

# Computational Structures in Data Science

---

## OOP Part 3 Midterm Review

Week 5, Summer 2024. 7/16 (Tues)

Lecture 16



# Announcements

- Midterm this week!
- Project01 ("Maps") due 7/18 (Thurs!)
- Project02 ("Ants") released on 7/17 (Wed)

# Announcements: Midterm

- Midterm this week!
- **Important:** please carefully read this Ed post: [Midterm Megathread](#). It is your responsibility to read and understand the entirety of this post, particularly the [Midterm Logistics](#) and [Online Midterm Logistics](#) posts.
- Failure to do so can, at worst, lead to issues like academic integrity violations or missed exams, and can lead to your midterm score being cancelled!
- "Primary" Midterm exam time: Wednesday July 17<sup>th</sup> 2024, 3 PM – 5 PM PST
- **(Alternate Exam Times, DSP):** on Friday (7/12) we sent an e-mail to all students that needed an alternate exam time, and assigned them their midterm time slot.
  - If you didn't receive an e-mail, please let us know ASAP by asking in Ed or e-mailing us at [cs88@berkeley.edu](mailto:cs88@berkeley.edu)

# Computational Structures in Data Science

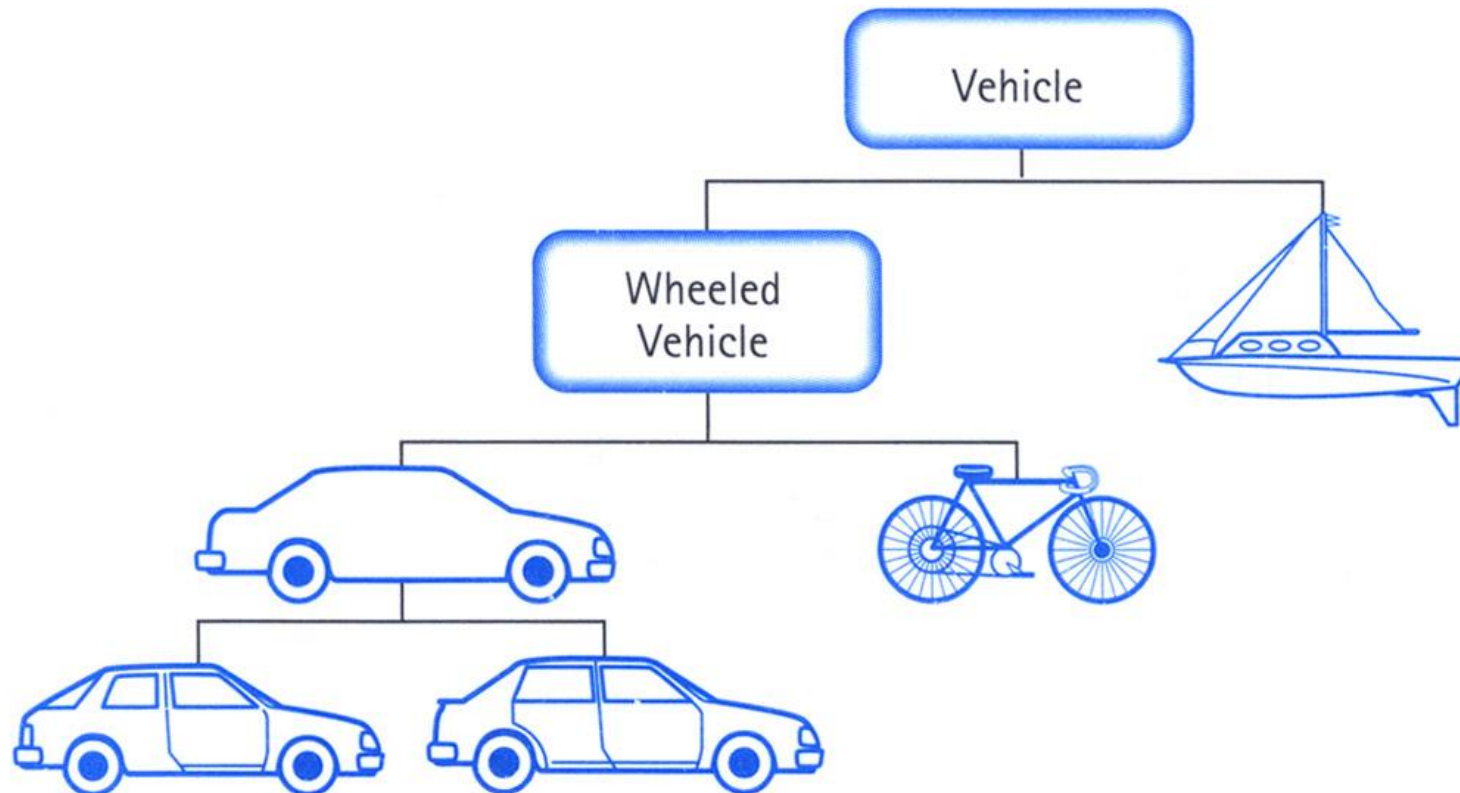
---

## Object-Oriented Programming: Inheritance Review



# Class Inheritance

- Classes can inherit methods and attributes from parent classes but extend into their own class.
- “is a” relationship



# Example

CheckingAccount  
inherits from the  
BaseAccount class

BaseAccount is the  
“parent class” of the  
CheckingAccount class.

(jargon)  
CheckingAccount  
“extends” BaseAccount



```
class BaseAccount:
    def __init__(self, name, initial_deposit):
        # Initialize the instance attributes
        self._name = name
        self._acct_no = Account._account_number_seed
        Account._account_number_seed += 1
        self._balance = initial_deposit

class CheckingAccount(BaseAccount):
    def __init__(self, name, initial_deposit):
        # Use superclass initializer
        BaseAccount.__init__(self, name, initial_deposit)
        # Alternatively (recommended):
        # super().__init__(name, initial_deposit)
        # Additional initialization
        self._type = "Checking"
```

# Accessing the Parent Class

- `super()` gives us access to methods in the parent or "superclass"
  - Can be called anywhere in our class
  - Handles passing `self` to the method
  - Handles looking up an attribute on a parent class, too.
- We can directly call `ParentClass.method(self, ...)`
  - This is not quite as flexible if our class structure changes.
- In general, prefer using `super()`!
- Outside of C88C, things can get complex...
  - <https://docs.python.org/3/library/functions.html#super>

# super() and Multiple Parent Classes

- In general, `super()` is "smart"
  - It tries to find the most correct parent class
  - Super will search through classes with multiple parent classes, or a long hierarchy of classes
- Hardcoding class name (eg ``ParentClass``) is less flexible, but very specific.
  - Use it if you know you ***always*** want the same class to be used.



# When Should You Use Inheritance?

Use inheritance to *refine* the behavior of a parent.

For example, our BaseAccount allows us to overdraft our account.

We might want to protect against this:

```
class CheckingAccount(BaseAccount):  
    # (...omitted...)  
    def withdraw(self, amount):  
        if self.account_balance() - amount < 0:  
            return "ERROR: You are not allowed to overdraft a  
CheckingAccount."  
        return super().withdraw(amount)
```

# Inheritance & Class Attributes - Warning

- Warning: when referencing class variables, be careful about hardcoding class names when combining with inheritance:

```
class BaseClass:
    my_global_var = 42
    def some_method(self):
        return BaseClass.my_global_var
```

```
class ChildClass(BaseClass):
    my_global_var = 9000 # override
```

```
>>> base_class = BaseClass()
>>> child_class = ChildClass()
>>> base_class.some_method()
```

```
# FILL ME IN
```

```
42
```

```
>>> child_class.some_method()
```

```
# FILL ME IN
```

```
42
```



Surprise! ChildClass is still using BaseClass.my\_global\_var.

# Inheritance & Class Attributes - Warning

How to get ChildClass's class var override to "do the right thing"?

```
class BaseClass:
    my_global_var = 42

    def some_method(self):
        return BaseClass.my_global_var

    def some_method_v2(self):
        # "works", but be careful, this can
        # confuse global vars vs instance vars
        return self.my_global_var

    def some_method_v3(self):
        # IMO a better version, dynamically
        # get class rather than hardcoding
        return type(self).my_global_var

class ChildClass(BaseClass):
    my_global_var = 9000 # override
```

```
>>> child_class = ChildClass()
>>> child_class.some_method_v2()
9000
>>> child_class.some_method_v3()
9000

# Tricky: create an INSTANCE VAR
>>> child_class.my_global_var = 'hi'
>>> child_class.some_method_v2()
'hi'
>>> child_class.some_method_v3()
9000
```

Main takeaways:

- (1) Hardcoding class names when accessing class vars can lead to issues for inheritance
- (2) Instance vars can "shadow" global vars if you're not careful

# Computational Structures in Data Science

---

## Midterm Review



# Announcements & Policies

- Midterm:
  - 2 hours, 120 Minutes
  - 5 **Handwritten** Cheat sheets – More than ~3 is counter-productive
  - 1 [CS88 Provided Reference Sheet](#)

You are not your grades!  
Do your best!

"Perseverance is the hard work you do after you get tired of doing the hard work you already did." —Newt Gingrich (Former Speaker of the United States House of Representatives)

"Continuous effort—not strength or intelligence—is the key to unlocking our potential."  
—Winston Churchill (Former Prime Minister of the United Kingdom)

"Motivation will almost always beat mere talent." —Norman Ralph Augustine (Former United States Under Secretary of the Army)

"[being bad] at something is the first step towards being sorta good at something." —  
Jake the Dog (Adventure Time)



"Be like Obi-Wan!" — Eric Kim

# My Advice

- Don't rush!
  - Slow is fast and fast is slow
  - **BREATHE!**
- Skim the exam first
  - It's ok to do questions out of order!
  - Get the stuff you're good without out of the way
  - BUT don't spend too much time planning the exam.
- Read through the question once
  - What's it asking you to do at a high level?
  - What do the doctests suggest?
  - What techniques should you be using?
- Use the scratch space! (writing/drawing on blank paper is very helpful)

(especially for Environment  
Diagrams)

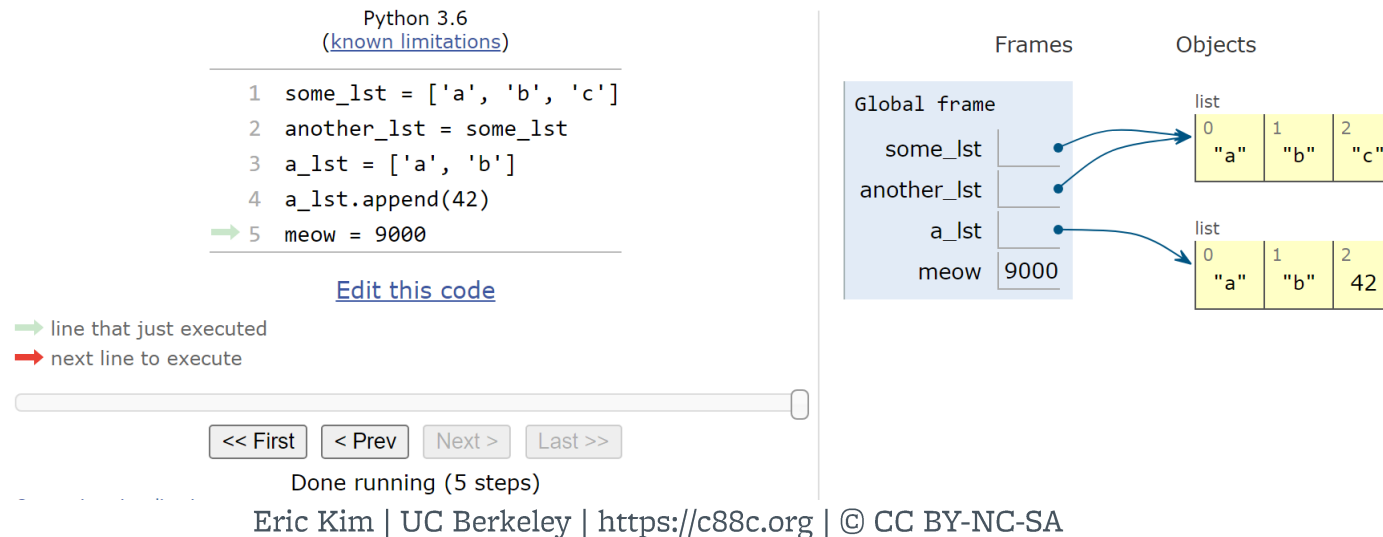


# Midterm Topics

- Everything Through OOP w/ Inheritance
- Functions
- Higher Order Functions
  - Functions as arguments
  - Functions as return values
- Environment Diagrams
- Lists, Dictionaries
- List Comprehensions, Dictionary Comprehensions
- Abstract Data Types
- Recursion
- Object-Oriented Programming, Inheritance

# Some specific advice

- Environment Diagrams, "What Would Python Print" (WWPP)
- My advice: draw it out on your blank scratch paper with paper/pen!
- Especially helpful for list/dict mutation "WWPP" questions
- You won't have Python Tutor for the exams, but you should be able to reproduce Python Tutor's visualizations "from scratch".
- Recall: multiple variables can point to the same compound object (eg list, dict, object)




# Some specific advice

- Recursion
- "Put on your recursion hats" and think about (1) Base cases, (2) Recursive cases.
- Given a problem  $X$ , how to solve it by solving a smaller problem(s)?
- (base case) What is the "smallest" problem I can solve directly/trivially?

# Some specific advice


- "Fill in the blank" coding
- If you do the practice midterms, many coding questions are of the "fill in the blank" variety.
- This requires a slightly different skillset than writing your own code "from scratch". Instead, you must become good at the following steps:
  - (1) Read existing (incomplete) code
  - (2) Determine what the code is trying to do, and
  - (3) Fill in the blanks to achieve the desired implementation.
- Tip: Resist the inner thought "I wouldn't have done it this way", and instead follow how the exam code is doing things.



Note: course teaching staff (TAs/Tutors/Instructors) become very good at this after grading student code hundreds of times :P

# Some specific advice


- "Fill in the blank" coding
- If you do the practice midterms, many coding questions are of the "fill in the blank" variety.
- This requires a slightly different skillset than writing your own code "from scratch". Instead, you must become good at the following steps:
  - (1) Read existing (incomplete) code
  - (2) Determine what the code is trying to do, and
  - (3) Fill in the blanks to achieve the desired implementation.
- Tip: Resist the inner thought "I wouldn't have done it this way", and instead follow how the exam code is doing things.



Note: course teaching staff (TAs/Tutors/Instructors) become very good at this after grading student code hundreds of times :P

# Some specific advice

- "Fill in the blank" coding
- If you do the practice midterms, many coding questions are of the "fill in the blank" variety.
- This requires a slightly different skillset than writing your own code "from scratch". Instead, you must become good at the following steps:
  - (1) Read existing (incomplete) code
  - (2) Determine what the code is trying to do, and
  - (3) Fill in the blanks to achieve the desired implementation.
- Tip: Resist the inner thought "I wouldn't have done it this way", and instead follow how the exam code is doing things.



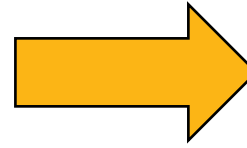
Note: course teaching staff (TAs/Tutors/Instructors) become very good at this after grading student code hundreds of times :P

# ADTs

- Tip: when working with ADT's in this class:

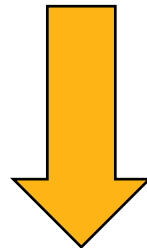
Is this  
code/function part  
of the ADT?

YES



Function CAN assume internal  
implementation details.  
Can also use ADT constructors and  
selectors.

Tip: synonyms for this  
are: "fn() is a **user** of  
the ADT", "fn() is an  
**operation**", "given an  
**ADT**, implement fn()"



NO

Tip: synonyms for this are:  
"fn() is **part** of the ADT".  
Constructors, selectors are  
part of the ADT.

Function CAN'T assume internal  
implementation details. Must use  
ADT's constructors/selectors  
("public API/interface")

# Computational Structures in Data Science

---

Some Practice Questions





# SP22 Q7

## 7. (5.0 points) Closet Overhaul

You've designed a closet abstract data type to help you organize your wardrobe.

A closet contains two things:

- **owner**: the name of the closet owner represented as a string
- **clothes**: the collection of clothes in the closet represented as a dictionary, where the key is the clothing item name and the value is the number of times the clothing item has been worn.

The `make_closet` constructor takes in **owner** (a string) and **clothes** (a list of strings representing clothing items) and returns a closet ADT.

Given this, you've implemented the abstract data type as follows:

```
def make_closet(owner, clothes):  
    """ Create and returns a new closet. """  
    clothes_dict = {}  
    for item in clothes:  
        clothes_dict[item] = 0  
    return (owner, clothes_dict)  
  
def get_owner(closet):  
    """ Returns the owner of the closet """  
    return closet[0]  
  
def get_clothes(closet):  
    """ Returns a dictionary of the clothes in the closet """  
    return closet[1]
```

Given the closet ADT, implement the functions `wear_clothes` and `favorite_clothing_item`. You may not need all the lines provided, and you may need to change the indentation for some lines.

**Question:** why can we assume that ``clothes`` is a dict, and do things like ``clothes[c] = 0``? Isn't this an abstraction violation?

**Interesting Exercise:** create a "Clothing" ADT, then rewrite both the Closet ADT and ``wear_clothes()`` to use the Clothing ADT.

- (a) (3.0 pt) Implement `wear_clothes`, which takes a closet `closet` and a list of clothing items `clothes_worn`, and increments the number of times each item is worn by 1. If the clothing item specified does not already exist in the closet, add it to the closet.

```
def wear_clothes(closet, clothes_worn):  
    """ Updates the number of times each clothing item is worn.  
    >>> adam_closet = make_closet("adam", ["polo", "tie", "shorts"])  
    >>> wear_clothes(adam_closet, ["shorts", "tie", "shorts"])  
    >>> get_clothes(adam_closet)  
    {'polo': 0, 'tie': 1, 'shorts': 2}  
    >>> wear_clothes(adam_closet, ["polo", "scarf"])  
    >>> get_clothes(adam_closet)  
    {'polo': 1, 'tie': 1, 'shorts': 2, 'scarf': 1}  
    """
```

**# FILL ME IN**

```
def wear_clothes(closet, clothes_worn):  
    clothes = get_clothes(closet)  
    for c in clothes_worn:  
        if c not in clothes:  
            clothes[c] = 0  
            clothes[c] += 1
```

**Answer:** the only ADT in this problem is the Closet ADT. There is no "clothes" ADT, thus it's safe for us to treat "clothes" as a dict. BUT: if there was a "clothes" ADT, then we'd have to use those to avoid abstraction violations.

# SP22 Q7

## 7. (5.0 points) Closet Overhaul

You've designed a closet abstract data type to help you organize your wardrobe.

A closet contains two things:

- **owner**: the name of the closet owner represented as a string
- **clothes**: the collection of clothes in the closet represented as a dictionary, where the key is the clothing item name and the value is the number of times the clothing item has been worn.

The `make_closet` constructor takes in **owner** (a string) and **clothes** (a **list** of strings representing clothing items) and returns a closet ADT.

Given this, you've implemented the abstract data type as follows:

```
def make_closet(owner, clothes):  
    """ Create and returns a new closet. """  
    clothes_dict = {}  
    for item in clothes:  
        clothes_dict[item] = 0  
    return (owner, clothes_dict)  
  
def get_owner(closet):  
    """ Returns the owner of the closet """  
    return closet[0]  
  
def get_clothes(closet):  
    """ Returns a dictionary of the clothes in the closet """  
    return closet[1]
```

Given the closet ADT, implement the functions `wear_clothes` and `favorite_clothing_item`. You may not need all the lines provided, and you may need to change the indentation for some lines.

- (b) (4.0 pt) Implement `favorite_clothing_item`, which takes in a closet `closet` and returns the name of the most frequently worn clothing item. Assume there are no ties.

```
def favorite_clothing_item(closet):  
    """ Finds the most frequently worn clothing item in a closet  
    >>> adam_closet = make_closet("adam", ["polo", "tie", "shorts"])  
    >>> wear_clothes(adam_closet, ["shorts", "tie", "shorts"])  
    >>> favorite_clothing_item(adam_closet)  
    'shorts'  
    >>> wear_clothes(adam_closet, ["tie", "polo", "polo", "scarf", "polo"])  
    >>> favorite_clothing_item(adam_closet)  
    'polo'  
    """  
    return max(_____, key = _____)
```

# FILL ME IN

```
def favorite_clothing_item(closet):  
    return max(  
        get_clothes(closet),  
        key = lambda item: get_clothes(closet)[item]  
    )
```

Returns the number of times clothing  
`item` has been worn (integer)

Tip: `max(a_dict)` does `max(a_dict.keys())`. Thus, the above could also be expressed as this to make this more clear:

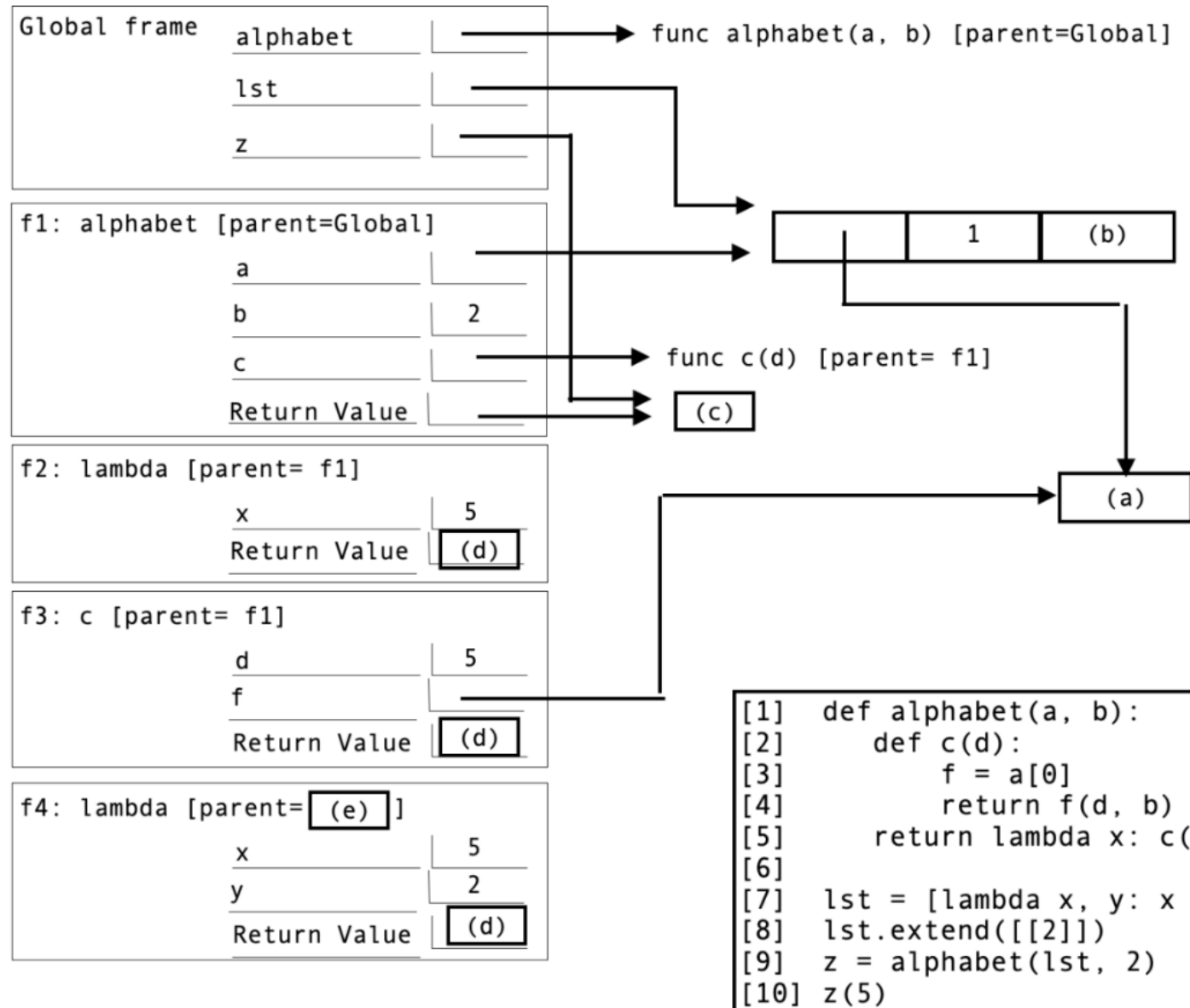
`return max(get_clothes(closet).keys(), key=<same lambda>)`

# SP24 Q2

**Demo:** how I approach  
env diagram questions  
like this

## 2. (5.0 points) Alphabet Galore!

Fill in the blanks to complete the environment diagram. All the code used is in the box to the right, and the code runs to completion.



# SP24 Q7 (sol)

```
def analyze_portfolio(portfolio, prices):
```

```
    """
    >>> prices = { "AAPL": 150, "TSLA": 100, "GOOG": 125 }
    >>> portfolio = [
        {"stock": "AAPL", "quantity": 10, "buy_price": 90},
        {"stock": "TSLA", "quantity": 5, "buy_price": 50},
        {"stock": "GOOG", "quantity": 2, "buy_price": 120},
        {"stock": "AAPL", "quantity": 5, "buy_price": 100}, # 2nd AAPL stock
        {"stock": "TSLA", "quantity": 2, "buy_price": 100}
    ]
```

```
    >>> analyze_portfolio(portfolio, prices)
    {'AAPL': 2250, 'TSLA': 700, 'GOOG': 250}
    """
```

```
    company_total = {}
    for stock_dict in portfolio:
        stock_name = stock_dict["stock"]
        current_price = ____ (a) ____
        if ____ (b) ____ in company_total:
            company_total[stock_name] += ____ (c) ____
        else:
            company_total[stock_name] = ____ (c) ____
```

```
    return company_total
```

We're iterating over every stock\_dict in portfolio, thus in each iteration I need to add something in `stock\_dict` into `company\_total` (our return value)

This is a pattern we've seen before, where if a key doesn't already exist in a dict, we first add it to the dict (`a\_dict[key] = val`), but if it already exists we can directly add to it (`a\_dict[key] += val`)

Thus, I bet blank (b) is `if stock\_name in company\_total`, and (c) is  $\text{buy\_price} * \text{quantity}$ .

We're returning `company\_total`, so I know that this must be a dict mapping `str stock\_name` to `int total\_value`

"Fill in the blank" coding! My first instinct:

Read doctests FIRST, understand the inputs/outputs and desired behavior. **If I don't understand what the function is supposed to do, I can't code it up!**

Next, I look at the provided code, and try to understand what the code is doing at a high-level/structural level.

# SP24 Q7 (sol)

```
def analyze_portfolio(portfolio, prices):
```

```
    """
    >>> prices = { "AAPL": 150, "TSLA": 100, "GOOG": 125 }
    >>> portfolio = [
        {"stock": "AAPL", "quantity": 10, "buy_price": 90},
        {"stock": "TSLA", "quantity": 5, "buy_price": 50},
        {"stock": "GOOG", "quantity": 2, "buy_price": 120},
        {"stock": "AAPL", "quantity": 5, "buy_price": 100}, # 2nd AAPL stock
        {"stock": "TSLA", "quantity": 2, "buy_price": 100}
    ]
```

```
    >>> analyze_portfolio(portfolio, prices)
    {'AAPL': 2250, 'TSLA': 700, 'GOOG': 250}
    """
    company_total = {}
    for stock_dict in portfolio:
        stock_name = stock_dict["stock"]
        current_price = ____ (a) ____
        if ____ (b) ____ in company_total:
            company_total[stock_name] += ____ (c) ____
        else:
            company_total[stock_name] = ____ (c) ____

    return company_total
```

Finally, to fill in blank (a), I re-read the docstring + problem spec to see how to get the current price of a stock, and find that I can get this info from `prices` -> blank (a) is: `prices[stock_name]`

```
def analyze_portfolio(portfolio, prices):
    company_total = {}
    for stock_dict in portfolio:
        stock_name = stock_dict["stock"]
        current_price = ____ (a) ____
        if stock_name in company_total:
            company_total[stock_name] +=
stock_dict["quantity"] * current_price
        else:
            company_total[stock_name] =
stock_dict["quantity"] * current_price
    return company_total
```