Announcements

- HW4 due today at 11:59pm

- Hog contest deadline on Friday
  - Completely optional, opportunity for extra credit
  - See website for details
Converting Recursion to Iteration
Can be tricky! Iteration is a special case of recursion
Converting Recursion to Iteration

Can be tricky! Iteration is a special case of recursion
Idea: Figure out what state must be maintained by the function
Can be tricky! Iteration is a special case of recursion
Idea: Figure out what state must be maintained by the function

```python
def summation(n, term):
    if n == 0:
        return 0
    return summation(n - 1, term) + term(n)
```
Can be tricky! Iteration is a special case of recursion
Idea: Figure out what state must be maintained by the function

```
def summation(n, term):
    if n == 0:
        return 0
    return summation(n - 1, term) + term(n)
```

Termination condition
Converting Recursion to Iteration

Can be tricky! Iteration is a special case of recursion
Idea: Figure out what state must be maintained by the function

```python
def summation(n, term):
    if n == 0:
        return 0
    return summation(n - 1, term) + term(n)
```

Termination condition

Initial value
Can be tricky! Iteration is a special case of recursion

Idea: Figure out what state must be maintained by the function

```python
def summation(n, term):
    if n == 0:
        return 0
    return summation(n - 1, term) + term(n)
```

Termination condition

Initial value

What's summed so far?
Converting Recursion to Iteration

Can be tricky! Iteration is a special case of recursion.

Idea: Figure out what state must be maintained by the function.

```
def summation(n, term):
    if n == 0:
        return 0
    return summation(n - 1, term) + term(n)
```

- **Termination condition**
- **Initial value**
- **What's summed so far?**
- **How to get each incremental piece**
Can be tricky! Iteration is a special case of recursion.

Idea: Figure out what state must be maintained by the function.

```python
def summation(n, term):
    if n == 0:
        return 0
    return summation(n - 1, term) + term
```

- **Termination condition**: `n == 0`
- **Initial value**: `0`
- **What's summed so far?**: `summation(n - 1, term) + term`
- **How to get each incremental piece**: `term(n)`

```python
def summation_iter(n, term):
    total = 0
    while n > 0:
        total, n = total + term(n), n - 1
    return total
```
Can be tricky! Iteration is a special case of recursion

Idea: Figure out what state must be maintained by the function

```python
def summation(n, term):
    if n == 0:
        return 0
    return summation(n - 1, term) + term(n)
```

What's summed so far?

```python
def summation_iter(n, term):
    total = 0
    while n > 0:
        total, n = total + term(n), n - 1
    return total
```

Initial value

Termination condition

How to get each incremental piece
Can be tricky! Iteration is a special case of recursion.

Idea: Figure out what state must be maintained by the function.

```python
def summation(n, term):
    if n == 0:
        return 0
    return summation(n - 1, term) + term(n)
```

What's summed so far?

```python
def summation_iter(n, term):
    total = 0
    while n > 0:
        total, n = total + term(n), n - 1
    return total
```

How to get each incremental piece?
Can be tricky! Iteration is a special case of recursion

Idea: Figure out what state must be maintained by the function

```python
def summation(n, term):
    if n == 0:
        return 0
    return summation(n - 1, term) + term(n)
```

```python
def summation_iter(n, term):
    total = 0
    while n > 0:
        total, n = total + term(n), n - 1
    return total
```
Can be tricky! Iteration is a special case of recursion
Idea: Figure out what state must be maintained by the function

```python
def summation(n, term):
    if n == 0:
        return 0
    return summation(n - 1, term) + term(n)

def summation_iter(n, term):
    total = 0
    while n > 0:
        total, n = total + term(n), n - 1
    return total
```
Converting Iteration to Recursion
More formulaic: Iteration is a special case of recursion
Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.
Idea: The state of iteration can be passed as parameters.
Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters

```python
def fib_iter(n):
    if n == 0:
        return 0
    fib_n, fib_n_1, k = 1, 0, 1
    while k < n:
        fib_n, fib_n_1 = fib_n + fib_n_1, fib_n
        k = k + 1
    return fib_n
```
More formulaic: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters

```python
def fib_iter(n):
    if n == 0:
        return 0
    fib_n, fib_n_1, k = 1, 0, 1
    while k < n:
        fib_n, fib_n_1 = fib_n + fib_n_1, fib_n
        k = k + 1
    return fib_n
```

Local names become...
Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters

```python
def fib_iter(n):
    if n == 0:
        return 0
    fib_n, fib_n_1, k = 1, 0, 1
    while k < n:
        fib_n, fib_n_1 = fib_n + fib_n_1, fib_n
        k = k + 1
    return fib_n

def fib_rec(n, fib_n, fib_n_1, k):
    if n == 0:
        return 0
    if k >= n:
        return fib_n
    return fib_rec(n, fib_n + fib_n_1, fib_n, k + 1)
```

Local names become…
More formulaic: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters

```python
def fib_iter(n):
    if n == 0:
        return 0
    fib_n, fib_n_1, k = 1, 0, 1
    while k < n:
        fib_n, fib_n_1 = fib_n + fib_n_1, fib_n
        k = k + 1
    return fib_n

def fib_rec(n, fib_n, fib_n_1, k):
    if n == 0:
        return 0
    if k >= n:
        return fib_n
    return fib_rec(n, fib_n + fib_n_1, fib_n, k + 1)
```

Local names become...

Parameters in a recursive function
Mutual Recursion
Mutual recursion is when the recursive process is split across multiple functions
Mutual recursion is when the recursive process is split across multiple functions.
Mutual Recursion

Mutual recursion is when the recursive process is split across multiple functions
Mutual Recursion

Mutual recursion is when the recursive process is split across multiple functions
Mutual Recursion

Mutual recursion is when the recursive process is split across multiple functions.
Mutual recursion is when the recursive process is split across multiple functions.

Decorating a recursive function generally results in mutual recursion.
Mutual Recursion

Mutual recursion is when the recursive process is split across multiple functions

Decorating a recursive function generally results in mutual recursion

```
@trace1
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

Example: [http://goo.gl/4LZZv](http://goo.gl/4LZZv)
Currying

We have used higher-order functions to produce a function to add a constant to its argument
Currying

We have used higher-order functions to produce a function to add a constant to its argument

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

We have used higher-order functions to produce a function to add a constant to its argument

What if we wanted to do the same for multiplication?

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

We have used higher-order functions to produce a function to add a constant to its argument.

What if we wanted to do the same for multiplication?

def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

>>> make_adder(2)(3)
5

>>> add(2, 3)
5

def make_multiplier(n):
    def multiplier(k):
        return mul(n, k)
    return multiplier

>>> make_multiplier(2)(3)
6

>>> mul(2, 3)
6
We have used higher-order functions to produce a function to add a constant to its argument.

What if we wanted to do the same for multiplication?

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

```python
def make_multiplier(n):
    def multiplier(k):
        return mul(n, k)
    return multiplier

>>> make_multiplier(2)(3)
6
>>> mul(2, 3)
6
```
Currying

We have used higher-order functions to produce a function to add a constant to its argument.

What if we wanted to do the same for multiplication?

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

def make_multiplier(n):
    def multiplier(k):
        return mul(n, k)
    return multiplier

>>> make_multiplier(2)(3)
6
>>> mul(2, 3)
6
```
Currying

We have used higher-order functions to produce a function to add a constant to its argument.

What if we wanted to do the same for multiplication?

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

def make_multiplier(n):
    def multiplier(k):
        return mul(n, k)
    return multiplier

>>> make_multiplier(2)(3)
6
>>> mul(2, 3)
6
```
Currying

We have used higher-order functions to produce a function to add a constant to its argument.

What if we wanted to do the same for multiplication?

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

def make_multiplier(n):
    def multiplier(k):
        return mul(n, k)
    return multiplier

>>> make_multiplier(2)(3)
6
>>> mul(2, 3)
6
```

Same relationship between functions.
Currying

We have used higher-order functions to produce a function to add a constant to its argument.

What if we wanted to do the same for multiplication?

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

def make_multiplier(n):
    def multiplier(k):
        return mul(n, k)
    return multiplier
```

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

```python
>>> make_multiplier(2)(3)
6
>>> mul(2, 3)
6
```

Same relationship between functions

How can we do this in general without repeating ourselves?
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

First, identify common structure.

def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

>>> make_adder(2)(3)
5

>>> add(2, 3)
5
Currying

First, identify common structure.

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

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

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

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

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

def curry2(f):
    def outer(n):
        def inner(k):
            return f(n, k)
        return inner
    return outer
```
Currying

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

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

def curry2(f):
    def outer(n):
        def inner(k):
            return f(n, k)
        return inner
    return outer

>>> curry2(mul)(2)(3)
6
>>> mul(2, 3)
6
```
First, identify common structure.

Then define a function that generalizes the procedure.

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

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

def curry2(f):
    def outer(n):
        def inner(k):
            return f(n, k)
        return inner
    return outer

>>> curry2(mul)(2)(3)
6
>>> mul(2, 3)
6
```

This process of converting a multi-argument function to consecutive single-argument functions is called *currying*. 
Functional Abstractions
def square(x):
    return mul(x, x)
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)
```python
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)
```

What does `sum_squares` need to know about `square`?
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)

What does sum_squares need to know about square?

• square takes one argument.
```python
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)
```

What does `sum_squares` need to know about `square`?

- `square` takes one argument.
- `square` has the intrinsic name `square`. 
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)

What does \texttt{sum\_squares} need to know about \texttt{square}?

• \texttt{square} takes one argument.

• \texttt{square} has the intrinsic name square.

• \texttt{square} computes the square of a number.
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)

What does \texttt{sum\_squares} need to know about \texttt{square}?

\begin{itemize}
  \item \texttt{square} takes one argument.
  \item \texttt{square} has the intrinsic name \texttt{square}.
  \item \texttt{square} computes the square of a number.
  \item \texttt{square} computes the square by calling \texttt{mul}.
\end{itemize}
Functional Abstractions

```python
def square(x):
    return mul(x, x)
def sum_squares(x, y):
    return square(x) + square(y)
```

What does `sum_squares` need to know about `square`?

- `square` takes one argument.  
  - **Yes**
- `square` has the intrinsic name `square`.
- `square` computes the square of a number.
- `square` computes the square by calling `mul`.
```python
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)
```

What does `sum_squares` need to know about `square`?

- `square` takes one argument.  
  Yes
- `square` has the intrinsic name `square`.  
  No
- `square` computes the square of a number.
- `square` computes the square by calling `mul`.  

Functional Abstractions

```python
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)
```

What does `sum_squares` need to know about `square`?

- `square` takes one argument. **Yes**
- `square` has the intrinsic name `square`. **No**
- `square` computes the square of a number. **Yes**
- `square` computes the square by calling `mul`.

def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)

What does sum_squares need to know about square?

• square takes one argument. Yes

• square has the intrinsic name square. No

• square computes the square of a number. Yes

• square computes the square by calling mul. No
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)

What does sum_squares need to know about square?

- square takes one argument. Yes
- square has the intrinsic name square. No
- square computes the square of a number. Yes
- square computes the square by calling mul. No

def square(x):
    return pow(x, 2)
Functional Abstractions

```
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)
```

What does `sum_squares` need to know about `square`?

- `square` takes one argument.  
  Yes

- `square` has the intrinsic name square.  
  No

- `square` computes the square of a number.  
  Yes

- `square` computes the square by calling `mul`.  
  No

```
def square(x):
    return pow(x, 2)

def square(x):
    return mul(x, x-1) + x
```
Functional Abstractions

```python
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return square(x) + square(y)
```

What does `sum_squares` need to know about `square`?

- `square` takes one argument.  Yes
- `square` has the intrinsic name `square`.  No
- `square` computes the square of a number.  Yes
- `square` computes the square by calling `mul`.  No

```python
def square(x):
    return pow(x, 2)
def square(x):
    return mul(x, x - 1) + x
```

If the name “square” were bound to a built-in function, `sum_squares` would still work identically.
What is Data?
What is Data?

**Data**: the things that programs fiddle with
What is Data?

Data: the things that programs fiddle with
Primitive values are the simplest type of data
**Data**: the things that programs fiddle with

Primitive values are the simplest type of data

- Integers: 2, 3, 2013, -837592010
- Floating point (decimal) values: -4.5, 98.6
- Booleans: True, False
What is Data?

Data: the things that programs fiddle with

Primitive values are the simplest type of data

- Integers: 2, 3, 2013, -837592010
- Floating point (decimal) values: -4.5, 98.6
- Booleans: True, False

How do we represent more complex data?
What is Data?

**Data**: the things that programs fiddle with

Primitive values are the simplest type of data

- Integers: 2, 3, 2013, -837592010
- Floating point (decimal) values: -4.5, 98.6
- Booleans: True, False

How do we represent more complex data?
**Data**: the things that programs fiddle with

Primitive values are the simplest type of data
- Integers: 2, 3, 2013, -837592010
- Floating point (decimal) values: -4.5, 98.6
- Booleans: True, False

How do we represent more complex data?

CS61A Lecture 11
Amir Kamil
UC Berkeley
February 20, 2013
**Data**: the things that programs fiddle with

Primitive values are the simplest type of data

Integers: 2, 3, 2013, -837592010

Floating point (decimal) values: -4.5, 98.6

Booleans: True, False

How do we represent more complex data?

We need data abstractions!
Data Abstraction
Data Abstraction

Compound data combine smaller pieces of data together.
Data Abstraction

Compound data combine smaller pieces of data together

- A data: a year, month, and day
Data Abstraction

Compound data combine smaller pieces of data together

- A data: a year, month, and day
- A geographic position: latitude and longitude
Data Abstraction

Compound data combine smaller pieces of data together

- A data: a year, month, and day
- A geographic position: latitude and logitude

An abstract data type lets us manipulate compound data as a unit
Data Abstraction

Compound data combine smaller pieces of data together

- A data: a year, month, and day
- A geographic position: latitude and longitude

An abstract data type lets us manipulate compound data as a unit

Isolate two parts of any program that uses data
Data Abstraction

Compound data combine smaller pieces of data together

- A data: a year, month, and day
- A geographic position: latitude and longitude

An abstract data type lets us manipulate compound data as a unit

Isolate two parts of any program that uses data

- How data are represented (as parts)
Data Abstraction

Compound data combine smaller pieces of data together

- A data: a year, month, and day
- A geographic position: latitude and longitude

An abstract data type lets us manipulate compound data as a unit

Isolate two parts of any program that uses data

- How data are represented (as parts)
- How data are manipulated (as units)
Data Abstraction

Compound data combine smaller pieces of data together

- A data: a year, month, and day
- A geographic position: latitude and logitude

An abstract data type lets us manipulate compound data as a unit

Isolate two parts of any program that uses data

- How data are represented (as parts)
- How data are manipulated (as units)

Data abstraction: A methodology by which functions enforce an abstraction barrier between representation and use
Data Abstraction

Compound data combine smaller pieces of data together

- A data: a year, month, and day
- A geographic position: latitude and logitude

An abstract data type lets us manipulate compound data as a unit

Isolate two parts of any program that uses data

- How data are represented (as parts)
- How data are manipulated (as units)

Data abstraction: A methodology by which functions enforce an abstraction barrier between representation and use
Data Abstraction

Compound data combine smaller pieces of data together

- A data: a year, month, and day
- A geographic position: latitude and longitude

An abstract data type lets us manipulate compound data as a unit

Isolate two parts of any program that uses data

- How data are represented (as parts)
- How data are manipulated (as units)

Data abstraction: A methodology by which functions enforce an abstraction barrier between representation and use
Rational Numbers
Rational Numbers

\[
\frac{\text{numerator}}{\text{denominator}}
\]
Rational Numbers

\[
\frac{\text{numerator}}{\text{denominator}}
\]

Exact representation of fractions
Rational Numbers

\[
\frac{\text{numerator}}{\text{denominator}}
\]

Exact representation of fractions

A pair of integers
Rational Numbers

\[
\frac{\text{numerator}}{\text{denominator}}
\]

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!
Rational Numbers

\[
\begin{array}{c}
\text{numerator} \\
\hline \\
\text{denominator}
\end{array}
\]

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!

Assume we can compose and decompose rational numbers:
Rational Numbers

\[
\frac{\text{numerator}}{\text{denominator}}
\]

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!

Assume we can compose and decompose rational numbers:

- \texttt{rational}(n, d) returns a rational number $x$
Rational Numbers

\[
\begin{align*}
\text{numerator} & \quad \text{denominator} \\
\end{align*}
\]

Exact representation of fractions
A pair of integers
As soon as division occurs, the exact representation is lost!
Assume we can compose and decompose rational numbers:

- \( \text{rational}(n, d) \) returns a rational number \( x \)
- \( \text{numer}(x) \) returns the numerator of \( x \)
Rational Numbers

\[
\begin{array}{c}
\text{numerator} \\
\hline
\text{denominator}
\end{array}
\]

Exact representation of fractions
A pair of integers
As soon as division occurs, the exact representation is lost!
Assume we can compose and decompose rational numbers:

• \texttt{rational}(n, d) returns a rational number \(x\)
• \texttt{numer}(x) returns the numerator of \(x\)
• \texttt{denom}(x) returns the denominator of \(x\)
Rational Numbers

A pair of integers

As soon as division occurs, the exact representation is lost!

Assume we can compose and decompose rational numbers:

\[ \text{rational}(n, d) \] returns a rational number \( x \)

- \( \text{numer}(x) \) returns the numerator of \( x \)
- \( \text{denom}(x) \) returns the denominator of \( x \)
Rational Numbers

\[
\begin{array}{c}
\text{numerator} \\
\hline
\text{denominator}
\end{array}
\]

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!

Assume we can compose and decompose rational numbers:

- **Constructor**: \( \text{rational}(n, d) \) returns a rational number \( x \)
- **Selectors**
  - \( \text{numer}(x) \) returns the numerator of \( x \)
  - \( \text{denom}(x) \) returns the denominator of \( x \)
Example: General Form:
Rational Number Arithmetic

Example:

\[
\frac{3}{2} \times \frac{3}{5}
\]

General Form:
Rational Number Arithmetic

Example:

\[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]
Rational Number Arithmetic

Example:

\[
\frac{3}{2} \ast \frac{3}{5} = \frac{9}{10}
\]

General Form:

\[
\frac{nx}{dx} \ast \frac{ny}{dy}
\]
Example: \[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]

General Form:
\[
\frac{nx}{dx} \times \frac{ny}{dy} = \frac{nx \times ny}{dx \times dy}
\]
Rational Number Arithmetic

Example:

\[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]

\[
\frac{3}{2} + \frac{3}{5}
\]

General Form:

\[
\frac{nx}{dx} \times \frac{ny}{dy} = \frac{nx \times ny}{dx \times dy}
\]
Example:

\[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]

\[
\frac{3}{2} + \frac{3}{5} = \frac{21}{10}
\]

General Form:

\[
\frac{nx}{dx} \times \frac{ny}{dy} = \frac{nx \times ny}{dx \times dy}
\]
Example:

\[
\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}
\]

\[
\frac{3}{2} + \frac{3}{5} = \frac{21}{10}
\]

General Form:

\[
\frac{nx}{dx} \times \frac{ny}{dy} = \frac{nx \times ny}{dx \times dy}
\]

\[
\frac{nx}{dx} + \frac{ny}{dy}
\]
Example:

\[
\frac{3}{2} * \frac{3}{5} = \frac{9}{10}
\]

\[
\frac{3}{2} + \frac{3}{5} = \frac{21}{10}
\]

General Form:

\[
\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}
\]

\[
\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}
\]
Rational Number Arithmetic Code
• `rational(n, d)` returns a rational number $x$
• `numer(x)` returns the numerator of $x$
• `denom(x)` returns the denominator of $x$
Rational Number Arithmetic Code

- `rational(n, d)` returns a rational number \( x \)
- `numer(x)` returns the numerator of \( x \)
- `denom(x)` returns the denominator of \( x \)
Rational Number Arithmetic Code

def mul_rational(x, y):
    return rational(numer(x) * numer(y),
                    denom(x) * denom(y))

- `rational(n, d)` returns a rational number `x`
- `numer(x)` returns the numerator of `x`
- `denom(x)` returns the denominator of `x`
def mul_rational(x, y):
    return rational(numer(x) * numer(y),
                   denom(x) * denom(y))

- rational(n, d) returns a rational number x
- numer(x) returns the numerator of x
- denom(x) returns the denominator of x
Rational Number Arithmetic Code

```python
def mul_rational(x, y):
    return rational(numer(x) * numer(y),
                    denom(x) * denom(y))
```

- `rational(n, d)` returns a rational number $x$
- `numer(x)` returns the numerator of $x$
- `denom(x)` returns the denominator of $x$
Rational Number Arithmetic Code

```
def mul_rational(x, y):
    return rational(numer(x) * numer(y),
                    denom(x) * denom(y))

def add_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx * dy)

def eq_rational(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

- rational(n, d) returns a rational number x
- numer(x) returns the numerator of x
- denom(x) returns the denominator of x

Wishful thinking
Tuples
Tuples

```python
>>> pair = (1, 2)
```
Tuples

```python
>>> pair = (1, 2)
>>> pair
```
Tuples

```python
>>> pair = (1, 2)
>>> pair
(1, 2)
```
Tuples

```python
>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
```
Tuples

>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
Tuples

>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
Tuples

```python
>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
```
Tuples

```python
>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2
```
Tuples

```python
>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
```
Tuples

>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
Tuples

>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
Tuples

```python
>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
2
```
Tuples

```python
>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
```
Tuples

```python
>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
"""
Tuples

```python
>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
1
```
Tuples

```python
>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
```

Tuples

>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
2

>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
Tuples

>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
2

>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2

A tuple literal:
Comma-separated expression
Tuples

A tuple literal:
Comma-separated expression

"Unpacking" a tuple

>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
Tuples

>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
2
>>> from operator importgetitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2

A tuple literal:
Comma-separated expression

"Unpacking" a tuple

Element selection
Tuples

```python
>>> pair = (1, 2)
>>> pair
(1, 2)

>>> x, y = pair
>>> x
1
>>> y
2

>>> pair[0]
1
>>> pair[1]
2

>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
```

A tuple literal:
Comma-separated expression

"Unpacking" a tuple

Element selection

More tuples next lecture
Representing Rational Numbers
def rational(n, d):
    """Construct a rational number x that represents n/d."
    return (n, d)
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return (n, d)
Representing Rational Numbers

def rational(n, d):
    """Construct a rational number x that represents n/d."
    return (n, d)

from operator import getitem
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return (n, d)

from operator import getitem

def numer(x):
    """Return the numerator of rational number x."""
    return getitem(x, 0)
Representing Rational Numbers

def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return (n, d)

from operator import getitem

def numer(x):
    """Return the numerator of rational number x."""
    return getitem(x, 0)

def denom(x):
    """Return the denominator of rational number x."""
    return getitem(x, 1)
Representing Rational Numbers

def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return (n, d)

from operator import getitem

def numer(x):
    """Return the numerator of rational number x."""
    return getitem(x, 0)

def denom(x):
    """Return the denominator of rational number x."""
    return getitem(x, 1)