Announcements

Roadmap

- This week (Data), the goals are:
  - To continue our journey through abstraction with data abstraction
  - To study useful data types we can construct with data abstraction

List Comprehensions (demo)

A combined expression that evaluates to a list using this evaluation procedure:
1. Add a new frame with the current frame as its parent
2. Create an empty result list
3. For each element in the sequence from <seq exp>:
   1. Bind <name> to that element in the new frame
   2. If <filter exp> evaluates to a true value, then add the value of <map exp> to the result list

Data Abstraction

- Python (and other languages) implements for us some primitive data types, such as numbers and strings
- But most data that we care about are compound values, rather than just a single value like a number or string
  - A date is three numbers: year, month, and day
  - A location is two numbers: latitude and longitude
- Data abstraction allows us to manipulate compound values as units, rather than having to deal with their parts

Data Abstraction

- Great programmers use data abstraction to separate:
  - How compound values are represented (the parts)
  - How compound values are used (the unit)
  - This leads to programs that are more understandable, easier to maintain, and just better in general
- The separation is called the abstraction barrier
- Most important thing I’ll say today:
  
  Never violate the abstraction barrier!
Example: Rational Numbers

- Rational numbers are numbers that can be expressed as \( \frac{n}{d} \)
  where \( n \) and \( d \) are both integers

- So a rational number can be represented as two numbers, making it a compound value

- This is an exact representation of fractions
  - If we instead use floats to represent fractions, we can lose the exact representation if we perform division

### Representing Rational Numbers

- To represent a compound data type, we must have:
  1. **Constructors** that allow us to construct new instances of the data type
  2. **Selectors** that allow us to access the different parts of the data type

- These are typically both functions

```python
def rational(n, d):
    """Return the rational number with numerator n and denominator d."""
    ...

def numer(rat):
    """Return the numerator of the rational number rat."""
    ...

def denom(rat):
    """Return the denominator of the rational number rat."""
    ...
```

### Implementing Rational Numbers

- There are many different ways we could choose to implement rational numbers
  - One of the simplest is to use lists

```python
from fractions import gcd  # Greatest common divisor

def rational(n, d):
    """Return the rational number with numerator n and denominator d."""
    divisor = gcd(n, d)  # Reduce to lowest terms
    return [n//divisor, d//divisor]

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

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

### The Abstraction Barrier

- The almighty abstraction barrier!
Abstraction Barrier Violations

- Constructors and selectors provide us with abstraction, allowing us to use the data type without having to know its implementation.
- An abstraction barrier violation is when we assume knowledge about the data type implementation, rather than using constructors and selectors.
- Remember the most important thing I’ll say today: Never violate the abstraction barrier!

- Why is this such a bad thing?

No selectors! No constructor either!

Abstraction Barrier Violations

from fractions import gcd
def mul_rational(rat1, rat2):
    return [rat1[0] * rat2[0], rat1[1] * rat2[1]]
def rational(n, d):
    divisor = gcd(n, d)
    return [n // divisor, d // divisor]
def numer(rat):
    return rat[0]
def denom(rat):
    return rat[1]

# You write many more lines of code
# with abstraction barrier violations...

A Dictionary Abstract Data Type

Summary

- Data abstraction provides us with a powerful set of ideas for working with compound values.
  - Using abstraction allows us to think about data types in terms of units and parts, rather than worrying about the implementation.
  - This leads to programs that are easier to maintain and easier to understand.
- An abstraction barrier violation is when we assume knowledge about the underlying data type implementation.
  - One more time for emphasis: Never violate the abstraction barrier!