Lists

['Demo']
Working with Lists

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

The number of elements
>>> len(digits)
4

An element selected by its index
>>> digits[3]
8

Concatenation and repetition
>>> [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
```

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

>>> getitem(digits, 3)
8

>>> add([[2, 7], mul(digits, 2))
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```
Containers
Containers

Built-in operators for testing whether an element appears in a compound value

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

(Demo)
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
```

Each name is bound to a value, as in multiple assignment

```python
>>> same_count
2
```
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

>>> list(range(-2, 2))
[-2, -1, 0, 1]

List constructor

Range with a 0 starting value

>>> list(range(4))
[0, 1, 2, 3]

* 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

\[
\left[ \text{map exp} \, \text{for} \, \text{name} \, \text{in} \, \text{iter exp} \, \text{if} \, \text{filter exp}\right]
\]

Short version: \[
\left[ \text{map exp} \, \text{for} \, \text{name} \, \text{in} \, \text{iter exp}\right]
\]

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 \text{iter exp}:
   A. Bind \text{name} to that element in the new frame from step 1
   B. If \text{filter exp} evaluates to a true value, then add the value of \text{map exp} to the result list
Strings
Strings are an Abstraction

Representing data:

'200'  '1.2e-5'  'False'  '[1, 2]'  

Representing language:

"""And, as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name."
"""

Representing programs:

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

(Demo)
String Literals Have Three Forms

>>> 'I am string!'
'I am string!'

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

>>> '您好'
'您好'

>>> """The Zen of Python
claims, Readability counts.
Read more: import this.""
'The Zen of Python
claims, Readability counts.
Read 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 unordered 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
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 rational numbers:

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

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

General Form
Rational Number Arithmetic Implementation

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

These functions implement an abstract representation for rational numbers
Pairs
Representing Pairs Using Lists

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

A list literal:
Comma-separated expressions in brackets

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

"Unpacking" a list

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

Element selection using the selection operator

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

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

Element selection function

```
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
```
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]

(Demo)
Reducing to Lowest Terms

Example:

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

\[
\frac{15}{6} \times \frac{1/3}{1/3} = \frac{5}{2} \\
\frac{25}{50} \times \frac{1/25}{1/25} = \frac{1}{2}
\]

```
from fractions 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

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!
Data Representations
What are 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 \( \text{numerator}(x)/\text{denominator}(x) \) must equal \( n/d \).

- Data abstraction uses selectors and constructors to define behavior.

- If behavior conditions are met, then the representation is valid.

You can recognize an abstract data representation by its behavior.

(Demo)
Rationals Implemented as Functions

```python
def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select

def numer(x):
    return x('n')

def denom(x):
    return x('d')
```

This function represents a rational number

Constructor is a higher-order function

Selector calls x

```
x = rational(3, 8)
numer(x)
```
Dictionaries

{"Dem": 0}
Limitations on Dictionaries

Dictionaries are *unordered* 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