61A LECTURE 8 – SEQUENCES, ITERABLES

Steven Tang and Eric Tzeng
July 8, 2013

Announcements

- Homework 4 due tonight
- Homework 5 is out, due Friday
- Midterm is Thursday, 7pm
- Thanks for coming to the potluck!

What is an abstract data type (ADT)?

- We need to guarantee that constructor and selector functions together specify the right behavior.
- Behavior condition: If we construct rational number *x* from numerator *n* and denominator *d*, then `numer(x)/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, the representation is valid.

You can recognize data types by behavior, not by bits

The pair ADT

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

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.

Not true for rational numbers because of GCD

Tuple-based pair implementation

```
def pair(x, y):
    """Return a tuple-based pair."""
    return (x, y)
```

```
def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p[i]
```

Functional pair implementation

```
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

This function represents a pair

Constructor is a higher-order function

```
def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p(i)
```

Selector defers to the functional pair

Using a pair

```
>>> 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 how the pairs are implemented!

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!

The sequence abstraction

red, orange, yellow, green, blue, indigo, violet.

0 , 1 , 2 , 3 , 4 , 5 , 6 .

There isn't just one sequence type (in Python or in general)

This abstraction is a collection of behaviors:

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

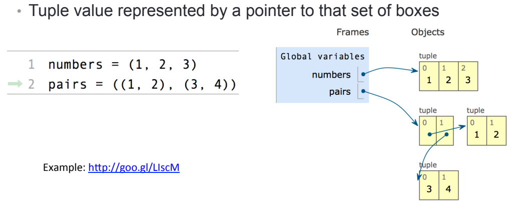
The sequence abstraction is shared among several types, including tuples.

Tuples in environment diagrams

Tuples introduce new memory locations outside of a frame

We use *box-and-pointer* notation to represent a tuple

- Tuple itself represented by a set of boxes that hold values
- Tuple value represented by a pointer to that set of boxes



Example: http://goo.gl/LlscM

Recursive Lists

Constructor:

```
def rlist(first, rest):
    """Return a recursive list from its first element and
    the rest."""
```

Selectors:

```
def first(s):
    """Return the first element of recursive list s."""

def rest(s):
    """Return the remaining elements of recursive list s."""
```

Behavior condition(s):

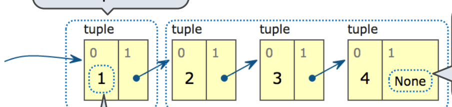
If a recursive list **s** is constructed from a first element **f** and a recursive list **r**, then

- **first(s)** returns **f**, and
- **rest(s)** returns **r**, which is a recursive list.

Implementing Recursive Lists Using Pairs

1 , 2 , 3 , 4

A recursive list is a pair



None represents the empty list

The first element of the pair is the first element of the list

The second element of the pair is the rest of the list

Example: http://goo.gl/UyekU

Implementing the Sequence Abstraction

```
def len_rlist(s):
    """Return the length of recursive list s."""
    if s == empty_rlist:
        return 0
    return 1 + len_rlist(rest(s))

def getitem_rlist(s, i):
    """Return the element at index i of recursive list s."""
    if i == 0:
        return first(s)
    return getitem_rlist(rest(s), i - 1)
```

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Break!

- We're transitioning from concepts to Python vocabulary

Python sequence abstraction

Built-in sequence types provide the following behavior

Type-specific constructor	`>>> a = (1, 2, 3)` `>>> b = tuple([4, 5, 6, 7])`
Length	`>>> len(a), len(b)` `(3, 4)` A list; more on this later
Element selection	`>>> a[1], b[-1]` `(2, 7)` Count from the end; -1 is last element
Slicing	`>>> a[1:3], b[1:1], a[:2], b[1:]` `((2, 3), (), (1, 2), (5, 6, 7))`
Membership	`>>> 2 in a, 4 in a, 4 not in b` `(True, False, False)`

Sequence iteration

Python has a special statement for iterating over the elements in a sequence

```
def count(s, value):
    total = 0
    for elem in s:
        if elem == value:
            total += 1
    return total
```

Name bound in the first frame of the current environment

For statement execution

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

1. Evaluate the header `<expression>`, which must yield an iterable value.
2. For each element in that sequence, in order:
   A. Bind `<name>` to that element in the first frame of the current environment.
   B. Execute the `<suite>`.

Demo: http://goo.gl/cWX38

Sequence unpacking in for statements

A sequence of fixed-length sequences

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))

>>> same_count = 0
```

A name for each element in a fixed-length sequence

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

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

>>> same_count
2
```

The range type

A range is a sequence of consecutive integers.*

```
..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...
```

```
range(-2, 3)
```

Length: ending value - starting value

Element selection: starting value + index

```
>>> tuple(range(-2, 3))
(-2, -1, 0, 1, 2)

>>> tuple(range(4))
(0, 1, 2, 3)
```

Tuple constructor

With a 0 starting value

* Ranges can actually represent more general integer sequences.

String literals

```
>>> '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\nclaims, Readability counts.\nRead
more: import this.'
```

Single- and double-quoted strings are equivalent

A backslash "escapes" the following character

"Line feed" character represents a new line

Strings are sequences

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

An element of a string is itself a string!

The **in** and **not in** operators match substrings

```
>>> 'here' in "Where's Waldo?"
True
```

Why? Working with strings, we care about words, not characters

Sequence arithmetic

Some Python sequences support arithmetic operations

```
>>> city = 'Berkeley'
>>> city + ', CA'
'Berkeley, CA'

>>> "Don't repeat yourself! " * 2
"Don't repeat yourself! Don't repeat yourself! "

>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> (1, 2, 3) + (4, 5, 6, 7)
(1, 2, 3, 4, 5, 6, 7)
```

Concatenate

Repeat twice

Sequences as conventional interfaces

We can apply a function to every element in a sequence

This is called *mapping* the function over the sequence

```
>>> fibs = tuple(map(fib, range(8)))
>>> fibs
(0, 1, 1, 2, 3, 5, 8, 13)
```

We can extract elements that satisfy a given condition

```
>>> even_fibs = tuple(filter(is_even, fibs))
>>> even_fibs
(0, 2, 8)
```

We can compute the sum of all elements

```
>>> sum(even_fibs)
10
```

Both **map** and **filter** produce an iterable, not a sequence

Iterables

Iterables provide access to some elements in order but do not provide length or element selection

Python-specific construct; more general than a sequence

Many built-in functions take iterables as argument

tuple	Construct a tuple containing the elements
map	Construct a map that results from applying the given function to each element
filter	Construct a filter with elements that satisfy the given condition
sum	Return the sum of the elements
min	Return the minimum of the elements
max	Return the maximum of the elements

For statements also operate on iterable values.

Generator expressions

One large expression that combines mapping and filtering to produce an iterable

(<map exp> for <name> in <iter exp> if <filter exp>)

- Evaluates to an iterable.
- <iter exp> is evaluated when the generator expression is evaluated.
- Remaining expressions are evaluated when elements are accessed.

No-filter version: **(<map exp> for <name> in <iter exp>)**

Precise evaluation rule introduced in Chapter 4.

More Functions on Iterables (Bonus)

Create an iterable of fixed-length sequences

```
>>> a, b = (1, 2, 3), (4, 5, 6, 7)
>>> for x, y in zip(a, b):
...     print(x + y)
...
5
7
9
```

Produces tuples with one element from each argument, up to length of smallest argument

The **`itertools`** module contains many useful functions for working with iterables

```
>>> from itertools import product, combinations
>>> tuple(product(a, b[:2]))
((1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5))
>>> tuple(combinations(a, 2))
((1, 2), (1, 3), (2, 3))
```