: x \times x \quad \text{VS} \quad \text{def square}(x): \text{return } x \times x
\]
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 arguments & body
Lambda Expressions Versus Def Statements

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

- 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

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

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

- Both create a function with the same 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

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

- Both create a function with the same arguments & 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

\[
\langle\text{lambda}\rangle(x): \quad \text{return } x \times x
\]
Lambda Expressions Versus Def Statements

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

- 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

\[ \begin{array}{l}
\text{<lambda>}(x):
\quad \text{return} \ x \times x \\
\end{array} \]
\[ \begin{array}{l}
\text{square}(x):
\quad \text{return} \ x \times x \\
\end{array} \]
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
```

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

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

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.
Newton's Method Background

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

Finds approximations to zeroes of differentiable functions

\[ y = x^2 - 2 \]
Newton's Method Background

Finds approximations to zeroes of differentiable functions

\[ y = x^2 - 2 \]
Newton's Method Background

Finds approximations to zeroes of differentiable functions

\[ y = x^2 - 2 \]
Newton's Method Background

Finds approximations to zeroes of differentiable functions

\[ y = x^2 - 2 \]

A "zero"

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

Finds approximations to zeroes of differentiable functions

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

\[ y = x^2 - 2 \]

A "zero"

\[ x = 1.414213562373095 \]
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

$x = 1.414213562373095$
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: $f(x)$

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

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?

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

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

How to find the square root of 2?

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

How to find the log base 2 of 1024?
Using Newton's Method

How to find the square root of 2?

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

How to find the log base 2 of 1024?

```python
>>> g = lambda x: pow(2, x) - 1024
>>> find_root(g, 1)
```
Using Newton's Method

How to find the square root of 2?

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

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

How to find the log base 2 of 1024?

```python
>>> 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**2 - 2
>>> find_root(f, 1)

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**2 - (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 \( \text{square\_root}(a) \)

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

**Update:**

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

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

**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 $\text{cube_root}(a)$

**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 $\text{cube}_\text{root}(a)$

**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?
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: Iteratively 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
    """
    k = 0
    while not done(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess

>>> iter_improve(golden_update, golden_test)
1.618033988749895
"""
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

golden_update:
golden_test:
it_improve:

update:
done:
guess: 1
Iterative Improvement

golden_update:
golden_test:
iter_improve:

update:
done:
guess: 1
iter_improve

guess: 1
golden_test
Iterative Improvement
Iterative Improvement

golden_update:
golden_test:
iter_improve:

update:
done:
guess: 1
iter_improve

guess: 1
golden_test

guess:
Iterative Improvement

1
2.0

guess:
guess: 1

golden_update:
golden_test:
iter_improve:

update:
done:
guess: 1

iter_improve

golden_update
golden_test
iter_improve
Iterative Improvement

golden_update:
golden_test:
iter_improve:

update:
done:
guess: 1

iter_improve

guess: 1
golden_test

guess:

1
2.0
1.5
Iterative Improvement

golden_update:
golden_test:
it_improve:

update:
done:
guess: 1

1
2.0
1.5
1.6666666666666665

guess:
Iterative Improvement

```
golden_update:
golden_test:
iter_improve:

golden_update
golden_test
iter_improve

update:
done:
guess: 1

iter_improve

1
2.0
1.5
1.6666666666666665
1.6

guess:
```
Iterative Improvement

guess: 1
golden_update:
golden_test:
iter_improve:

update:
done:
guess: 1

iter_improve

guess: 1
golden_test

1
2.0
1.5
1.6666666666666665
1.6
1.625

guess:
Iterative Improvement

golden_update:
golden_test:
iter_improve:

update:
done:
guess: 1

iter_improve
guess: 1
golden_test

guess:

1
2.0
1.5
1.6666666666666665
1.6
1.625
1.6153846153846154
Iterative Improvement

update:
done:
guess: 1

iter_improve:

golden_update:
golden_test:
iter_improve:

golden_update
golden_test
iter_improve

guess: 1

1
2.0
1.5
1.6666666666666665
1.6
1.625
1.615384615384615
1.619047619047619

guess:

1.619047619047619

Friday, September 9, 2011
Iterative Improvement

golden_update:
golden_test:
iter_improve:

update:
done:
guess: 1

iter_improve

golden_update
golden_test
iter_improve

guess:
1
2.0
1.5
1.6666666666666665
1.6
1.625
1.6153846153846154
1.619047619047619
1.6176470588235294

guess:
Iterative Improvement

golden_update:
golden_test:
iter_improve:

golden_update
golden_test
iter_improve

update:
done:
guess: 1

iter_improve

golden_test

guess:

1
2.0
1.5
1.6666666666666665
1.6
1.625
1.6153846153846154
1.619047619047619
1.6176470588235294
1.6181818181818182
Square Roots by Iterative Improvement
Square Roots by Iterative Improvement

\[
\text{square_root:} \\
\text{iter_improve:} ... \\
\]

(Demo)
Square Roots by Iterative Improvement

square_root:
iter_improve:
...

square_root(256)
Square Roots by Iterative Improvement

square_root:

iter_improve:

... (Demo)

square_root(256)
Square Roots by Iterative Improvement

\[
\text{square_root:} \\
\text{iter_improve:} \\
\ldots
\]

(square root)
Square Roots by Iterative Improvement

\[ \sqrt{256} \]

\[ \text{iter}_\text{improve}: \ldots \]

\[ \text{a: 256} \]

\[ \text{square}_\text{root}(256) \]
Square Roots by Iterative Improvement

\[
\text{square_root:} \\
\text{iter_improve:} \\
... \\
\]

\[
a: 256 \\
\]

\[
\text{square_root(256)} \\
def \text{update}(\text{guess}): \\
... \\
... \\
\text{return iter_improve(...) } \\
\]

(Demo)
Square Roots by Iterative Improvement

```
square_root:
iter_improve:
...
```

```
def update(guess):
...
...
return iter_improve(...)  # square_root(256)
```

```
a: 256
```

(Demo)
Square Roots by Iterative Improvement

```
def update(guess):
    ...
    return iter_improve(...)
```

```
square_root: ...
iter_improve: ...
```

```
a: 256
update:
```

```
square_root(256)
```

(Demo)
Square Roots by Iterative Improvement

```
def update(guess):
    ...
    return iter_improve(...)
```

```
square_root(256)
```

(Demo)
Square Roots by Iterative Improvement

\[ \text{square_root:} \]
\[ \text{iter_improve:} \]
\[ \ldots \]

**Demo**

\[ a: 256 \]
\[ \text{update:} \]
\[ \text{done:} \ldots \]

\[ \text{update(guess):} \]
\[ \text{return} \ldots \]

\[ \text{square_root(256)} \]
\[ \ldots \]
\[ \text{return iter_improve(...)} \]
\[ \text{square_root(256)} \]
Square Roots by Iterative Improvement

\[
\text{square_root: }
\]

\[
\text{iter_improve: }
\]

\[
\text{\quad ...}
\]

\[
\text{\quad (Demo)}
\]

\[
\text{\quad a: 256}
\]

\[
\text{\quad update: }
\]

\[
\text{\quad done: ...}
\]

\[
\text{\quad guess: 1}
\]

\[
\text{\quad update(guess): }
\]

\[
\text{\quad return ...}
\]

\[
\text{\quad return iter_improve(...)}
\]

\[
\text{\quad square_root(256)}
\]

\[
\text{\quad ...}
\]
Square Roots by Iterative Improvement

\[ \text{square_root:} \]

\[ \text{iter_improve:} \]

\[ \text{...} \]

\[ \text{square_root(256)} \]

\[ \text{update(guess):} \]

\[ \text{return ...} \]

\[ \text{guess: 1} \]

\[ \text{update} \]

\[ \text{done: ...} \]

\[ \text{a: 256} \]

**Demo**

Friday, September 9, 2011
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

\[ f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \]
Derivatives of Single-Argument Functions

\[ f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \]
Derivatives of Single-Argument Functions

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

\[ x = 1 \]

\[ x + h = 1.1 \]
Derivatives of Single-Argument Functions

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


(Demo)
Approximating Derivatives

(Demo)
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

```python
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

```python
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

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
```

Could be replaced with the exact derivative
Implementing Newton's Method

```python
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

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
```

"Could be replaced with the exact derivative"

"Limit approximated by a small value"
Implementing Newton's Method

```python
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

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

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)
```

Limit approximated by a small value

Could be replaced with the exact derivative
Implementing Newton's Method

```python
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

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

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*

*Could be replaced with the exact derivative*

*Limit approximated by a small value*