Announcements
Announcements

* Homework 3 due Tuesday 10/1 @ 11:59pm
* Optional Hog Contest entries due Thursday 10/3 @ 11:59pm
* Composition scores will be assigned this week (perhaps by Monday).
  - 3/3 is very rare on the first project.
  - You can gain back any points you lose on the first project by revising it (November).
Data Types

Every value has a type

demo
Data Types

Every value has a type

(demo)

Properties of native data types:
Data Types

Every value has a type

(demo)

Properties of native data types:

1. There are primitive expressions that evaluate to values of these types.
Data Types

Every value has a type

(demo)

Properties of native data types:

1. There are primitive expressions that evaluate to values of these types.
2. There are built-in functions, operators, and methods to manipulate those values.
Data Types

Every value has a type

(demo)

Properties of native data types:

1. There are primitive expressions that evaluate to values of these types.
2. There are built-in functions, operators, and methods to manipulate those values.

   Numeric types in Python:
Data Types

Every value has a type

(demo)

Properties of native data types:

1. There are primitive expressions that evaluate to values of these types.
2. There are built-in functions, operators, and methods to manipulate those values.

Numeric types in Python:

>>> type(2)
Data Types

Every value has a type

(demo)

Properties of native data types:

1. There are primitive expressions that evaluate to values of these types.
2. There are built-in functions, operators, and methods to manipulate those values.

Numeric types in Python:

```python
>>> type(2)
<class 'int'>
```
Data Types

Every value has a type

Properties of native data types:

1. There are primitive expressions that evaluate to values of these types.
2. There are built-in functions, operators, and methods to manipulate those values.

 numeric types in Python:

  >>> type(2)
  <class 'int'>

  >>> type(1.5)
Data Types

Every value has a type

demo

Properties of native data types:

1. There are primitive expressions that evaluate to values of these types.
2. There are built-in functions, operators, and methods to manipulate those values.

Numeric types in Python:

```python
>>> type(2)
<class 'int'>

>>> type(1.5)
<class 'float'>
```
Data Types

Every value has a type
(demo)

Properties of native data types:

1. There are primitive expressions that evaluate to values of these types.
2. There are built-in functions, operators, and methods to manipulate those values.

Numeric types in Python:

```python
>>> type(2)
<class 'int'>

>>> type(1.5)
<class 'float'>

>>> type(1+1j)
```
Data Types

Every value has a type

demo

Properties of native data types:

1. There are primitive expressions that evaluate to values of these types.
2. There are built-in functions, operators, and methods to manipulate those values.

Numeric types in Python:

>>> type(2)
<class 'int'>

>>> type(1.5)
<class 'float'>

>>> type(1+1j)
<class 'complex'>
Data Types

Every value has a type
(demo)

Properties of native data types:
1. There are primitive expressions that evaluate to values of these types.
2. There are built-in functions, operators, and methods to manipulate those values.

Numeric types in Python:

```python
>>> type(2)
<class 'int'>

>>> type(1.5)
<class 'float'>

>>> type(1+1j)
<class 'complex'>
```

Repsents integers exactly
Data Types

Every value has a type

(demo)

Properties of native data types:

1. There are primitive expressions that evaluate to values of these types.
2. There are built-in functions, operators, and methods to manipulate those values.

Numeric types in Python:

```python
>>> type(2)
<class 'int'> Represents integers exactly

>>> type(1.5)
<class 'float'> Represents real numbers approximately

>>> type(1+1j)
<class 'complex'>
```
Objects
Objects

• Objects represent information.
Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create *abstractions*.
Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create *abstractions*.
- Objects can represent things, but also properties, interactions, & processes.
Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create *abstractions*.
- Objects can represent things, but also properties, interactions, & processes.
- A type of object is called a class; classes are first-class values in Python.
Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create *abstractions*.
- Objects can represent things, but also properties, interactions, & processes.
- A type of object is called a class; classes are first-class values in Python.
- Object-oriented programming:
Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create abstractions.
- Objects can represent things, but also properties, interactions, & processes.
- A type of object is called a class; classes are first-class values in Python.
- Object-oriented programming:
  - A metaphor for organizing large programs
Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create *abstractions*.
- Objects can represent things, but also properties, interactions, & processes.
- A type of object is called a class; classes are first-class values in Python.
- Object-oriented programming:
  - A metaphor for organizing large programs
  - Special syntax that can improve the composition of programs
Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create **abstractions**.
- Objects can represent things, but also properties, interactions, & processes.
- A type of object is called a class; classes are first-class values in Python.
- Object-oriented programming:
  - A metaphor for organizing large programs
  - Special syntax that can improve the composition of programs
- In Python, every value is an object.
Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create **abstractions**.
- Objects can represent things, but also properties, interactions, & processes.
- A type of object is called a class; classes are first-class values in Python.
- Object-oriented programming:
  - A metaphor for organizing large programs
  - Special syntax that can improve the composition of programs
- In Python, every value is an object.
  - All objects have attributes.
Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create abstractions.
- Objects can represent things, but also properties, interactions, & processes.
- A type of object is called a class; classes are first-class values in Python.
- Object-oriented programming:
  - A metaphor for organizing large programs
  - Special syntax that can improve the composition of programs
- In Python, every value is an object.
  - All objects have attributes.
  - A lot of data manipulation happens through object methods.
Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create **abstractions**.
- Objects can represent things, but also properties, interactions, & processes.
- A type of object is called a class; classes are first-class values in Python.

Object-oriented programming:
- A metaphor for organizing large programs
- Special syntax that can improve the composition of programs

In Python, every value is an object.
- All objects have attributes.
- A lot of data manipulation happens through **object methods**.
- Functions do one thing; objects do many related things.
Objects

- Objects represent information.
- They consist of data and behavior, bundled together to create **abstractions**.
- Objects can represent things, but also properties, interactions, & processes.
- A type of object is called a class; classes are first-class values in Python.

Object-oriented programming:
- A metaphor for organizing large programs
- Special syntax that can improve the composition of programs

In Python, every value is an object.
- All objects have attributes.
- A lot of data manipulation happens through object **methods**.
- Functions do one thing; objects do many related things.

(Demo)
Data Abstraction
Data Abstraction
Data Abstraction

• Compound objects combine objects together
Data Abstraction

- Compound objects combine objects together
- A date: a year, a month, and a day
Data Abstraction

- Compound objects combine objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
Data Abstraction

- Compound objects combine objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- An abstract data type lets us manipulate compound objects as units
Data Abstraction

• Compound objects combine objects together
• A date: a year, a month, and a day
• A geographic position: latitude and longitude
• An *abstract data type* lets us manipulate compound objects as units
• Isolate two parts of any program that uses data:
Data Abstraction

• Compound objects combine objects together

• A date: a year, a month, and a day

• A geographic position: latitude and longitude

• An abstract data type lets us manipulate compound objects as units

• Isolate two parts of any program that uses data:
  ▪ How data are represented (as parts)
Data Abstraction

- Compound objects combine objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- An abstract data type lets us manipulate compound objects as units
- Isolate two parts of any program that uses data:
  - How data are represented (as parts)
  - How data are manipulated (as units)
Data Abstraction

• Compound objects combine objects together

• A date: a year, a month, and a day

• A geographic position: latitude and longitude

• An abstract data type lets us manipulate compound objects as units

• 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 objects combine objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- An *abstract data type* lets us manipulate compound objects as units
- 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 objects combine objects together
- A date: a year, a month, and a day
- A geographic position: latitude and longitude
- An abstract data type lets us manipulate compound objects as units
- 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

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

Exact representation of fractions

A pair of integers

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

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

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

Assume we can compose and decompose rational numbers:

- \text{rational}(n, d) \text{ returns a rational number } 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 may be lost!

Assume we can compose and decompose rational numbers:

* `rational(n, d)` returns a rational number \( x \)

* `numer(x)` returns the numerator of \( x \)
Rational Numbers

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

- `rational(n, d)` returns a rational number $x$
- `numer(x)` returns the numerator of $x$
- `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 may be lost!
Assume we can compose and decompose rational numbers:

\begin{itemize}
  \item `rational(n, d)` returns a rational number \( x \)
  \item `numer(x)` returns the numerator of \( x \)
  \item `denom(x)` returns the denominator of \( x \)
\end{itemize}
Rational Numbers

A rational number is a pair of integers.

As soon as division occurs, the exact representation may be lost!

Assume we can compose and decompose rational numbers:

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

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

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

Example

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

General Form

\[
\frac{nx}{dx} \times \frac{ny}{dy}
\]
Rational Number Arithmetic

Example

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

General Form

\[
\frac{nx}{dx} \times \frac{ny}{dy} = \frac{nx*ny}{dx*dy}
\]
Rational Number Arithmetic

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

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

Example

General Form

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

\[
\begin{align*}
\frac{3}{2} \times \frac{3}{5} &= \frac{9}{10} \\
\frac{3}{2} + \frac{3}{5} &= \frac{21}{10}
\end{align*}
\]

Example

General Form

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

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

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

**Example**

**General Form**

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

\[
\frac{nx}{dx} + \frac{ny}{dy}
\]
Rational Number Arithmetic

\[
\begin{array}{ccc}
\frac{3}{2} \times \frac{3}{5} &=& \frac{9}{10} \\
\frac{3}{2} + \frac{3}{5} &=& \frac{21}{10}
\end{array}
\]

Example

General Form

\[
\begin{array}{ccc}
\frac{nx}{dx} \times \frac{ny}{dy} &=& \frac{nx \times ny}{dx \times dy} \\
\frac{nx}{dx} + \frac{ny}{dy} &=& \frac{nx \times dy + ny \times dx}{dx \times dy}
\end{array}
\]
Rational Number Arithmetic Implementation

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

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

\[
\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx \cdot dy + ny \cdot dx}{dx \cdot dy}
\]
Rational Number Arithmetic Implementation

```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`
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 Implementation

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))

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)

- 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 Implementation

```python
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 equal_rational(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

- `rational(n, d)` returns a rational number \( \frac{n}{d} \)
- `numerator(x)` returns the numerator of \( x \)
- `denominator(x)` returns the denominator of \( x \)
Rational Number Arithmetic Implementation

```python
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 equal_rational(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

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

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

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

>>> pair = (1, 2)
>>> pair
(1, 2)
Pairs as Tuples

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

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

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

>>> x, y = pair
Pairs as Tuples

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

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

12
Pairs as Tuples

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

>>> pair[0]
1
>>> pair[1]
2
```
**Pairs as 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
```
Pairs as 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
```
Pairs as 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
```
Pairs as 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
```
Pairs as 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
```
Pairs as 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
**Pairs as 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
Pairs as 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
Pairs as 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)
Representing Rational Numbers

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

Construct a tuple
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)
Representing Rational Numbers

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

from operator importgetitem

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

def denom(x):
    """Return the denominator of rational number \( x \).""
    returngetitem(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)
Reducing to Lowest Terms

Example:
Reducing to Lowest Terms

Example:

\[
\begin{align*}
\frac{3}{2} \times \frac{5}{3}
\end{align*}
\]
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2}
\]
Reducing to Lowest Terms

Example:

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

\[
\frac{15}{6} \times \frac{1/3}{1/3} = \frac{5}{2}
\]
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2} \quad \frac{2}{5} + \frac{1}{10}
\]

\[
\frac{15}{6} \times \frac{1/3}{1/3} = \frac{5}{2}
\]
Reducing to Lowest Terms

Example:

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

\[
\frac{2}{5} + \frac{1}{10} = \frac{1}{2}
\]

\[
\frac{15}{6} \times \frac{1}{3} = \frac{5}{2}
\]
Reducing to Lowest Terms

Example:

$$\frac{3}{2} \times \frac{5}{3} = \frac{5}{2}$$

$$\frac{2}{5} + \frac{1}{10} = \frac{1}{2}$$

$$\frac{15}{6} \times \frac{1}{3} = \frac{5}{2}$$

$$\frac{25}{50} \times \frac{1}{25} = \frac{1}{2}$$
Reducing to Lowest Terms

Example:

\[
\begin{align*}
\frac{3}{2} \times \frac{5}{3} &= \frac{5}{2} \\
\frac{2}{5} + \frac{1}{10} &= \frac{1}{2}
\end{align*}
\]

\[
\begin{align*}
\frac{15}{6} \times \frac{1/3}{1/3} &= \frac{5}{2} \\
\frac{25}{50} \times \frac{1/25}{1/25} &= \frac{1}{2}
\end{align*}
\]

from fractions import gcd
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2} \quad \quad \quad \frac{2}{5} + \frac{1}{10} = \frac{1}{2}
\]

\[
\frac{15}{6} \times \frac{1/3}{1/3} = \frac{5}{2} \quad \quad \quad \frac{25}{50} \times \frac{1/25}{1/25} = \frac{1}{2}
\]

```python
from fractions import gcd

def rational(n, d):
```

from fractions import gcd

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

Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2} \quad \frac{2}{5} + \frac{1}{10} = \frac{1}{2}
\]

\[
\frac{15}{6} \times \frac{1}{3} = \frac{5}{2} \quad \frac{25}{50} \times \frac{1}{25} = \frac{1}{2}
\]
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2} \quad \frac{2}{5} + \frac{1}{10} = \frac{1}{2}
\]

\[
\frac{15}{6} \times \frac{1/3}{1/3} = \frac{5}{2} \quad \frac{25}{50} \times \frac{1/25}{1/25} = \frac{1}{2}
\]

```
from fractions import gcd

def rational(n, d):
    """Construct a rational number x that represents n/d."""
    g = gcd(n, d)
```
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2} \\
\frac{2}{5} + \frac{1}{10} = \frac{1}{2}
\]

\[
\frac{15}{6} \times \frac{1/3}{1/3} = \frac{5}{2} \\
\frac{25}{50} \times \frac{1/25}{1/25} = \frac{1}{2}
\]

```python
from fractions import gcd
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    g = gcd(n, d)
    return (n//g, d//g)
```
Reducing to Lowest Terms

Example:

\[
\frac{3}{2} \times \frac{5}{3} = \frac{5}{2} \\
\frac{2}{5} + \frac{1}{10} = \frac{1}{2}
\]

\[
\frac{15}{6} \times \frac{1}{3} = \frac{5}{2} \\
\frac{25}{50} \times \frac{1}{25} = \frac{1}{2}
\]

```python
from fractions import gcd

def rational(n, d):
    """Construct a rational number x that represents n/d."""
    g = gcd(n, d)
    return (n//g, d//g)
```
Abstraction Barriers
Abstraction Barriers

**Rational numbers as whole data values**

- add_rational
- mul_rational
- equal_rational

**Rational numbers as numerators & denominators**

- rational
- numer
- denom

**Rational numbers as tuples**

- tuple
- getitem

*However tuples are implemented in Python*
add_rational( (1, 2), (1, 4) )

def divide_rational(x, y):
    return (x[0] * y[1], x[1] * y[0])
Violating Abstraction Barriers

Does not use constructors

add_rational((1, 2), (1, 4))

def divide_rational(x, y):
    return (x[0] * y[1], x[1] * y[0])
Violating Abstraction Barriers

\[
\begin{align*}
\text{add_rational}( (1, 2), (1, 4) )
\end{align*}
\]

def divide_rational(x, y):
    return (x[0] * y[1], x[1] * y[0])
Violating Abstraction Barriers

add_rational((1, 2), (1, 4))

def divide_rational(x, y):
    return (x[0] * y[1], x[1] * y[0])

Does not use constructors
Twice!
No selectors!
Violating Abstraction Barriers

```python
add_rational( (1, 2), (1, 4) )

def divide_rational(x, y):
    return (x[0] * y[1], x[1] * y[0])
```

- Does not use constructors
- Twice!
- No selectors!
- And no constructor!
Violating Abstraction Barriers
Data Representations
What is Data?
What is Data?

• We need to guarantee that constructor and selector functions work together to specify the right behavior.
What is Data?

• We need to guarantee that constructor and selector functions work together to specify the right behavior.

• **Behavior condition:** If we construct rational number $x$ from numerator $n$ and denominator $d$, then $\text{numer}(x)/\text{denom}(x)$ must equal $n/d$. 
What is Data?

• We need to guarantee that constructor and selector functions work together to specify the right behavior.

• **Behavior condition:** If we construct rational number $x$ from numerator $n$ and denominator $d$, then $\text{numer}(x)/\text{denom}(x)$ must equal $n/d$.

• An abstract data type is some collection of selectors and constructors, together with some behavior condition(s).
What is Data?

• We need to guarantee that constructor and selector functions work together to specify the right behavior.

• **Behavior condition:** If we construct rational number \( x \) from numerator \( n \) and denominator \( d \), then \( \text{numer}(x)/\text{denom}(x) \) must equal \( n/d \).

• An abstract data type is some collection of selectors and constructors, together with some behavior condition(s).

• If behavior conditions are met, then the representation is valid.
What is Data?

• We need to guarantee that constructor and selector functions work together to specify the right behavior.

• **Behavior condition**: If we construct rational number \( x \) from numerator \( n \) and denominator \( d \), then \( \text{numer}(x)/\text{denom}(x) \) must equal \( n/d \).

• An abstract data type is some collection of selectors and constructors, together with some behavior condition(s).

• If behavior conditions are met, then the representation is valid.

You can recognize abstract data types by their behavior, not by their class...
Behavior Conditions of a Pair
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple.
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple.

But is that the only way to make pairs of values? *No!*
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple.

But is that the only way to make pairs of values? No!

Constructors, selectors, and behavior conditions:
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple. But is that the only way to make pairs of values? No!

Constructors, selectors, and behavior conditions:

If a pair p was constructed from elements x and y, then

• `getitem_pair(p, 0)` returns x, and
• `getitem_pair(p, 1)` returns y.
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple.

But is that the only way to make pairs of values? No!

Constructors, selectors, and behavior conditions:

If a pair p was constructed from elements x and y, then

- `getitem_pair(p, 0)` returns x, and
- `getitem_pair(p, 1)` returns y.

Together, selectors are the inverse of the constructor.
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple.

But is that the only way to make pairs of values? *No!*

Constructors, selectors, and behavior conditions:

If a pair \( p \) was constructed from elements \( x \) and \( y \), then

- \( \text{getitem}_\text{pair}(p, 0) \) returns \( x \), and
- \( \text{getitem}_\text{pair}(p, 1) \) returns \( y \).

Together, selectors are the inverse of the constructor

Generally true of *container types*.
To implement our rational number abstract data type, we used a two-element tuple.

But is that the only way to make pairs of values? *No!*

Constructors, selectors, and behavior conditions:

If a pair $p$ was constructed from elements $x$ and $y$, then

- `getitem_pair(p, 0)` returns $x$, and
- `getitem_pair(p, 1)` returns $y$.

Together, selectors are the inverse of the constructor.

Generally true of *container types.*

Not true for rational numbers because of GCD.
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple.

But is that the only way to make pairs of values? No!

Constructors, selectors, and behavior conditions:

If a pair p was constructed from elements x and y, then

- `getitem_pair(p, 0)` returns x, and
- `getitem_pair(p, 1)` returns y.

Together, selectors are the inverse of the constructor.

Generally true of container types.

(Demo)
Functional Pair Implementation

Example: http://goo.gl/9hVt8f
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch

Example: http://goo.gl/9hVt8f
def pair(x, y):
    """Return a functional pair."""

def dispatch(m):
    if m == 0:
        return x
    elif m == 1:
        return y
    return dispatch

This function represents a pair

Example: http://goo.gl/9hVt8f
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch

Example: http://goo.gl/9hVt8f
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch

def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p(i)

Example: http://goo.gl/9hVt8f
Functional Pair Implementation

```python
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch

def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p(i)
```

This function represents a pair

Constructor is a higher-order function

Selector defers to the object itself

Example: [http://goo.gl/9hVt8f](http://goo.gl/9hVt8f)
Functional Pair Implementation

def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch

def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p(i)

Example: http://goo.gl/9hVt8f
Functional Pair Implementation

```python
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch

def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p(i)
```

This function represents a pair.

Constructor is a higher-order function.

Selector defers to the object itself.

Example: [http://goo.gl/9hVt8f](http://goo.gl/9hVt8f)
Using a Functionally Implemented Pair

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

>>> getitem_pair(p, 0)
1

>>> getitem_pair(p, 1)
2
```
Using a Functionally Implemented Pair

```python
>>> p = pair(1, 2)
>>> getitem_pair(p, 0)
1
>>> getitem_pair(p, 1)
2
As long as we do not violate the abstraction barrier, we don't need to know that pairs are just functions
```
Using a Functionally Implemented Pair

```python
>>> p = pair(1, 2)
>>> getitem_pair(p, 0)
1
>>> getitem_pair(p, 1)
2
As long as we do not violate the abstraction barrier, we don't need to know that pairs are just functions
```

If a pair p was constructed from elements x and y, then

- `getitem_pair(p, 0)` returns x, and
- `getitem_pair(p, 1)` returns y.
Using a Functionally Implemented Pair

```python
>>> p = pair(1, 2)
>>> getitem_pair(p, 0)
1
>>> getitem_pair(p, 1)
2
```

As long as we do not violate the abstraction barrier, we don't need to know that pairs are just functions.

If a pair p was constructed from elements x and y, then

- `getitem_pair(p, 0)` returns x, and
- `getitem_pair(p, 1)` returns y.

This pair representation is valid!