Containers + Data Abstraction
Announcements
Lists

['Demo']
Working with Lists

>>> digits = [1, 8, 2, 8]

The number of elements

>>> len(digits)
4

An element selected by its index

>>> digits[3]
8

Concatenation and repetition

>>> digits * 2
[1, 8, 2, 8, 1, 8, 2, 8]

>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]

Nested lists

>>> pairs = [[10, 20], [30, 40]]

>>> pairs[1]
[30, 40]

>>> pairs[1][0]
30

>>> digits = [2/2, 2+2+2+2, 2, 2*2*2]

>>> digits
[1, 8, 2, 8, 1, 8, 2, 8, 1, 8, 2, 8]

>>> getitem(digits, 3)
8

>>> add([2, 7], mul(digits, 2))
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
Containers
Built-in operators for testing whether an element appears in a compound value / container.

```python
>>> digits = [1, 8, 2, 8]
>>> 1 in digits
True
>>> 8 in digits
True
>>> 5 not in digits
True
>>> not(5 in digits)
True
```
For Statements

(Demo)
def count(s, value):
    total = 0
    for element in s:
        if element == value:
            total = total + 1
    return total
For Statement Execution Procedure

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header `<expression>`, which must yield an iterable value (a sequence)

2. For each element in that sequence, in order:
   
   A. Bind `<name>` to that element in the current frame

   B. Execute the `<suite>`
Sequence Unpacking in For Statements

A sequence of fixed-length sequences

```python
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]

>>> same_count = 0
```

A name for each element in a fixed-length sequence

```python
>>> for x, y in pairs:
...    if x == y:
...        same_count = same_count + 1

>>> same_count
2
```

Each name is bound to a value, as in multiple assignment.
Ranges
The Range Type

A range is a sequence of consecutive integers.*

\[
..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...
\]

Length: ending value - starting value

Element selection: starting value + index

\[
\begin{align*}
\text{list}(\text{range}(-2, 2)) & \quad \text{List constructor} \\
[-2, -1, 0, 1] & \\
\text{list}(\text{range}(4)) & \\
[0, 1, 2, 3] & \quad \text{Range with a 0 starting value}
\end{align*}
\]

* Ranges can actually represent more general integer sequences.
List Comprehensions

```python
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o', 'p']
>>> [letters[i] for i in [3, 4, 6, 8]]
['d', 'e', 'm', 'o']
```
List Comprehensions

[<map exp> for <name> in <iter exp> if <filter exp>]

Short version: [<map exp> for <name> in <iter exp>]

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 that is the value of the expression
3. For each element in the iterable value of <iter exp>:
   A. Bind <name> to that element in the new frame from step 1
   B. If <filter exp> evaluates to a true value, then add the value of <map exp> to the result list
Strings
Strings are an Abstraction

Representing data:

'Hello'   '1.2e-5'   'False'   '[1, 2]'

Representing language:

"""According to all known laws of aviation, there is no way a bee should be able to fly. Its wings are too small to get its fat little body off the ground. The bee, of course, flies anyway because bees don't care what humans think is impossible."

Representing programs:

'curry = lambda f: lambda x: lambda y: f(x, y)'

(Demo)
Here are Three Forms of String Literals

```python
>>> 'I am string! 您好'
'I am string! 您好'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> """"The Zen of Python claims, Readability counts.
Read more: import this.""
'The Zen of Python claims, Readability counts.\nRead more: import this.'
```

- A backslash "escapes" the following character
- "Line feed" character represents a new line

Single-quoted and double-quoted strings are equivalent
Dictionaries

{"Dem": 0}
Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

• A key of a dictionary **cannot be** a list or a dictionary (or any mutable type)

• Two **keys cannot be equal**; There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction

If you want to associate multiple values with a key, store them all in a sequence value
Containers - Summary

• Containers store values
  ▪ We can ask for their length and index into them
  ▪ We can iterate over containers using for statements and while statements

• 4 types of containers today
  ▪ Lists — flexible, store any values at all — i.e. [1, 2, “hello”]
  ▪ Ranges — store a range of integers — i.e. range(1, 5)
  ▪ Strings — store a collection of characters — i.e. “hello”
  ▪ Dictionaries — store key-value mappings — i.e. {“h”: “ello”, “w”: “orld”}
Data Abstraction
Data Abstraction

• Compound values combine other values together
  ▪ A date: a year, a month, and a day
  ▪ A geographic position: latitude and longitude

• Data abstraction lets us manipulate compound values 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! (Demo)

Assume we can compose and decompose the representation of rational numbers:

Constructor

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

Selectors

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

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

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

**Example**

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

\[
\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx \times dy + ny \times dx}{dx \times dy}
\]
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 print_rational(x):
    print(numer(x), '/', denom(x))

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

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]
from math import gcd

def rational(n, d):
    """Construct a rational that represents n/d in lowest terms."""
    g = gcd(n, d)
    return [n//g, d//g]

(Demo?)
Abstraction Barriers
### Abstraction Barriers

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

**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] ]