61A Lecture 6

Friday, September 7
Lambda Expressions
Lambda Expressions

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

>>> ten = 10

>>> square = x * x
Lambda Expressions

>>> ten = 10

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

An expression: this one evaluates to a number
Lambda Expressions

```python
>>> ten = 10

>>> square = x * x

>>> square = lambda x: 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

Also an expression: evaluates to a function

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

```python
>>> ten = 10

>>> square = x * x
```

An expression: this one evaluates to a number

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

Also an expression: evaluates to a function

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

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

>>> 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"
Lambda Expressions

>>> ten = 10

>>> square = x * x

An expression: this one evaluates to a number

Also an expression: evaluates to a function

>>> 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

>>> 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"

Notice: no "return"

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

>>> 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

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

Notice: no "return"

>>> square(4)

16

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

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

square = lambda x: x * x  

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

• Both create a function with the same arguments & behavior
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 & behavior
- Both of those functions are associated with the environment in which they are defined
Lambda Expressions Versus Def Statements

square = lambda \( x \): x * x  \hspace{1cm} \text{VS} \hspace{1cm} \text{def square}(x): \text{ 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"
Lambda Expressions Versus Def Statements

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

- Both create a function with the same 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 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 & 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

![Diagram showing the difference between lambda expression and def statement](image-url)
Lambda Expressions Versus Def Statements

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

- 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 Expressions Versus Def Statements

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

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

- 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 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 & 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
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

```python
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.

**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
```

```bash
>>> 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

```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.

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

*Schoenfinkeling?*
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

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

A "zero"
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

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

A "zero"

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 \( f(x) = x^2 - a \) 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 \( f(x) = x^2 - a \) 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 $x$.

$$x = x - \frac{f(x)}{f'(x)}$$
Newton's Method

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

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

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

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

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

(Demo)

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?

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

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

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

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

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

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

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

```python
>>> f = lambda x: x**2 - 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**2 - 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**2 - 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
```

What number is one less than its square?
Using Newton's Method

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

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

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

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

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

\[ g(x) = 2^x - 1024 \]

What number is one less than its square?

```python
>>> h = lambda x: x**2 - (x+1)
>>> find_zero(h)
1.618033988749895
```

\[ h(x) = x^2 - (x+1) \]
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

What number is one less than its square?

```python
>>> h = lambda x: x*x - (x+1)
>>> find_zero(h)
1.618033988749895
```  
h(x) = x^2 - (x+1)
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
```

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

\[ g(x) = 2^x - 1024 \]

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

What number is one less than its square?

\[ h(x) = x^2 - (x+1) \]

```python
>>> h = lambda x: x**2 - (x+1)
>>> find_zero(h)
1.618033988749895
```

f(x) = x² - 2

\[ g(x) = 2^x - 1024 \]

h(x) = x² - (x+1)
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 square_root(a)

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

**Update:**
Special Case: Square Roots

How to compute square_root(a)

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

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

**Implementation questions:**
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?
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: 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:**

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

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
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

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

\[ x = 1 \]
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

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

```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)
```

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)
```

- **newton_update(f)**: Returns an update function for f using Newton's method.
- **approx_derivative(f, x, delta=1e-5)**: Returns an approximation to the derivative of f at x.
- **find_root(f, guess=1)**: Returns a guess of a zero of the function f, near guess.

---

Could be replaced with the exact derivative

Limit approximated by a small value

Definition of a function zero