CSC 148

Week 3

Sadia ‘Rain’ Sharmin

Classes begin 10 minutes after the hour

Week 3

Inheritance

SUBTOPICS:

- Introduction to Inheritance
- Abstract Classes
- Interface vs. Implementation

Announcements

Notetaker needed

Just a reminder, we still need a couple of notetakers for this section!

Any volunteers?

More information here: https://q.utoronto.ca/courses/249810/discussion_topics/1565311

Announcements

New to-do list

We added a new weekly to-do list page on Quercus:

https://q.utoronto.ca/courses/249810/pages/weekly-to-dos

Use it to make sure you stay on track with all the things going on in this course. :)

Announcements

Assignment 0 is out!

More information about this is on Quercus:

https://q.utoronto.ca/courses/249810/pages/assignment-0

Also, note the FAQ on Piazza (REQUIRED READING): https://piazza.com/class/ky50y49v8002n5?cid=189 – look through this anytime you sit down to work on A0!

Have you started A0? *Even just skimming the handout/starter code counts as getting started.*

Inheritance

Inheritance

Scenario: Similar Classes

Say we have a SalariedEmployee class and want a new kind of employee: HourlyEmployee

Specs for HourlyEmployee would be very similar!

- Same attributes: id_, name
- Same methods: get_monthly_payment, pay
- Slight differences: salary vs. hourly wage + hours worked

Implementation ideas ... ?

Inheritance

We could try ...

Copy-paste-modify SalariedEmployee => HourlyEmployee

... that's a lot of duplicate code though!

Inheritance

We could try ...

Copy-paste-modify SalariedEmployee => HourlyEmployee

... that's a lot of duplicate code though!

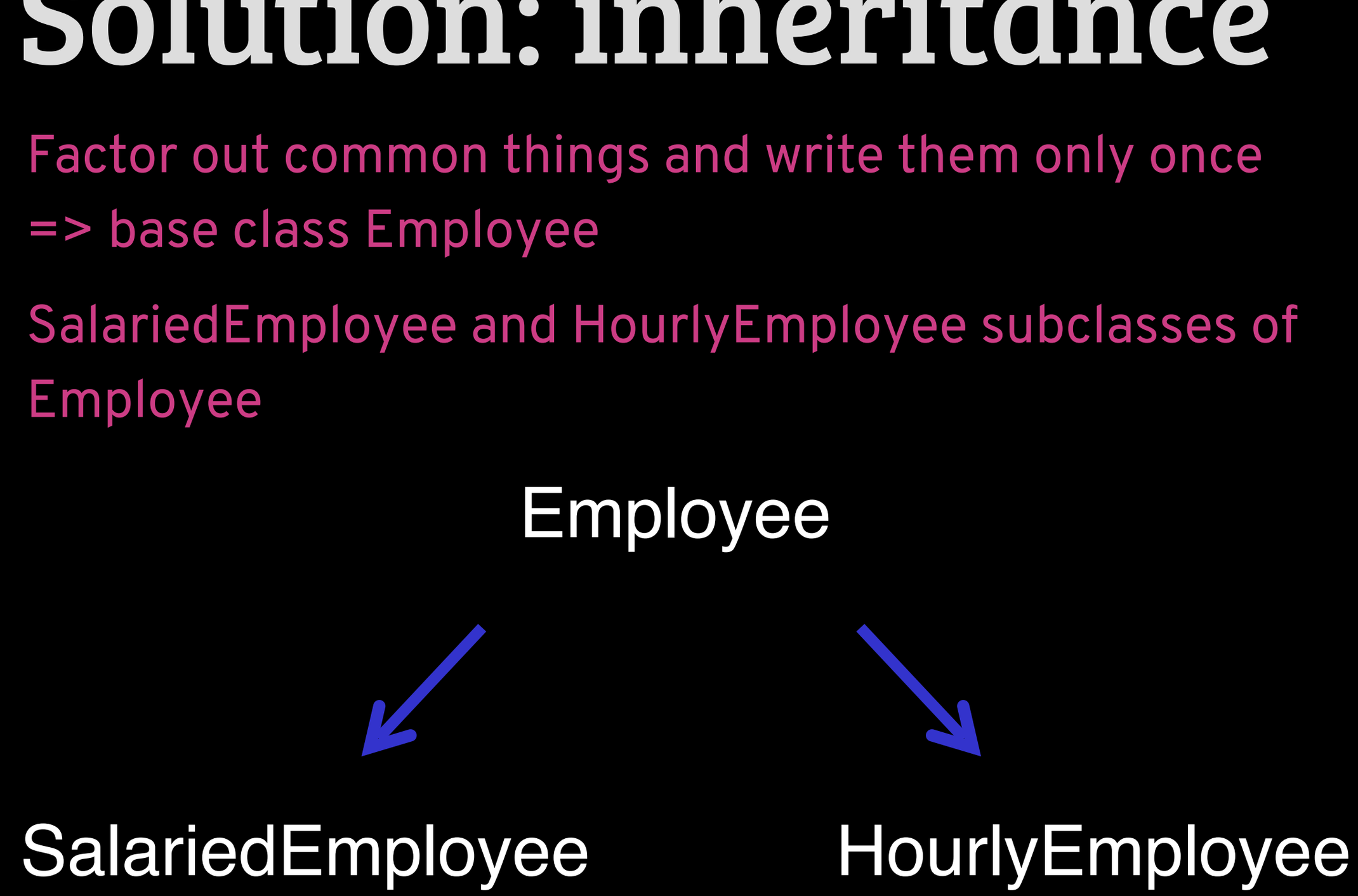
- What if there is a bug in the original class?
- What if you need many classes with small differences?

Inheritance

Solution: inheritance

Factor out common things and write them only once
=> base class Employee

SalariedEmployee and HourlyEmployee subclasses of Employee



Inheritance

Solution: inheritance

Inheritance allows a new class to specialize an existing class by specifying only what is different between them

The class that inherits is called a **subclass**, and the class that is inherited from is its **superclass**

Based on a IS-A relationship (i.e. Class A should inherit from Class B if and only if Class A IS A type of Class B e.g. SalariedEmployee is a type of Employee)

Inheritance

“Is a” vs. “Has a”

Don’t forget about composition (for HAS-A relationships)!

Inheritance

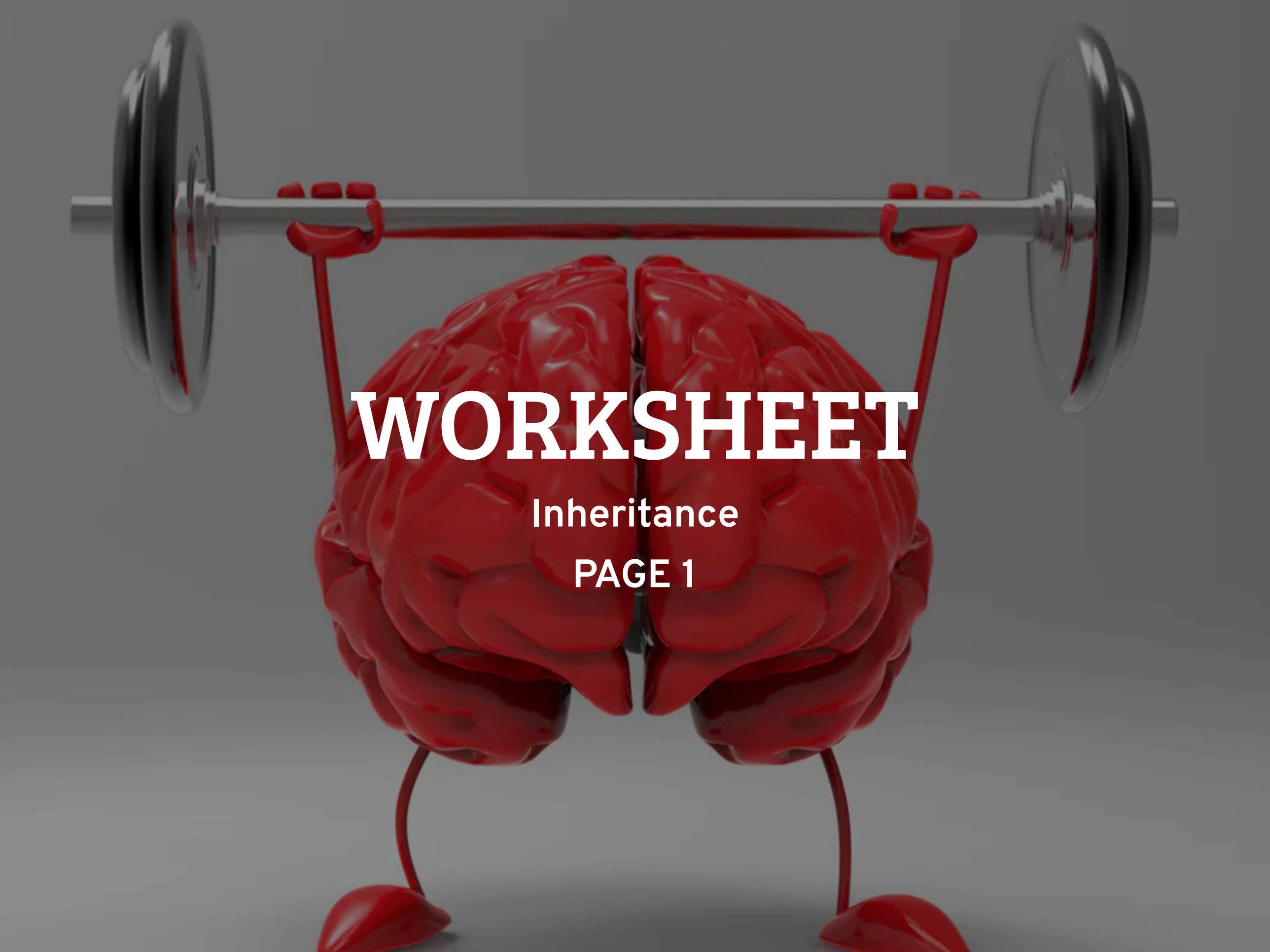
“Is a” vs. “Has a”

Inheritance is **not** always appropriate to describe the logical relationship between the entities you want to model

- Same goes for composition...

When should you use composition and when inheritance?

- Think about the relationships between objects!
- Inheritance: **"is a"** relationship
- Composition: **"has a"** relationship



WORKSHEET

Inheritance

PAGE 1

Abstract Classes

Interfaces

An abstract class is first and foremost the explicit representation of an interface in a Python program.

Remember - interface vs implementation:

Abstract

Classes

Abstract

Abstract methods

An abstract class (as with all superclasses) also enables the sharing of code through **method inheritance**

Most methods will be left up to the subclass to implement in that context

```
raise NotImplementedError
```

Some methods can be implemented in an abstract class, if behaviour will be identical in subclasses anyway

Classes

Abstract

Why do we need them?

You should never make an instance of an abstract class!

So why have it defined at all?

Because it lays out the structure of all its subclasses. Defines a *shared interface.*

Interface vs. Implementation

Why separate the two?

Advantages of separating the two, for example, with a watch:

Wearer: don't need to understand the mechanism in order to use the watch.

Maker: can change the mechanism and everyone still knows how to use the watch.

Interface

Implementation

Interface/Implementation

Why separate the two?

Advantages of separating the two for a function:

- Client: don't need to understand the body in order to use the function.
- Implementer: can change the implementation and all client code still works.

Interface

the function header and docstring

```
def add_vehicle(self, vehicle_type: str, id_: str, fuel: int) -> None:
    """Add a new vehicle with the given type, id_, and fuel to the system.

    Do nothing if there is already a vehicle with the given id.

    Preconditions:
      - <vehicle_type> is one of 'Car', 'Helicopter', or
        'UnreliableMagicCarpet'.
      - fuel >= 0
    """
```

Implementation

the function body

```
# Check to make sure the identifier isn't already used.
if id_ not in self._vehicles:
    if vehicle_type == 'Car':
        self._vehicles[id_] = Car(fuel)
    elif vehicle_type == 'Helicopter':
        self._vehicles[id_] = Helicopter(fuel)
    elif vehicle_type == 'UnreliableMagicCarpet':
        self._vehicles[id_] = UnreliableMagicCarpet(fuel)
```

Interface/Implementation

Why separate the two?

Advantages of separating the two for a class:

Similar to what we said about function

Interface

documentation and headers of public attributes and methods

```
class SuperDuperManager:
    """A class responsible for keeping track of all vehicles
    in the system."""
    # === Private Attributes ===
    # _vehicles:
    #     Maps a string that uniquely identifies a vehicle to
    #     the corresponding Vehicle object. For example, _vehicles['car1']
    #     would be a Vehicle object with the id_ 'car1'.
    _vehicles: Dict[str, Vehicle]

    def __init__(self) -> None:
        """Initialize a new SuperDuperManager.

        There are no vehicles in the system when first created.
        """
        self._vehicles = {}

    def add_vehicle(self, vehicle_type: str, id_: str,
                    fuel: int) -> None:
        """Add a new vehicle with the given type, id_, and fuel
        to the system.

        Do nothing if there is already a vehicle with the given id.

        Preconditions:
          - <vehicle_type> is one of 'Car', 'Helicopter', or
            'UnreliableMagicCarpet'.
          - fuel >= 0
        """
        # Check to make sure the identifier isn't already used.
        if id_ not in self._vehicles:
            if vehicle_type == 'Car':
                self._vehicles[id_] = Car(fuel)
            elif vehicle_type == 'Helicopter':
                self._vehicles[id_] = Helicopter(fuel)
            elif vehicle_type == 'UnreliableMagicCarpet':
                self._vehicles[id_] = UnreliableMagicCarpet(fuel)
```

Interface/Implementation

Why separate the two?

Advantages of separating the two for a class:

Similar to what we said about function

(Notice how the documentation for private attributes is not part of the public class docstring / interface)

Implementation

private attributes/methods

all method bodies

```
class SuperDuperManager:
    """A class responsible for keeping track of all vehicles
    in the system."""
    # === Private Attributes ===
    # _vehicles:
    #     Maps a string that uniquely identifies a vehicle to
    #     the corresponding Vehicle object. For example, _vehicles['car1']
    #     would be a Vehicle object with the id_ 'car1'.
    _vehicles: Dict[str, Vehicle]

    def __init__(self) -> None:
        """Initialize a new SuperDuperManager.

        There are no vehicles in the system when first created.
        """
        self._vehicles = {}

    def add_vehicle(self, vehicle_type: str, id_: str,
                    fuel: int) -> None:
        """Add a new vehicle with the given type, id_, and fuel
        to the system.

        Do nothing if there is already a vehicle with the given id.

        Preconditions:
          - <vehicle_type> is one of 'Car', 'Helicopter', or
            'UnreliableMagicCarpet'.
          - fuel >= 0
        """
        # Check to make sure the identifier isn't already used.
        if id_ not in self._vehicles:
            if vehicle_type == 'Car':
                self._vehicles[id_] = Car(fuel)
            elif vehicle_type == 'Helicopter':
                self._vehicles[id_] = Helicopter(fuel)
            elif vehicle_type == 'UnreliableMagicCarpet':
                self._vehicles[id_] = UnreliableMagicCarpet(fuel)
```

Class hierarchy

Interface: the shared public interface defined by the parent class

Advantages of this:

Similar to what we said before, but also –

Client: don’t even need to know what kind you have!

Implementer: can even define new kinds and all client code still works!

This is monumentally powerful.

Class hierarchy

For example, because we have an Employee interface defined by the Employee abstract class, any general client code written to use Employee will now work with subclasses of Employee – even other subclasses written in the future

The client code can rely on the subclasses having methods such as pay and get_monthly_payment

Polymorphism

Same code, different types

A company has a list of employees

Some could be salaried, others hourly

"One code to rule them all"

Same code to pay an employee regardless of their type:

```
class Company:
  """ ...
  """
  employees: List[Employee]

  ...

  def pay_all(self) -> None:
    for emp in self.employees:
      emp.pay(date.today())
```

Terminology: **polymorphism** ("taking multiple forms")

Polymorphism

Employee Example

```
employees: List[Employee]

for emp in self.employees:
    emp.pay(date.today())
```

We don’t know what type this is, but we do know:

It is some kind of Employee.

So it has a pay method, because every subclass inherits that. The Employee class defines a common public interface. *Same code, different types.*

Polymorphism

SuperDuperManager Example

```
_vehicles: Dict[str, Vehicle]
self._vehicles[id].move(new_x, new_y)
```

We don’t know what type this is, but we do know:

It is some kind of Vehicle.

So it has a move method, because every subclass inherits that. The Vehicle class defines a common public interface.

Polymorphism

SuperDuperManager Example

```
_vehicles: Dict[str, Vehicle]
self._vehicles[id].move(new_x, new_y)
```

We say that the highlighted expression is **polymorphic.**

- poly: many; morph: form
- The expression can take many forms.
  It can refer to a Car, an UnreliableMagicCarpet, even a subclass of vehicle that has not been defined yet!

Design decisions

Things to consider when designing classes with inheritance:

What attributes and methods should comprise the **shared public interface**?

For each method, should its implementation be **shared or separate** for each subclass?

Inheritance

Design Decisions

Subclasses use several approaches to recycling the code from their superclass, using the same name

1. Subclass **inherits** superclass methods
2. Subclass **overrides** an abstract method (to **implement** it)
3. Subclass **overrides** an implemented method (to **extend** it)
4. Subclass **overrides** an implemented method (to **replace** it)

Find examples for each from the worksheet..

Inheritance

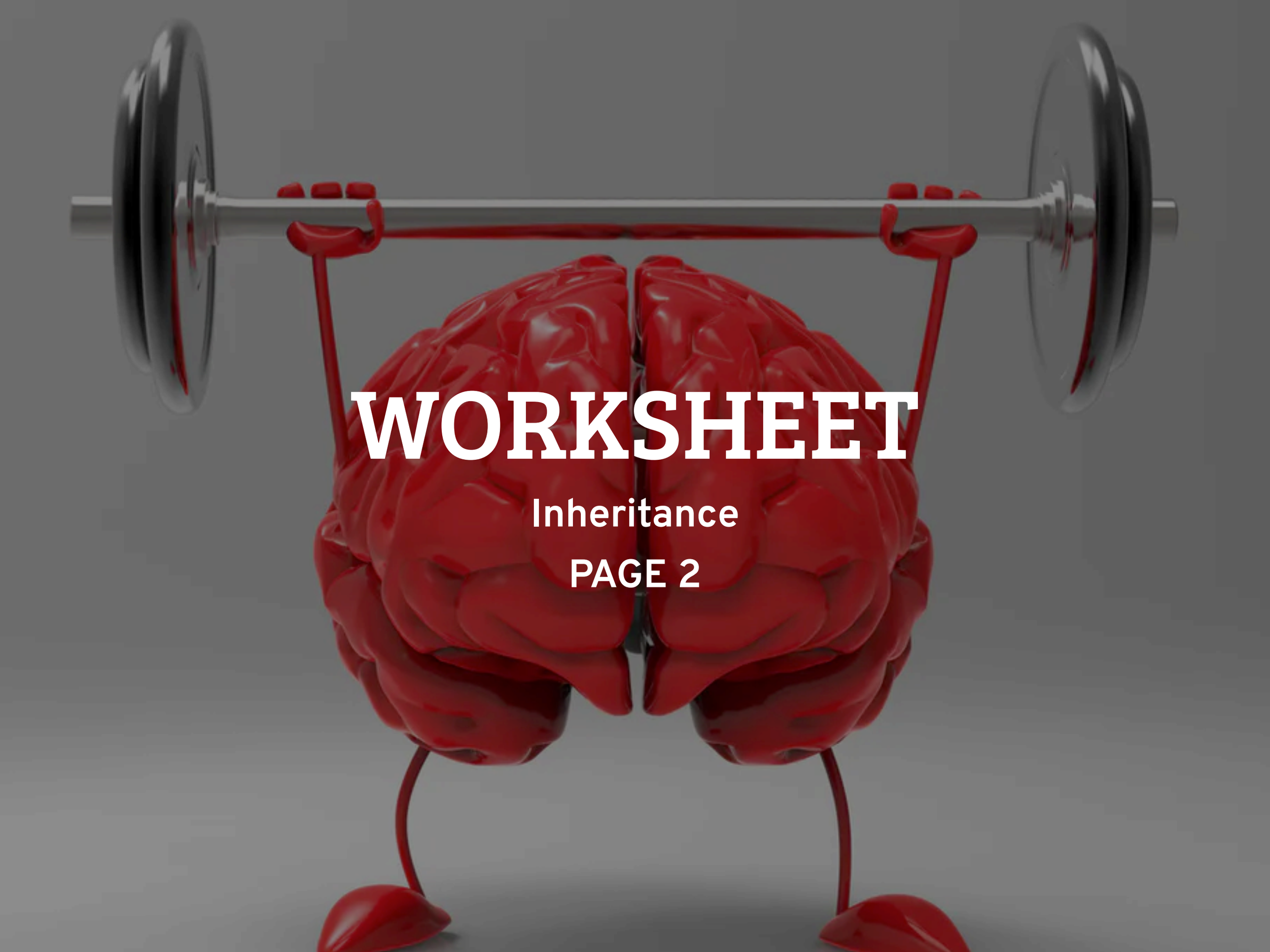
Avoid Duplicate Documentation

Don’t maintain documentation in two places, e.g. superclass and subclass (unless there’s no other choice)

Inherited methods, common public attributes – no need to document again in subclass

Overridden methods – still document them, even if no differences

- Sometimes there may be differences that need to be explained
- Remember though: docstring is part of the public API => it should say how to use a method, NOT how it is implemented internally



WORKSHEET

Inheritance

PAGE 2

Memory model

Let's solidify our understanding of inheritance by drawing a memory model diagram for the following code:

```
class Vehicle:
    def __init__(self, initial_fuel: int,
                 initial_position: Tuple[int, int]) -> None:
        self.fuel = initial_fuel
        self.position = initial_position


class Car(Vehicle):
    def __init__(self, fuel: int):
        Vehicle.__init__(self, fuel, (0, 0))


my_car = Car(50)
```

Catching up and keeping up

Staying on track

Checking in

You've now had ~3 weeks of preps, exercises, lectures, and labs.

Ask yourself:

Am I **confident** with the material covered so far, or am I starting to fall behind?

Do I have **effective strategies** for approaching conceptual and programming problems, or does it feel like I'm often trying random things, or need a lot of help getting started?

If I'm feeling worried, **do I have a plan**, or am I avoiding thinking about it?

Staying on track

Study tips

Before lecture: Do preps & readings

During lecture: Do worksheets, ask questions

After lecture: Revise worksheets, do the lab

- Anything you're unsure about? Discuss with classmates. Study groups can be helpful! Are you on the course Discord?
- Still unsure about things? Write down your questions, bring them to the next office hours. Attend the group office hours (you can even just go to listen in! It's a good review of tricky concepts.)

If you do all these each week, it should really help solidify weekly concepts.

Staying on track

Group Office Hours

Group Office Hours are a place where you are:

welcome to ask anything – no question is too basic

your question will be answered to the whole group

just listening is fine -- very welcome to do so! so, you can attend these even if you don’t have questions

Staying on track

Coding tips

Play with code (coding is best learned using a 'sandbox' approach; it's all about getting as much practice as you can)

Practice using the debugger

Staying on track

Assignment tips

GET STARTED! EARLY!

First step: Read through the handout/starter, write down all the tasks you need to do.

Break it down into the tiniest tasks – make a list, and then check things off as you work (it really helps you visualize your progress, and get a sense of how much you have left to do).

Staying on track

Wellness tips

Remember to take some time off to relax every now and then :)

Have you tried/considered meditation?

https://www.healthline.com/nutrition/12-benefits-of-meditation#1.-Reduces-stress

https://news.harvard.edu/gazette/story/2018/04/harvard-researchers-study-how-mindfulness-may-change-the-brain-in-depressed-patients/

Some good apps: Headspace, Insight Timer, Playne

Mental health support at UofT:

https://studentlife.utoronto.ca/task/access-mental-health-support-at-u-of-t/, https://mentalhealth.utoronto.ca/

staying on track

Any more tips?

Feel free to share what works well for you :)

Extra Notes and Examples

Inheritance

Inheritance

Parent-Child Example

```
class Parent:
    def __init__(self, x):
        self.x = x

    def say_hello_parent(self):
        print("I am a parent!")

class Child(Parent):

    # Child shares all of Parent's method
    # This includes the constructor

    def say_hello_child(self):
        print("I am a child!")
```

Inheritance

Parent-Child Example

```
class Parent:
    def __init__(self, x):
        self.x = x

    def say_hello_parent(self):
        print("I am a parent!")

class Child(Parent):

    # Child shares all of Parent's method
    # This includes the constructor

    def say_hello_child(self):
        print("I am a child!")
```

```
c = Child(3)
print(c.x)
c.say_hello_child()
c.say_hello_parent()

p = Parent(4)
print(p.x)
p.say_hello_parent()
p.say_hello_child()
```

What would this code print?

Inheritance

Parent-Child Example

```
class Parent:
    def __init__(self, x):
        self.x = x

    def say_hello_parent(self):
        print("I am a parent!")

class Child(Parent):

    # Child shares all of Parent's method
    # This includes the constructor

    def say_hello_child(self):
        print("I am a child!")
```

```
c = Child(3)
print(c.x)
c.say_hello_child()
c.say_hello_parent()


p = Parent(4)
print(p.x)
p.say_hello_parent()
p.say_hello_child()
```

Output of the above code

```
3
I am a child!
I am a parent!
4
I am a parent!
Traceback (most recent call last):
  File "/Users/sadiasharmin/2020Winter/csc108_s20/week 9/parent_child.py", line
38, in <module>
    p.say_hello_child()
AttributeError: 'Parent' object has no attribute 'say_hello_child'
```

Inheritance

Parent-Child Example

```
c = Child(3) # this passes in 3 to Parent.__init__(self, x)
print(c.x)
c.say_hello_child()
c.say_hello_parent() # the child has access to all of Parent's methods

p = Parent(4)
print(p.x)
p.say_hello_parent()
# p.say_hello_child() # THIS WILL GIVE AN ERROR.
# -- A parent does NOT have access to its child's methods/attributes
```

Inheritance

Parent-Child Example

```
# A Parent can have more than one Child:
class Child2(Parent):

    # This class also shares all of Parent's method and attributes
    # But a Child is also able to OVERRIDE its Parent's methods

    # This child has an x like Parent, but also its own attribute, y
    # So we OVERRIDE the Parent's constructor with this:
    def __init__(self, x, y):
        super().__init__(x) # Let Parent handle its part
        # Alternative way to do the above: Parent.__init__(self, x)
        self.y = y # Let Child handle its unique part
```

Inheritance

Parent-Child Example

```
class Parent:
    def __init__(self, x):
        self.x = x

    def say_hello_parent(self):
        print("I am a parent!")

class Child(Parent):

    # Child shares all of Parent's method and attributes
    # This includes the constructor

    def say_hello_child(self):
        print("I am a child!")

# A Parent can have more than one Child:
class Child2(Parent):

    # This class also shares all of Parent's method and attributes
    # But a Child is also able to OVERRIDE its Parent's methods

    # This child has an x like Parent, but also its own attribute, y
    # So we OVERRIDE the Parent's constructor with this:
    def __init__(self, x, y):
        super().__init__(x) # Let Parent handle its part
        # Alternative way to do the above: Parent.__init__(self, x)
        self.y = y # Let Child handle its unique part
```

```
>>> c = Child2(2)
>>> c = Child2(2, 3)
>>> c.x
>>> c.y
>>> c.say_hello_parent()
>>> c.say_hello_child()
```

What is the output?

Inheritance

Parent-Child Example

```
>>> c = Child2(2)
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    c = Child2(2)
TypeError: __init__() missing 1 required positional argument: 'y'
>>> c = Child2(2, 3)
>>> c.x
2
>>> c.y
3
>>> c.say_hello_parent()
I am a parent!
>>> c.say_hello_child()
Traceback (most recent call last):
  File "<pyshell#45>", line 1, in <module>
    c.say_hello_child()
AttributeError: 'Child2' object has no attribute 'say_hello_child'
```

Inheritance

Parent-Child Example

```
# Here is another example of a Child
# overriding BOTH of the parent's methods

class Parent:
    def __init__(self, x):
        self.x = x

    def print_x(self):
        print(self.x)

class Child(Parent):
    def __init__(self, x, y):
        Parent.__init__(self, x)
        self.y = y

    def print_x(self):
        print("No, I don't want to.")

c = Child(3, 4)
c.print_x()
```