61A Lecture 10

Wednesday, September 25
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
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:

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

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

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

• 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

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 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

\[
\begin{align*}
\frac{nx}{dx} \times \frac{ny}{dy} &= \frac{nx \times ny}{dx \times dy} \\
\frac{nx}{dx} + \frac{ny}{dy} &= \frac{nx \times dy + ny \times dx}{dx \times dy}
\end{align*}
\]

General Form
Rational Number Arithmetic Implementation

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

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
```

```
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)
```

• These functions implement an abstract data type for rational numbers
Pairs
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)

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)
Reduction 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

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.
Violating Abstraction Barriers

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

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

Does not use constructors

Twice!

No selectors!

And no constructor!
Data Representations
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

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)
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
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!