Data Abstraction

Class outline:

- Lecture 11 follow-ups
- Data abstraction
- Dictionaries

Data abstraction

Data abstractions

Many values in programs are compound values, a value composed of other values.

- A date: a year, a month, and a day
- A geographic position: latitude and longitude

A **data abstraction** lets us manipulate compound values as units, without needing to worry about the way the values are stored.

A pair abstraction

If we needed to frequently manipulate "pairs" of values in our program, we could use a `pair` data abstraction.

`pair(a, b)`	constructs a new pair from the two arguments.
`first(pair)`	returns the first value in the given pair.
`second(pair)`	returns the second value in the given pair.

```
couple = pair("Neil", "David")
neil = first(couple)      # 'Neil'
david = second(couple)    # 'David'
```

A pair implementation

Only the developers of the `pair` abstraction needs to know/decide how to implement it.

```
def pair(a, b):


def first(pair):


def second(pair):

```

How else could it be implemented?

A pair implementation

Only the developers of the `pair` abstraction needs to know/decide how to implement it.

```
def pair(a, b):
    return [a, b]

def first(pair):



def second(pair):

```

How else could it be implemented?

A pair implementation

Only the developers of the `pair` abstraction needs to know/decide how to implement it.

```
def pair(a, b):
    return [a, b]

def first(pair):
    return pair[0]

def second(pair):
```

How else could it be implemented?

A pair implementation

Only the developers of the `pair` abstraction needs to know/decide how to implement it.

```
def pair(a, b):
    return [a, b]

def first(pair):
    return pair[0]

def second(pair):
    return pair[1]
```

How else could it be implemented?

Rational abstraction

Rational numbers

If we needed to represent fractions exactly...

$$\frac{numerator}{denominator}$$

We could use this data abstraction:

Constructor	`rational(n,d)`	constructs a new rational number.
Selectors	`numer(rat)`	returns the numerator of the given rational number.
	`denom(rat)`	returns the denominator of the given rational number.

```
quarter = rational(1, 4)
top = numer(quarter)   # 1
bot = denom(quarter)   # 4
```

Rational number arithmetic

Example	**General form**
$\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}$	$\frac{n_x}{d_x} \times \frac{n_y}{d_y} = \frac{n_x \times n_y}{d_x \times d_y}$
$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$	$\frac{n_x}{d_x} + \frac{n_y}{d_y} = \frac{n_x \times d_y + n_y \times d_x}{d_x \times d_y}$

Rational number arithmetic code

We can implement arithmetic using the data abstractions:

Implementation	**General form**
`def mul_rational(x, y): return rational( numer(x) * numer(y), denom(x) * denom(y))`	$\frac{n_x}{d_x} \times \frac{n_y}{d_y} = \frac{n_x \times n_y}{d_x \times d_y}$
`def add_rational(x, y): (nx, dx) = numer(x), denom(x) (ny, dy) = numer(y), denom(y) return rational(nx * dy + ny * dx`	$\frac{n_x}{d_x} + \frac{n_y}{d_y} = \frac{n_x \times d_y + n_y \times d_x}{d_x \times d_y}$

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

$$\frac{n_x}{d_x} \times \frac{n_y}{d_y} = \frac{n_x \times n_y}{d_x \times d_y}$$

```
def add_rational(x, y):
    (nx, dx) = numer(x), denom(x)
    (ny, dy) = numer(y), denom(y)
    return rational(nx * dy + ny * dx
```

$$\frac{n_x}{d_x} + \frac{n_y}{d_y} = \frac{n_x \times d_y + n_y \times d_x}{d_x \times d_y}$$

```
mul_rational( rational(3, 2), rational(3, 5))
add_rational( rational(3, 2), rational(3, 5))
```

Rational numbers utilities

A few more helpful functions:

```
def print_rational(x):
    print(numer(x), '/', denom(x))
```

```
def rationals_are_equal(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

```
print_rational( rational(3, 2) )    # 3/2
rationals_are_equal( rational(3, 2), rational(3, 2) ) # True
```

Rational numbers implementation

```
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]
```

Reducing to lowest terms

What's the current problem with...

```
add_rational( rational(3, 4), rational(2, 16) )  # 56/64
add_rational( rational(3, 4), rational(4, 16) )  # 64/64
```

Reducing to lowest terms

What's the current problem with...

```
add_rational( rational(3, 4), rational(2, 16) )  # 56/64
add_rational( rational(3, 4), rational(4, 16) )  # 64/64
```

$$\frac{3}{2} \times \frac{5}{3} = \frac{15}{6}$$

Multiplication results in a non-reduced fraction...

$$\frac{15 \div 3}{6 \div 3} = \frac{5}{2}$$

...so we always divide top and bottom by GCD!

Reducing to lowest terms

What's the current problem with...

```
add_rational( rational(3, 4), rational(2, 16) )  # 56/64
add_rational( rational(3, 4), rational(4, 16) )  # 64/64
```

$$\frac{3}{2} \times \frac{5}{3} = \frac{15}{6}$$

Multiplication results in a non-reduced fraction...

$$\frac{15 \div 3}{6 \div 3} = \frac{5}{2}$$

...so we always divide top and bottom by GCD!

```
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]
```

Using rationals

User programs can use the rational data abstraction for their own specific needs.

```
def exact_harmonic_number(n):
    """Return 1 + 1/2 + 1/3 + ... + 1/N as a rational
    s = rational(0, 1)
    for k in range(1, n + 1):
        s = add_rat(s, rational(1, k))
    return s
```

Abstraction barriers

Layers of abstraction

Primitive Representation	`[..,..]` `[0]` `[1]`
Data abstraction	`make_rat()` `numer()` `denom()`
	`add_rat()` `mul_rat()` `print_rat()` `equal_rat()`
User program	`exact_harmonic_number()`

Each layer only uses the layer above it.

Violating abstraction barriers

What's wrong with...

```
add_rational( [1, 2], [1, 4] )
```

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

Violating abstraction barriers

What's wrong with...

```
add_rational( [1, 2], [1, 4] )
# Doesn't use constructors!
```

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

Violating abstraction barriers

What's wrong with...

```
add_rational( [1, 2], [1, 4] )
# Doesn't use constructors!
```

```
def divide_rational(x, y):
    return [ x[0] * y[1], x[1] * y[0] ]
    # Doesn't use selectors!
```

Other rational implementations

The `rational()` data abstraction could use an entirely different underlying representation.

```
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')
```

View example usage in PythonTutor

Data types

Review: Python types

Type	Examples
Integers	`0 -1 0xFF 0b1101`
Booleans	`True False`
Functions	`def f(x)...` `lambda x: ...`
Strings	`"pear"` `"I say, \"hello!\""`
Ranges	`range(11) range(1, 6)`
Lists	`[] ["apples", "bananas"]` `[x**3 for x in range(2)]`

Dictionaries

Dictionaries

A `dict` is a mutable mapping of key-value pairs

```
states = {
        "CA": "California",
        "DE": "Delaware",
        "NY": "New York",
        "TX": "Texas",
        "WY": "Wyoming"
}
```

Queries:

```
>>> len(states)
```

```
>>> "CA" in states
```

```
>>> "ZZ" in states
```

Dictionaries

A `dict` is a mutable mapping of key-value pairs

```
states = {
        "CA": "California",
        "DE": "Delaware",
        "NY": "New York",
        "TX": "Texas",
        "WY": "Wyoming"
}
```

Queries:

```
>>> len(states)
5
```

```
>>> "CA" in states
```

```
>>> "ZZ" in states
```

Dictionaries

A `dict` is a mutable mapping of key-value pairs

```
states = {
        "CA": "California",
        "DE": "Delaware",
        "NY": "New York",
        "TX": "Texas",
        "WY": "Wyoming"
}
```

Queries:

```
>>> len(states)
5
```

```
>>> "CA" in states
True
```

```
>>> "ZZ" in states
```

Dictionaries

A `dict` is a mutable mapping of key-value pairs

```
states = {
        "CA": "California",
        "DE": "Delaware",
        "NY": "New York",
        "TX": "Texas",
        "WY": "Wyoming"
}
```

Queries:

```
>>> len(states)
5
```

```
>>> "CA" in states
True
```

```
>>> "ZZ" in states
False
```

Dictionary selection

```
words = {
         "más": "more",
         "otro": "other",
         "agua": "water"
}
```

Select a value:

```
>>> words["otro"]
```

```
>>> first_word = "agua"
>>> words[first_word]
```

```
>>> words["pavo"]
```

```
>>> words.get("pavo", "")
```

Dictionary selection

```
words = {
          "más": "more",
          "otro": "other",
          "agua": "water"
}
```

Select a value:

```
>>> words["otro"]
'other'
```

```
>>> first_word = "agua"
>>> words[first_word]
```

```
>>> words["pavo"]
```

```
>>> words.get("pavo", "")
```

Dictionary selection

```
words = {
        "más": "more",
        "otro": "other",
        "agua": "water"
}
```

Select a value:

```
>>> words["otro"]
'other'
```

```
>>> first_word = "agua"
>>> words[first_word]
'water'
```

```
>>> words["pavo"]
```

```
>>> words.get("pavo", "")
```

Dictionary selection

```
words = {
        "más": "more",
        "otro": "other",
        "agua": "water"
}
```

Select a value:

```
>>> words["otro"]
'other'
```

```
>>> first_word = "agua"
>>> words[first_word]
'water'
```

```
>>> words["pavo"]
KeyError: pavo
```

```
>>> words.get("pavo", "")
```

Dictionary selection

```
words = {
        "más": "more",
        "otro": "other",
        "agua": "water"
}
```

Select a value:

```
>>> words["otro"]
'other'
```

```
>>> first_word = "agua"
>>> words[first_word]
'water'
```

```
>>> words["pavo"]
KeyError: pavo
```

```
>>> words.get("pavo", "")
''
```

Dictionary rules

- A key **cannot** be a list or dictionary (or any mutable type)
- All keys in a dictionary are distinct (there can only be one value per key)
- The values can be any type, however!

```
spiders = {
  "smeringopus": {
          "name": "Pale Daddy Long-leg",
          "length": 7
  },
  "holocnemus pluchei": {
          "name": "Marbled cellar spider",
          "length": (5, 7)
  }
}
```

Dictionary iteration

```python
insects = {"spiders": 8, "centipedes": 100, "bees": 6}
for name in insects:
    print(insects[name])
```

What will be the order of items?

Dictionary iteration

```python
insects = {"spiders": 8, "centipedes": 100, "bees": 6}
for name in insects:
    print(insects[name])
```

What will be the order of items?

```
8 100 6
```

Keys are iterated over in the order they are first added.

Dictionary comprehensions

General syntax:

```
{key: value for <name> in <iter exp>}
```

Example:

```
{x: x*x for x in range(3,6)}
```

Exercise: Prune

```
def prune(d, keys):
    """Return a copy of D which only contains key/value pairs
    whose keys are also in KEYS.
    >>> prune({"a": 1, "b": 2, "c": 3, "d": 4}, ["a", "b", "c"])
    {'a': 1, 'b': 2, 'c': 3}
    """
```

Exercise: Prune (Solution)

```
def prune(d, keys):
    """Return a copy of D which only contains key/value pairs
    whose keys are also in KEYS.
    >>> prune({"a": 1, "b": 2, "c": 3, "d": 4}, ["a", "b", "c"])
    {'a': 1, 'b': 2, 'c': 3}
    """
    return {k: d[k] for k in keys}
```

Exercise: Index

```
def index(keys, values, match):
    """Return a dictionary from keys k to a list of values v for which
    match(k, v) is a true value.

    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}
    """
```

Exercise: Index (solution)

```
def index(keys, values, match):
    """Return a dictionary from keys k to a list of values v for which
    match(k, v) is a true value.

    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}
    """
    return {k: [v for v in values if match(k, v)] for k in keys}
```

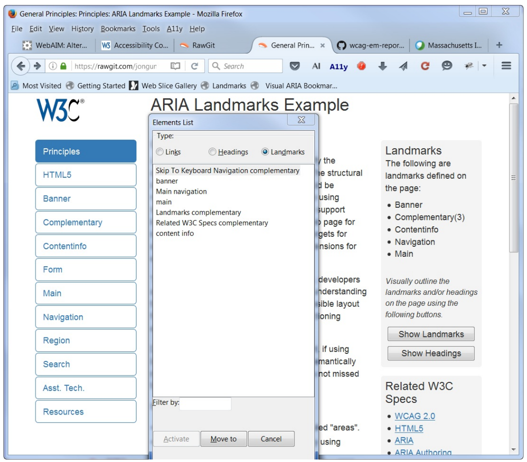
Nested data

Lists of lists	`[ [1, 2], [3, 4] ]`
Dicts of dicts	`{"name": "Brazilian Breads", "location": {"lat": 37.8, "lng": -122}}`
Dicts of lists	`{"heights": [89, 97], "ages": [6, 8]}`
Lists of dicts	`[{"title": "Ponyo", "year": 2009}, {"title": "Totoro", "year": 1993}]`

Python Project of The Day!

NVDA

NVDA (NonVisual Desktop Access): An open-source screen reader for Microsoft Windows.



Technologies used: Python, eSpeak, Sonic, etc.
(Github repository)