CS 61A SU26

Lecture 08: Mutability and Data Abstraction

July 2, 2026

Rebecca Dang

Join pollev.com/rebeccadang831 (or scan QR code) on your phone or laptop

We'll begin at Berkeley time (10 minutes after), as per tradition!

Potpourri

5 min

- Announcements
- Autoremind
- Computing in the News

Announcements

- **Please make sure to redownload the Lecture 7 slides PDF, starter code, and solution code because there were many typos that I fixed during/after lecture.**
  - After lecture changes:
    - Add `from operator import getitem` in slide 7
    - Update slide 34 to add appropriate hints for the `decode` problem
    - Update slide 34 to add the `encode` problem which actually requires you to use `%` (you actually don't need it for `decode` since we're shifting left)
- **Quiz 1 grades are released (Ed post)**
  - We highly recommend going to OH to go over your quiz with a staff member, especially since the coding question doctests turned out to not be comprehensive
  - Don't be discouraged if you didn't get the score you wanted — there are many opportunities to improve and your grade does not define you!
- **Tomorrow Fri 7/3 is an academic and administrative holiday** (staff have the day off)

Autoremind

If you consented to the Autoremind study in the pre-semester survey:

- **Please check your email**: You should register on the Autoremind website, **even if you're in group 2** (you will automatically receive reminders with no additional setup necessary when approved for access)
- If you are interested and haven't opted in yet, email autoremind@berkeley.edu to get added by tomorrow, Fri 7/3
- Thank you for participating in research!

Computing in the News

- ‘We are screwed’: People near data centers dread heat wave pollution (POLITICO)
- Meta Is Charging a Subscription for Smart Glasses Features. Welcome to the New Era of Consumer Tech (WIRED)
- Apple's Hide My Email feature has a bug that's been exposing real email addresses, researcher claims (TechCrunch)

Review

10 min

- Lec 7 follow-up: `any` and `all` "short-circuiting"
- WWPD: List Comprehension and Lambdas
- WWPD: Identity and Integers

Lec 7 follow-up: `any` and `all` "short-circuiting"

- Short answer: Yes, they do
- According to the Python docs, they're equivalent to checking each value in a for loop and exiting once you see the first truthy or falsy value (for `any` and `all`, respectively)

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

List Comprehension and Lambdas: What would Python display?

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Identity and Integers: What would Python display?

Mutability

20 min

- Definitions
- Tuples
- Name change vs. mutation
- WWPD: Mutable default arguments
- Dictionaries
- WWPD: Dictionary comprehension

Definitions

An object is **mutable** if it can be changed.

An object is **immutable** if it cannot be changed.

Why immutability? Sometimes we don't want things to change (ex: read-only data).

Tuples (1 of 3)

A **tuple** is an immutable sequence (basically, an unchangeable list).

Pronunciation: "too-ple" or "tuh-ple," it doesn't really matter

Tuples (2 of 3)

```
>>> tup = (1, 2, 3)
>>> tup = 1, 2, 3  # parentheses optional, but recommended
>>> tup
(1, 2, 3)
>>> a, b, c = tup  # tuple unpacking
>>> a
1
>>> b
2
>>> c
3
```

Tuples (3 of 3)

```
>>> tup = (1, 2, 3)
>>> tup[1]
2
>>> tup[0] = 'go bears'
Error
```

Name change vs. mutation

Name changes are **not** the same as mutation!

The value of an expression can change because of changes in names or objects.

Name change

```
>>> x = 2
>>> x + x
4
>>> x = 4
>>> x + x
8
```

Mutation

```
>>> x = [1, 2]
>>> x + x
[1, 2, 1, 2]
>>> x.append(3)
>>> x + x
[1, 2, 3, 1, 2, 3]
```

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Mutable default arguments: What would Python display?

Dictionaries (1 of 5)

A **dictionary** is an unordered* container made of key-value pairs (entries).

Also called maps or hashmaps in other languages.

*As of Python 3.7, they are guaranteed insertion-ordered (Python Docs)

Why dictionaries? A lot of data is naturally represented using mappings from one set of values to another. Ex:

- Names to email addresses
- Students to letter grades
- Names to physical addresses
- IP addresses to domain names (URLs)

Dictionaries (2 of 5)

```
>>> roman_numerals = {'I': 1, 'V': 5, 'X': 10}
>>> roman_numerals['I']
1
>>> roman_numerals['L'] = 50  # add a new entry
>>> roman_numerals
{'I': 1, 'V': 5, 'X': 10, 'L': 50}
>>> roman_numerals['V'] = 'five'  # modify an existing entry
>>> roman_numerals
{'I': 1, 'V': 'five', 'X': 10, 'L': 50}
>>> del roman_numerals['V']  # delete an entry
>>> roman_numerals
{'I': 1, 'X': 10, 'L': 50}
```

Dictionaries (3 of 5)

```
>>> {1: 'one', 2: 'two', 1: 'ONE'}  # no duplicate keys
{1: 'ONE', 2: 'two'}
>>> {[1, 2]: 3}  # keys must be hashable
Error
>>> dct = {(1, 2): 3}  # this is fine
>>> dct[(1, 2)]
3
>>> dct = {(1, [2, 3]): 4}  # this is not
Error
```

Dictionaries (4 of 5)

```
>>> pokemon = {'charizard': 'fire', 'squirtle': 'water', 'pikachu': 'electric'}
>>> for name in pokemon.keys():
...     print(name)
...
charizard
squirtle
pikachu
>>> for pokemon_type in pokemon.values():
...     print(pokemon_type)
...
fire
water
electric
```

Dictionaries (5 of 5)

```
>>> pokemon = {'charizard': 'fire', 'squirtle': 'water', 'pikachu': 'electric'}
>>> for name, pokemon_type in pokemon.items():
...         print(f"{name}: {pokemon_type}")
...
charizard: fire
squirtle: water
pikachu: electric
>>> pokemon['bulbasaur']
Error
>>> pokemon.get('bulbasaur')  # default: returns None if it doesn't exist
>>> pokemon.get('bulbasaur', 'grass')
'grass'
>>> pokemon.pop('squirtle')
'water'
>>> pokemon
{'charizard': 'fire', 'pikachu': 'electric'}
```

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Dictionary comprehension: What would Python display?

Data Abstraction

20 min

- What is abstraction?
- Abstraction in Python
- What are abstract data types?
- Mutable Types for "Proto" ADTs
- Abstract Data Types (ADTs)
- Abstraction Barrier
- Bank Account
- Rational Number

What is abstraction?

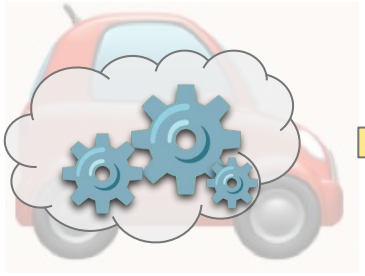
Know **what** it does, not **how** it works/implementation

→ we can still use it without knowing how it's implemented!

Example:

Pressing on the accelerator speeds up the car

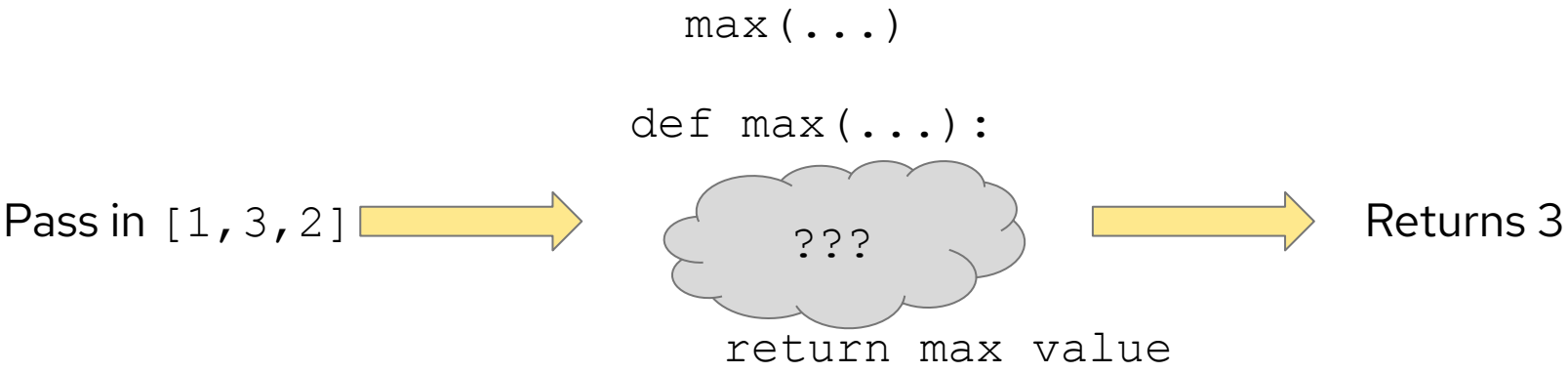
Press accelerator



We don't know how the car works internally when we press this

But we know that we end up speeding up!

Abstraction in Python



We don't know how `max` is implemented, but we know that we can use it to give us the largest element in an iterable.

Mutable Types for "Proto" ADTs (1 of 2)

How can we represent objects/data in code?

One idea: Use **functions** and **mutable types**

Mutable Types for "Proto" ADTs (2 of 2)

```
def create_account(balance):
    account = [balance]
    def withdraw(amount):
        if amount > account[0]:
            return 'Insufficient funds'
        account[0] -= amount
        return account[0]
    return withdraw


>>> withdraw = create_account(100)
>>> withdraw(25)
75
>>> withdraw(5)
70
```

Abstract Data Types (ADTs)

Problems with previous approach:

- Annoying to have multiple attributes
- If we change the underlying implementation of a bank account (e.g. from a list to a dictionary), the functions that use the account (e.g. `withdraw`) may break

Instead, let's separate concerns: An ADT is made up of:

1. A **constructor** function, which creates an instance of the ADT
2. One or more **selector** functions, which each return an attribute of the ADT

ADTs are an example of **functional data abstraction!**

(If you are familiar with object-oriented programming, you might think this is weird — but remember not all programming languages have OOP, such as Scheme!)

Abstraction Barrier

- **IMPORTANT:** Everything outside the ADT is **outside of the abstraction barrier.**
- Do **NOT** make assumptions about the underlying implementation of the ADT when you are outside the barrier
  - This is called **violating the abstraction barrier**
- *"Code that violates abstraction barriers should burn"* -Prof. John DeNero

ADTs: Bank Account

```
def create_account(owner, balance):
    return [owner, balance]

def get_owner(account):
    return account[0]

def get_balance(account):
    return account[1]

# ------- abstraction barrier -------
def withdraw(account, amount):
    balance = get_balance(account)
    if amount > balance:
        return 'Insufficient funds'
    return create_account(get_owner(account), balance - amount)
```

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Which line(s) contain abstraction barrier violations? Select all that apply.

ADTs: Rational Number (1 of 5)

Recall that a **rational number** can be represented as a **numerator** and **denominator** of a fraction.

(We do have floats in Python, but they do not have infinite precision.)

$$\frac{1}{2} \qquad \frac{22}{7} \qquad \frac{42}{100}$$

ADTs: Rational Number (2 of 5)

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

def numer(x):
    return x['numer']

def denom(x):
    return x['denom']
```

ADTs: Rational Number (3 of 5)

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

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

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

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

ADTs: Rational Number (4 of 5)

```
>>> half = rational(1, 2)
>>> print_rational(half)
1 / 2
>>> third = rational(1, 3)
>>> print_rational(mul_rationals(half, third))
1 / 6
>>> print_rational(add_rationals(third, third))
6 / 9
```

ADTs: Rational Number (5 of 5)

```
from math import gcd


def rational(n, d):
    g = gcd(n, d)
    return {'numer': n // g, 'denom': d // g}
```

Takeaway:
Because we did not violate the abstraction barrier in the previous slide, we could change our implementation of `rational` to whatever we want (as long as the behavior stays constant) and the code on the previous slide will still work!

5 min break

Practice

(reminder to self to use high contrast theme)

20 min

- Film Appearances
- Coordinate ADT

Practice: Film Appearances

- Implement `film_appearances`
- Download starter code `08.py` from the course website under Lecture 8
- Run doctests with `python3 -m doctest 08.py`
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?

Practice: Coordinate ADT

- Implement the Coordinate ADT, which represents a coordinate `(x, y)` pair, and a function `distance` that returns the distance between two Coordinates.
  - You may use the imported `sqrt` function.
  - **Challenge**: Implement it using HOFs rather than any built-in data structures (e.g. lists/tuples/dictionaries)
- Download starter code `08.py` from the course website under Lecture 8
- Run doctests with `python3 -m doctest 08.py`
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?