Sequences
&
Data Abstraction

Sequences

Sequences

A **sequence** is an ordered collection of values.

```
"hello world"
"abcdefghijkl"
```

strings

sequence of characters

```
[1, 2, 3, 4, 5]
[True, "hi", 0]
```

lists

sequence of values of any data type

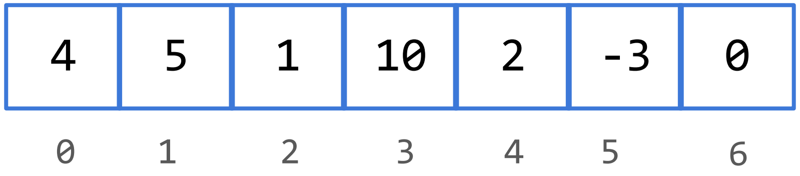
Demo

Sequence Abstraction

All sequences have finite length.

Each element in a sequence has a discrete integer index.

```
>>> [4, 5, 1, 10, 2, 3, 0]
[4, 5, 1, 10, 2, 3, 0]
```



Sequences share common behaviors based on the shared trait of having a finite length and indexed elements.

- Retrieve an element at a particular position
- Create a copy of a subsequence
- Check for membership
- Concatenate two sequences together
- ...

What can you do with sequences?

Get item: get the ***i*th** element `<seq>[i]`

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[2]
3
>>> "cs61a"[3]
'1'
```

Slice a subsequence: create a copy of the sequence from ***i*** to ***j*** `<seq>[i:j:skip]`

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[1:4]
[2, 3, 4]
>>> "lololololol"[3::2]
'ooooo'
```

Check membership: check if the value of <expr> is in <seq> `<expr> in <seq>`

```
>>> 3 in [1, 2, 3, 4, 5]
True
>>> 'z' in "socks"
False
>>> 2 + 4 in [7, 6, 5, 4, 3]
True
```

Concatenate: combine two sequences into a single sequence `<s1> + <s2>`

```
>>> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
>>> "hello "  + "world"
"hello world"
>>> [-1] + [0] + [1]
[-1, 0, 1]
```

Sequence Processing

Iterating through sequences

You can use a for statement to iterate through the elements of a sequence:

```
for <name> in <seq>:
    <body>
```

Rules for execution:

For each element in <seq>:

1) Bind it to <name>
2) Execute <body>

```
i = 0
for elem in [8, 9, 10]:
    print(i, ":", elem)
    i += 1
```

```
Output
-------
0 : 8
1 : 9
2 : 10
```

Range

The range function creates a sequence containing the values within a specified range.

```
range(<start>, <end>, <skip>)
```

Creates a range object from <start> (inclusive) to <end> (exclusive), skipping every <skip> element

This is useful for looping:

```
>>> for e in range(1, 8, 2):
...     print(e)
1
3
5
7
```

```
>>> lst = [8, 9, 10]
>>> for i in range(len(lst)):
...     print(i, ":", lst[i])
0: 8
1: 9
2: 10
```

List Comprehensions

You can create out a list out of a sequence using a list comprehension:

```
[<expr> for <name> in <seq> if <cond>]
```

```
lst = []
for <name> in <seq>:
    if <cond>:
        lst += [<expr>]
```

Rules for execution

1. Create an empty result list that will be the value of the list comprehension
2. For each element in <seq>:
   A. Bind to that element to <name>
   B. If <cond> evaluates to a true value, then add the value of <expr> to the result list

Note: binding to <name> will not overwrite local bindings

List Comprehension Examples

```
>>> [x ** 2 for x in [1, 2, 3]]
[1, 4, 9]

>>> [c + “0” for c in "cs61a"]
['c0', 's0', '60', '10', 'a0']

>>> [e for e in "skate" if e > "m"]
['s', 't']

>>> [[e, e+1] for e in [1, 2, 3]]
[[1, 2], [2, 3], [3, 4]]
```

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

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

Exact representation as 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:

Constructor → `rational(n, d)` returns a rational number x

Selectors →
- `numer(x)` returns the numerator of x
- `denom(x)` returns the denominator of x

Rational Numbers Arithmetic

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$$

Example

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx * ny}{dx * dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx * dy}$$

General Form

Rational Numbers Arithmetic Implementation

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

Constructor: `rational`

Selectors: `numer`, `denom`

```
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)
```

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx * ny}{dx * dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx * dy}$$

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

Implementation | General Form

Rational Numbers Arithmetic Implementation

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

Constructor

Selectors

```python
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)
```

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

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

These functions implement an abstract representation for rational numbers

rational(n, d) returns a rational number x
numer(x) returns the numerator of x
denom(x) returns the denominator of x

Representing Rational Numbers

```
def rational(n, d):
    """A representation of the rational number N/D."""
    return [n, d]
```

Construct a list

```
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]
```

Select item from a list

Demo

Reducing to Lowest Terms

$$\frac{3}{2} * \frac{5}{3} = \frac{5}{2}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2}$$

$$\frac{2}{5} * \frac{1}{10} = \frac{1}{2}$$

$$\frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```python
from fractions import gcd
```

Greatest common divisor

```python
def rational(n, d):
    """A representation of the rational number N/D."""
    g = gcd(n, d) # Always has the sign of d
    return [n//g, d//g]
```

Demo

Abstraction Barriers

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

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!

Dictionaries

(if time)

Demo