61A Lecture 8

Wednesday, September 12
Data Abstraction
Data Abstraction

- Compound objects combine primitive objects together
Data Abstraction

• Compound objects combine primitive objects together

• A date: a year, a month, and a day
Data Abstraction

• Compound objects combine primitive objects together

• A date: a year, a month, and a day

• A geographic position: latitude and longitude
Data Abstraction

• Compound objects combine primitive 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 primitive 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 primitive 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 primitive 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 primitive 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 primitive 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 primitive 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

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

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 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

A pair of integers

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

Assume we can compose and decompose rational numbers:

- rational(n, d) returns a rational number x
Rational Numbers

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 $x$
Rational Numbers

```
numerator
---
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:

```
Constructor 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 is lost!
Assume we can compose and decompose rational numbers:

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

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

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 \( x \)
- \[ \text{denom}(x) \] returns the denominator of \( x \)
**Rational Number Arithmetic**

<table>
<thead>
<tr>
<th>Example:</th>
<th>General Form:</th>
</tr>
</thead>
</table>


Rational Number Arithmetic

Example:

General Form:

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

Example:

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

General Form:
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:

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

**Example:**

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

**General Form:**

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

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

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

Example:

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

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

General Form:

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

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

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

Wishful thinking

- `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))
```

- `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))
```

- `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)

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
Tuples
Tuples

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

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

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

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

A tuple literal:
Comma-separated expression
Tuples

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

A tuple literal:
Comma-separated expression

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

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

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

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

A tuple literal:
Comma-separated expression
Tuples

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

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

A tuple literal:
Comma-separated expression
Tuples

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

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

A tuple literal:
Comma-separated expression
Tuples

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

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

A tuple literal:
Comma-separated expression
Tuples

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

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

A tuple literal:
Comma-separated expression

"Unpacking" a tuple
**Tuples**

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

A tuple literal:
Comma-separated expression

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

"Unpacking" a tuple

>>> pair[0]
Tuples

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

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

>>> pair[0]
1
```

A tuple literal:
Comma-separated expression

"Unpacking" a tuple
Tuples

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

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

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

A tuple literal: Comma-separated expression

"Unpacking" a tuple
Tuples

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

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

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

A tuple literal:
Comma-separated expression

"Unpacking" a tuple
Tuples

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

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

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

A tuple literal: Comma-separated expression

"Unpacking" a tuple

```python
>>> 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)
```

A tuple literal:
Comma-separated expression

"Unpacking" a tuple
### 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
```

A tuple literal:
Comma-separated expression

"Unpacking" a tuple
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)

A tuple literal:
Comma-separated expression

"Unpacking" a tuple
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
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

"Unpacking" a tuple

Element selection
Tuples

```python
>>> pair = (1, 2)  # A tuple literal: Comma-separated expression
(1, 2)

>>> x, y = pair  # "Unpacking" a tuple
>>> x
1
>>> y
2

>>> pair[0]  # Element selection
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2

More tuples next lecture
Representing Rational Numbers
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 importgetitem

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

```python
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:

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

Example:

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

Example:

\[
\begin{align*}
\frac{3}{2} \times \frac{5}{3} &= \frac{5}{2} \\
\frac{15}{6} \times \frac{1/3}{1/3} &= \frac{5}{2}
\end{align*}
\]
Reducing to Lowest Terms

Example:

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

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

Example:

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

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

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

Example:

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

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

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

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

Example:

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

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

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

\[
\frac{25}{50} \times \frac{1/25}{1/25} = \frac{1}{2}
\]

from fractions import gcd
Reducing to Lowest Terms

Example:

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

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

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

\[
\frac{25}{50} \times \frac{1}{25} = \frac{1}{2}
\]

from fractions import (gcd)

Greatest common divisor
Reducing to Lowest Terms

Example:

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

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

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

\[
\frac{25}{50} \times \frac{1/25}{1/25} = \frac{1}{2}
\]

```
from fractions import gcd

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

Rational numbers as whole data values

add_rationals  mul_rationals  eq_rationals

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

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

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

\[
\text{add\_rational}( (1, 2), (1, 4) )
\]

\[
\text{def divide\_rational}(x, y):
    \text{return } (x[0] \ast y[1], x[1] \ast y[0])
\]
Violating Abstraction Barriers
What is Data?
What is Data?

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

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

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

• We need to guarantee that constructor and selector functions together 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 together 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, the representation is valid.
What is Data?

• We need to guarantee that constructor and selector functions together 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, the representation is valid.

You can recognize data types by behavior, not by bits
Behavior Conditions of a Pair
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

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

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

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 \( y \).
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

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.
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

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.
Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

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.

Not true for rational numbers because of GCD
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 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
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

Constructor is a higher-order function
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)
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
```

This function represents a pair

Constructor is a higher-order function

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

Selector defers to the object itself
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)
```

```python
>>> getitem_pair(p, 0)
1
```

```python
>>> 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)
```

```python
>>> getitem_pair(p, 0)
1
```

```python
>>> 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!