61A Lecture 10

Wednesday, September 24

Announcements

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm
  - Homework party on Monday evening, details TBD

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm
  - Homework party on Monday evening, details TBD
- Optional Hog Contest entries due Wednesday 10/1 @ 11:59pm

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm
  - Homework party on Monday evening, details TBD
- Optional Hog Contest entries due Wednesday 10/1 @ 11:59pm
- Composition scores for Project 1 will mostly be assigned this week

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm
  - Homework party on Monday evening, details TBD
- Optional Hog Contest entries due Wednesday 10/1 @ 11:59pm
- Composition scores for Project 1 will mostly be assigned this week
  - 3/3 is unusual on the first project

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm
  - Homework party on Monday evening, details TBD
- Optional Hog Contest entries due Wednesday 10/1 @ 11:59pm
- Composition scores for Project 1 will mostly be assigned this week
  - 3/3 is unusual on the first project
  - You can gain back composition points you lost on Project 1 by revising it (in November)

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm
  - Homework party on Monday evening, details TBD
- Optional Hog Contest entries due Wednesday 10/1 @ 11:59pm
- Composition scores for Project 1 will mostly be assigned this week
  - 3/3 is unusual on the first project
  - You can gain back composition points you lost on Project 1 by revising it (in November)
- Midterm 1 should be graded by Friday

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm
  - Homework party on Monday evening, details TBD
- Optional Hog Contest entries due Wednesday 10/1 @ 11:59pm
- Composition scores for Project 1 will mostly be assigned this week
  - 3/3 is unusual on the first project
  - You can gain back composition points you lost on Project 1 by revising it (in November)
- Midterm 1 should be graded by Friday
  - Solutions to Midterm 1 will be posted after lecture

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm
  - Homework party on Monday evening, details TBD
- Optional Hog Contest entries due Wednesday 10/1 @ 11:59pm
- Composition scores for Project 1 will mostly be assigned this week
  - 3/3 is unusual on the first project
  - You can gain back composition points you lost on Project 1 by revising it (in November)
- Midterm 1 should be graded by Friday
  - Solutions to Midterm 1 will be posted after lecture
- Guerrilla section this Saturday 12-2 *and* 2:30-5 on recursion

Data

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)
<class 'int'>
```

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'>
```

Represents 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:

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

Represents integers exactly

```
>>> 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:

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

Represents integers exactly

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

Represents real numbers approximately

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'>
```

Represents integers exactly

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

Represents real numbers approximately

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

Objects

(Demo)

Objects

(Demo)

- Objects represent information.

Objects

(Demo)

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

Objects

(Demo)

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

Objects

(Demo)

- 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

(Demo)

- 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

(Demo)

- 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

(Demo)

- 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

(Demo)

- 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

(Demo)

- 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

(Demo)

- 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

(Demo)

- 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.

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***

All Programmers

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***

All Programmers

Great Programmers

Rational Numbers

Rational Numbers

$$\frac{\texttt{numerator}}{\texttt{denominator}}$$

Rational Numbers

```
numerator
─────────────
denominator
```

Exact representation of fractions

Rational Numbers

$$\frac{\texttt{numerator}}{\texttt{denominator}}$$

Exact representation of fractions

A pair of integers

Rational Numbers

numerator

denominator

Exact representation of fractions

A pair of integers

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

Rational Numbers

$$\frac{\texttt{numerator}}{\texttt{denominator}}$$

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

$$\frac{\texttt{numerator}}{\texttt{denominator}}$$

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

Rational Numbers

numerator

---

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

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

Rational Numbers

$$\frac{\texttt{numerator}}{\texttt{denominator}}$$

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:

Constructor

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

Rational Numbers

numerator
―――――――――――
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:

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

Example

General Form

Rational Number Arithmetic

$$\frac{3}{2} * \frac{3}{5}$$

Example

General Form

Rational Number Arithmetic

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

Example

General Form

Rational Number Arithmetic

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

Example

$$\frac{nx}{dx} * \frac{ny}{dy}$$

General Form

Rational Number Arithmetic

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

Example

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

General Form

Rational Number Arithmetic

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5}$$

Example

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

General Form

Rational Number Arithmetic

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$$

Example

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

General Form

Rational Number Arithmetic

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$$

Example

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy}$$

General Form

Rational Number Arithmetic

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$$

Example

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

General Form

Rational Number Arithmetic Implementation

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

- `rational(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))
```

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

- `rational(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))
```

Constructor

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

- rational(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))
```

Constructor

Selectors

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

- `rational(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))
```

Constructor

Selectors

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

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

These functions implement an abstract data type for rational numbers

Rational Number Arithmetic Implementation

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

Constructor

Selectors

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

```
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)
```

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

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

These functions implement an abstract data type for rational numbers

Rational Number Arithmetic Implementation

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

Constructor

Selectors

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

```
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 print_rational(x):
    print(numer(x), '/', denom(x))
```

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

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

These functions implement an abstract data type for rational numbers

Rational Number Arithmetic Implementation

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

Constructor

Selectors

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

```
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 print_rational(x):
    print(numer(x), '/', denom(x))

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

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

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

These functions implement an abstract data type for rational numbers

Pairs

Representing Pairs Using Lists

Representing Pairs Using Lists

```
>>> pair = [1, 2]
```

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]
```

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]
```

A list literal:
Comma-separated expressions in brackets

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]

>>> x, y = pair
```

```
A list literal:
Comma-separated expressions in brackets
```

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]

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

A list literal:
Comma-separated expressions in brackets

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]

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

A list literal:
Comma-separated expressions in brackets

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]

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

A list literal:
Comma-separated expressions in brackets

"Unpacking" a list

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]

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

>>> pair[0]
1
```

A list literal:
Comma-separated expressions in brackets

"Unpacking" a list

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]
```

A list literal:
Comma-separated expressions in brackets

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

"Unpacking" a list

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

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]
```

A list literal:
Comma-separated expressions in brackets

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

"Unpacking" a list

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

Element selection using the selection operator

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]
```

A list literal:
Comma-separated expressions in brackets

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

"Unpacking" a list

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

Element selection using the selection operator

```
>>> from operator import getitem
```

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]
```

A list literal:
Comma-separated expressions in brackets

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

"Unpacking" a list

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

Element selection using the selection operator

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

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]
```

A list literal:
Comma-separated expressions in brackets

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

"Unpacking" a list

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

Element selection using the selection operator

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

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]
```

A list literal:
Comma-separated expressions in brackets

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

"Unpacking" a list

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

Element selection using the selection operator

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

Element selection function

Representing Pairs Using Lists

```
>>> pair = [1, 2]
>>> pair
[1, 2]
```

A list literal:
Comma-separated expressions in brackets

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

"Unpacking" a list

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

Element selection using the selection operator

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

Element selection function

More lists next lecture

Representing Rational Numbers

```
def rational(n, d):
    """Construct a rational number that represents N/D."""
    return [n, d]
```

Representing Rational Numbers

```
def rational(n, d):
    """Construct a rational number that represents N/D."""
    return [n, d]
```

Construct a list

Representing Rational Numbers

```
def rational(n, d):
    """Construct a rational number that represents N/D."""
    return [n, d]
```

Construct a list

```
def numer(x):
    """Return the numerator of rational number X."""
    return x[0]
```

Representing Rational Numbers

```
def rational(n, d):
    """Construct a rational number that represents N/D."""
    return [n, d]
```

Construct a list

```
def numer(x):
    """Return the numerator of rational number X."""
    return x[0]

def denom(x):
    """Return the denominator of rational number X."""
    return x[1]
```

Representing Rational Numbers

```
def rational(n, d):
    """Construct a rational number that represents N/D."""
    return [n, d]
```

Construct a list

```
def numer(x):
    """Return the numerator of rational number X."""
    return x[0]

def denom(x):
    """Return the denominator of rational number X."""
    return x[1]
```

Select item from a list

Reducing to Lowest Terms

Example:

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3}$$

Reducing to Lowest Terms

Example:

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

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \boxed{\frac{5}{2}}$$

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

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \boxed{\frac{5}{2}} \qquad \frac{2}{5} + \frac{1}{10}$$

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

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \boxed{\frac{5}{2}} \qquad \frac{2}{5} + \frac{1}{10} = \frac{1}{2}$$

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

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \boxed{\frac{5}{2}} \qquad \frac{2}{5} + \frac{1}{10} = \boxed{\frac{1}{2}}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2} \qquad \frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \boxed{\frac{5}{2}} \qquad \frac{2}{5} + \frac{1}{10} = \boxed{\frac{1}{2}}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2} \qquad \frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```
from fractions import gcd
```

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \boxed{\frac{5}{2}} \qquad \frac{2}{5} + \frac{1}{10} = \boxed{\frac{1}{2}}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2} \qquad \frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```
from fractions import gcd

def rational(n, d):
```

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \boxed{\frac{5}{2}} \qquad \frac{2}{5} + \frac{1}{10} = \boxed{\frac{1}{2}}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2} \qquad \frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```python
from fractions import gcd

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

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \boxed{\frac{5}{2}} \qquad \frac{2}{5} + \frac{1}{10} = \boxed{\frac{1}{2}}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2} \qquad \frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```
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:

$$\frac{3}{2} * \frac{5}{3} = \boxed{\frac{5}{2}} \qquad \frac{2}{5} + \frac{1}{10} = \boxed{\frac{1}{2}}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2} \qquad \frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```
from fractions import gcd

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

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \boxed{\frac{5}{2}} \qquad \frac{2}{5} + \frac{1}{10} = \boxed{\frac{1}{2}}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2} \qquad \frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```
from fractions import gcd
```

Greatest common divisor

```
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

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation		

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational
Create rationals or implement rational operations		

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	`add_rational`, `mul_rational` `rationals_are_equal`, `print_rational`
Create rationals or implement rational operations	numerators and denominators	

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational
Create rationals or implement rational operations	numerators and denominators	rational, numer, denom

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	`add_rational`, `mul_rational` `rationals_are_equal`, `print_rational`
Create rationals or implement rational operations	numerators and denominators	`rational`, `numer`, `denom`

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational
Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
Implement selectors and constructor for rationals		

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational
Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
Implement selectors and constructor for rationals	two-element lists	

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational
Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
Implement selectors and constructor for rationals	two-element lists	list literals and element selection

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational
Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
Implement selectors and constructor for rationals	two-element lists	list literals and element selection

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational
Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
Implement selectors and constructor for rationals	two-element lists	list literals and element selection

Implementation of lists

Abstraction Barriers

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational
Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
Implement selectors and constructor for rationals	two-element lists	list literals and element selection

Implementation of lists

Violating Abstraction Barriers

```
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

Does not use constructors

Twice!

```
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

Twice!

```
add_rational( [1, 2], [1, 4] )

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

No selectors!

Violating Abstraction Barriers

Does not use constructors

Twice!

```
add_rational( [1, 2], [1, 4] )

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

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 numer(x)/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 numer(x)/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 numer(x)/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 numer(x)/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 list.

Behavior Conditions of a Pair

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

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 list.

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 list.

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
>
> - select(p, 0) returns x, and
> - select(p, 1) returns y.

Behavior Conditions of a Pair

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

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
>
> - select(p, 0) returns x, and
> - select(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 list.

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
>
> - select(p, 0) returns x, and
> - select(p, 1) returns y.

Together, selectors are the inverse of the constructor

Generally true of container types.

Behavior Conditions of a Pair

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

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
>
> - select(p, 0) returns x, and
> - select(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 list.

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

- select(p, 0) returns x, and
- select(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

(Demo)

Functional Pair Implementation

Interactive Diagram

Functional Pair Implementation

```
def pair(x, y):
    """Return a function that represents a pair."""
    def get(index):
        if index == 0:
            return x
        elif index == 1:
            return y
    return get
```

Interactive Diagram

Functional Pair Implementation

```
def pair(x, y):
    """Return a function that represents a pair."""
    def get(index):
        if index == 0:
            return x
        elif index == 1:
            return y
    return get
```

This function represents a pair

Interactive Diagram

Functional Pair Implementation

```
def pair(x, y):
    """Return a function that represents a pair."""
    def get(index):
        if index == 0:
            return x
        elif index == 1:
            return y
    return get
```

This function represents a pair

Constructor is a higher-order function

Interactive Diagram

Functional Pair Implementation

```
def pair(x, y):
    """Return a function that represents a pair."""
    def get(index):
        if index == 0:
            return x
        elif index == 1:
            return y
    return get
```

This function represents a pair

Constructor is a higher-order function

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


Interactive Diagram

Functional Pair Implementation

```
def pair(x, y):
    """Return a function that represents a pair."""
    def get(index):
        if index == 0:
            return x
        elif index == 1:
            return y
    return get
```

This function represents a pair

Constructor is a higher-order function

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

Selector defers to the object itself

Interactive Diagram

Functional Pair Implementation

```
def pair(x, y):
    """Return a function that represents a pair."""
    def get(index):
        if index == 0:
            return x
        elif index == 1:
            return y
    return get
```

This function represents a pair

Constructor is a higher-order function

```
point = pair(2, 4)
select(point, 1)
```

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

Selector defers to the object itself

Interactive Diagram

Functional Pair Implementation

```
def pair(x, y):
    """Return a function that represents a pair."""
    def get(index):
        if index == 0:
            return x
        elif index == 1:
            return y
    return get
```

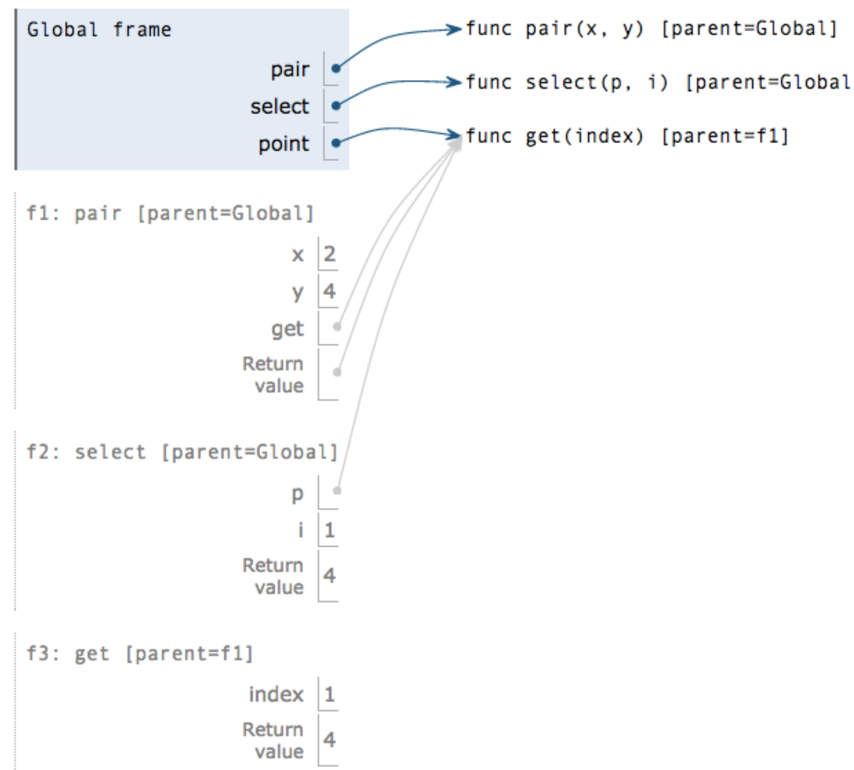
This function represents a pair

Constructor is a higher-order function

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

Selector defers to the object itself

```
point = pair(2, 4)
select(point, 1)
```



Interactive Diagram

Using a Functionally Implemented Pair

Using a Functionally Implemented Pair

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

>>> select(p, 0)
1

>>> select(p, 1)
2
```

Using a Functionally Implemented Pair

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

>>> select(p, 0)
1

>>> select(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

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

>>> select(p, 0)
1

>>> select(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

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

Using a Functionally Implemented Pair

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

>>> select(p, 0)
1

>>> select(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

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

This pair representation is valid!