

# LAB 05

---

## Data Abstraction, Sequences


Mingxiao Wei

[mingxiaowei@berkeley.edu](mailto:mingxiaowei@berkeley.edu)

Feb 21, 2023

# LOGISTICS

---

- Lab 05 due Wed 02/22
- CATS 
  - Try out the game here: [cats.cs61a.org](https://cats.cs61a.org)
  - Checkpoint 1 due today 02/21
  - The whole project due Fri 02/24
  - Submit everything by Thu 02/23 for one extra point
- NO homework due this week - homework 04 due next Thu 03/02

# FROM LAST TIME 🙄

If you could hold any number of pigeons, how many would you hold and why?

0	None, I'd let them go so they could be free :D
0	1
0	1 because pigeons are scary
0	1 max bc I am lowkey afraid of them
0	I would hold one so we could bond and he can become my best friend.
0	2
0 :)	2 - with 2 hands
0, birds are creepy to me	2 (one in each hand)
0, dont want to hold any	2 because it's small even #
0 they are gross	5, because that's a lot of pigeons but not too many that you can't watch them all for if they're going to poop on you
0, i dont like birds	6 because 9
0 because I don't like pigeons (:	10 and teach them all how to steal food
0. I'm afraid of them.	15
0 because I don't like pigeons	16 because that's my favorite number
0 because they are gross	100 the more the merrier
0 they're dirty	

# LIST COMPREHENSION

---



# LIST COMPREHENSION

---

- List comprehensions are a compact and powerful way of creating new lists out of sequences.

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

- In English, this translates to:
  - Compute the expression for each element in the sequence if the condition is true for that element (or skip this check if there's no condition)
- Note:
  - `if <cond>` is optional.
  - `<expr>` and `<cond>` may refer to `<var>`, which is essentially every element in the sequence

# LIST COMPREHENSION

- List comprehensions are a compact and powerful way of creating new lists out of sequences.

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

- In Python, this translates to:

```
lst = []  
for <var> in <seq>:  
    if <cond>:  
        lst.append(<expr>)
```

- Note:
  - `if <cond>` is optional.
  - `<expr>` and `<cond>` may refer to `<var>`, which is essentially every element in the sequence

# EXAMPLES

---

```
>>> lst = [1, 2, 3, 4]
>>> [i ** 2 for i in lst if i % 2 == 1]
[1, 9]
>>> [[i, j] for i in lst for j in lst if i < j]
[[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
>>> # here [i, j] for i in lst is considered as
>>> # the expression for the second list comprehension
```

# SEQUENCES

---





# SEQUENCES - INTRO

---

- Sequences = ordered collections of values
  - Examples: lists, strings, tuples, ranges
- Supported operations:
  - `s[0]` - element selection / indexing
  - `len(s)` - length of `s`
  - `s[i:j:k]` - slicing
  - `max(s)` / `min(s)` - largest / smallest item from `s`
  - `x in s` / `x not in s` - checking whether or not `x` is in `s`

# RANGE

---

```
range(start, end, step)
```

- a sequence of integers
- can be converted to a list by calling the list constructor `list()`
- `start`: optional, inclusive, default to 0
- `end`: required, exclusive
- `step`: optional, default to 1

```
>>> list(range(4))  
[0, 1, 2, 3]  
>>> list(range(1, 6))  
[1, 2, 3, 4, 5]  
>>> list(range(2, 9, 2))  
[2, 4, 6, 8]
```

# SEQUENCES - EXAMPLES

---

```
>>> x = '61a is fun!'
```

```
>>> len(x) # space and punctuation counts!
```

```
11
```

```
>>> x[8:] # alternatively, x[-4:]
```

```
'fun!'
```

```
>>> x[::-1] # reverse the string
```

```
'!nuf si a16'
```

```
>>> '61a' in x and 'fun' in x # look for a substring in x
```

```
True
```

```
>>> '61A' not in x # case sensitive
```

```
True
```

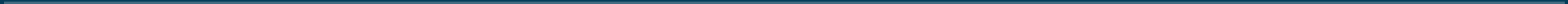
# SEQUENCES - EXAMPLES

- To iterate through a sequence, we can either
  - iterate over the items in the sequence directly, e.g.  
`for elem in seq:`
  - iterate through its indices using `range`, e.g.  
`for i in range(len(seq)):`

```
lst = ['go', 'bears', '🐻']  
for elem in lst:  
    print(elem)  
# output  
'go'  
'bears'  
'🐻'
```

```
lst = ['go', 'bears', '🐻']  
for i in range(len(lst)):  
    print(i, lst[i])  
# output  
0 'go'  
1 'bears'  
2 '🐻'
```

# DATA ABSTRACTION



# ABSTRACTION - INTRO

---

- Imagine driving a car (or if you can't drive like me, think of driving a bumper car)
- When you hit the brake, you know the car will stop (otherwise that'll be an accident...)
- Even if you don't know how a brake is implemented mechanistically, you know it'll stop the car, and you know how to hit the brake, therefore you can stop the car - life's good :)
- This is an example of abstraction - mechanistical details about the brake are abstracted away, and you don't need to know anything about the details to use the brake

# DATA ABSTRACTION - INTRO

---

- Treat code as "objects"
  - no need to know about the actual implementation (e.g., how information is stored and calculated)
  - just need to know what it does
- One can assume that the functions work as described
- A data abstraction consists of two types of functions:
  - **Constructors**: build and return the abstract data type
  - **Selectors**: retrieve information from the data type

# RATIONAL NUMBERS REVISIT

---

- Constructor: `rational(n, d)` returns a rational number  $x = \frac{n}{d}$  using some underlying representation, which we, as users, do not need to know anything of
- Selectors:
  - `numer(x)` returns the numerator of  $x$
  - `denom(x)` returns the denominator of  $x$
- Arithmetic Operations:

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

- We can manipulate rational numbers using their constructor/selectors without knowing their implementation
- Side note: the `\` is used to indicate that the expression continues on the next line



# RATIONAL NUMBERS REVISIT

---

There are many ways to implement the rational number data abstraction. Below are two examples:

```
def rational(n, d):  
    return [n, d]  
def numer(x):  
    return x[0]  
def denom(x):  
    return x[1]
```

```
def rational(n, d):  
    return {'n': n, 'd': d}  
def numer(x):  
    return x['n']  
def denom(x):  
    return x['d']
```

No matter which one we use, the rational number data abstraction has the same, correct behavior from the users' end.

As programmers, we can design the underlying implementation however we want as long as it behaves as expected

# DON'T BREAK THE ABSTRACTION BARRIER!

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

Source: [lecture 13 slides](#)

# DON'T BREAK THE ABSTRACTION BARRIER!

---

- Why?
  - Breaking the abstraction barrier is repeating other function's job
  - Once the underlying implementation changes, the code may not work anymore
- Examples of violating the abstraction barriers:
  - `mul_rational([1, 2], [3, 4])` should be `mul_rational(rational(1, 2), rational(1, 2))`
  - ```
def divide_rational(x, y):  
    return [ x[0] * y[1], x[1] * y[0] ]  
# should be:  
def divide_rational(x, y):  
    return rational(denom(x) * numer(y), \  
                   numer(x) * denom(y))
```
- Takeaway: call an existing function could do whenever you can!

# NOW IT'S LAB TIME 🤠

---

- Get started on the lab and raise your hand whenever you need help!
- Get to know your neighbors and collaborate if you'd like!
- Slides: [go.cs61a.org/mingxiao-index](https://go.cs61a.org/mingxiao-index)
- Leave any anonymous feedback here: [go.cs61a.org/mingxiao-anon](https://go.cs61a.org/mingxiao-anon)

AND REMEMBER TO GET  
CHECKED OFF! 🧺

---

[go.cs61a.org/mingxiao-att](https://go.cs61a.org/mingxiao-att)

The secret phrase is ...  
(NOT 3 dots! I'll announce it 🙊)

