BY: Zhu_Ming

<CS1010S>

Tutorial 8

Object-Oriented Programming

Lecture_Recap.tut08

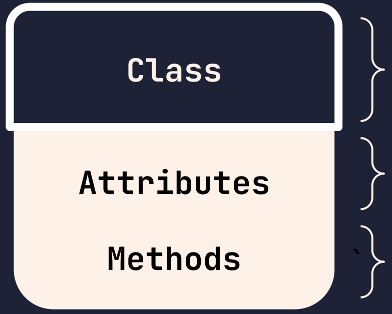
Lecture Recap

1. Object-Oriented Programming
   - Abstraction & Encapsulation
   - Inheritance
   - Polymorphism

2. Optional argument & Default argument

1. Object-Oriented Programming (OOP)

- programming paradigm based on the concept of **"objects"**
- data (often known as attributes or properties),
- method (a.k.a function/procedures)

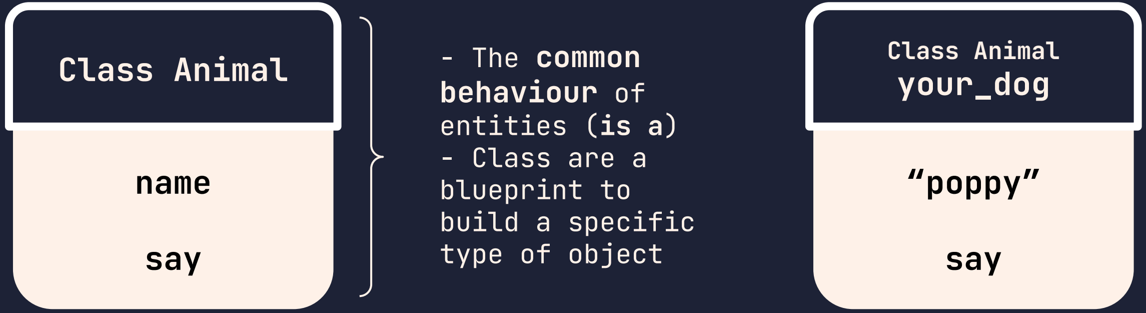


- The **common behaviour** of entities (**is a**)
- Class are a blueprint to build a specific type of object

- **Properties** that this class has (**has a**)

- **Function** that this class can do (**can do**)

Class vs Instance/Object



Class Animal

name

say

- The **common behaviour** of entities (**is a**)
- Class are a blueprint to build a specific type of object

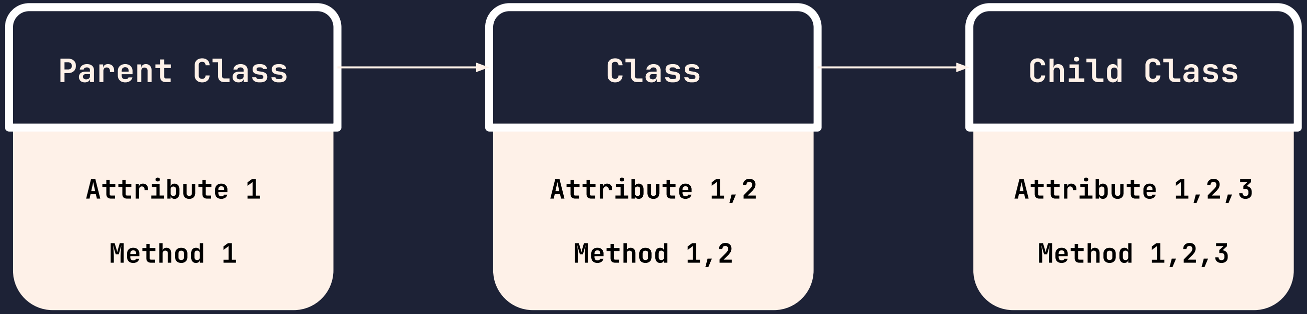
Class Animal
your_dog

“poppy”

say

```
Class Animal: # Creating the class
   def __init__(self, name):
   ...

your_pet = Animal(“poppy”) # Creating object/instance belongs to Animal Class

isinstance(your_pet, Animal) # True
```

- Inheritance



Parent Class

Attribute 1

Method 1

Class

Attribute 1,2

Method 1,2

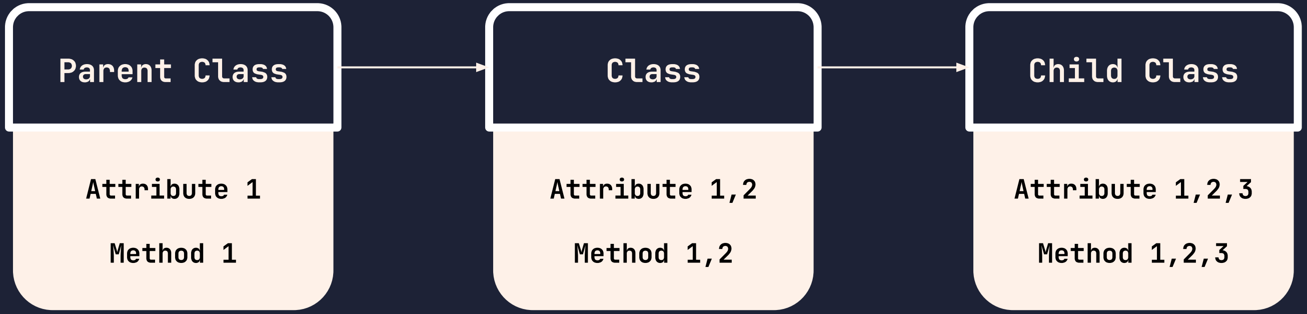
Child Class

Attribute 1,2,3

Method 1,2,3

- A **is a** B → A (child class) is a subclass of B (parent class)
- Subclasses inherit ALL attributes/methods from its superclass and can implement additional attributes/methods.

- **Inheritance**



Parent Class

Attribute 1

Method 1

Class

Attribute 1,2

Method 1,2

Child Class

Attribute 1,2,3

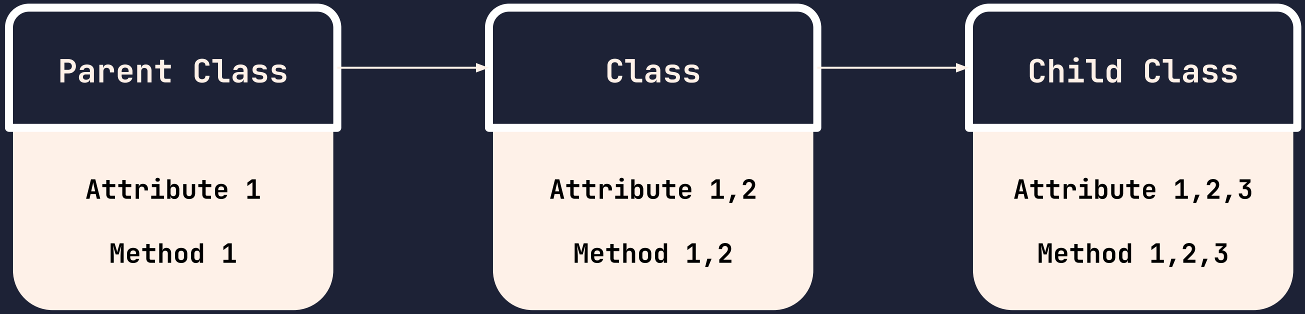
Method 1,2,3

```
class Animal:
   def __init__(self, name):
      self.name

   def say(self, voice):
      return f’{self.name} {voice}’
```

```
class Dog():
   def __init__(self, name):
      self.name

   def say(self, voice):
      return f’{self.name} {voice}’
```

Lecture_Recap.tut08

- # Inheritance



Parent Class

Attribute 1

Method 1

Class

Attribute 1,2

Method 1,2

Child Class

Attribute 1,2,3

Method 1,2,3

```
class Animal:
    def __init__(self, name):
        self.name

    def say(self, voice):
        return f’{self.name} {voice}’
```

```
class Dog(Animal):
    def __init__(self, name):
        super().__init__(name)

    def say(self, voice):
        return super().say(voice)
```

Lecture_Recap.tut08

Abstraction

- Reduce redundant code
- Same as ADT/Data Structures. We only tell the user what they can do with this class, but not how it was done.“Tell, Don’t Ask” principle

```
class Animal:
    def __init__(self, name):
        self.name

    def say(self, voice):
        return f’{self.name} {voice}’
```

Encapsulation

- Information hiding
- Every object/instance of a class has a unique set of data for its own. They usually don’t affect each other!

```
class Dog(Animal):
    def __init__(self, name):
        self.name

    def say(self, voice):
        return f’{self.name} {voice}’
```

Lecture_Recap.tut08

- ## Abstraction

  - Reduce redundant code
  - Same as ADT/Data Structures. We only tell the user what they can do with this class, but not how it was done.“Tell, Don’t Ask” principle

```
class Animal:
   def __init__(self, name):
      self.name

   def say(self, voice):
      return f’{self.name} {voice}’
```

- ## Encapsulation

  - Information hiding
  - Every object/instance of a class has a unique set of data for its own. They usually don’t affect each other!

```
class Dog(Animal):
   def __init__(self, name):
      super().__init__(name)

   def say(self, voice):
      return super().say(voice)
```

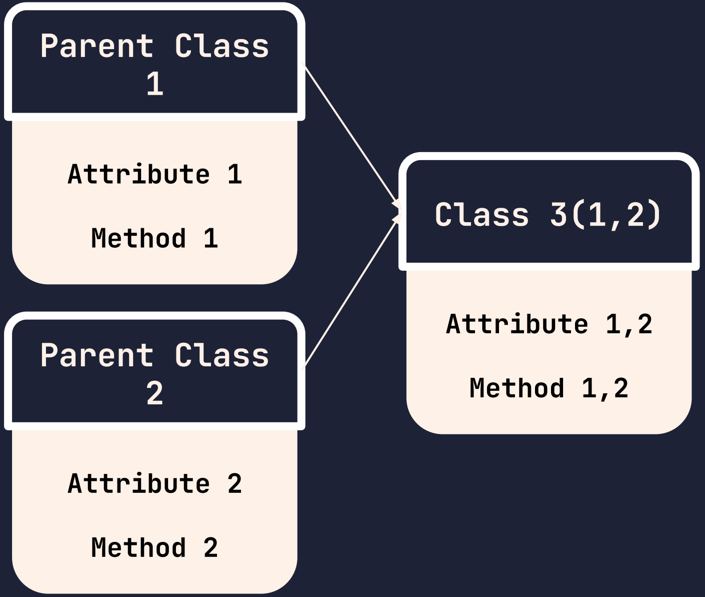
- ## Multiple Inheritance

Python allow multiple inheritance!!

Which super() is being invoked, is determined by MRO

MRO = Method Resolution Order

```
>>> 3.mro()
[<class '__main__.3'>,
 <class '__main__.1'>,
 <class '__main__.2'>,
 <class 'object'>]
```



PythonTips!

By default, if class is inherited from class **object**

class Animal ⇒ class Animal(object)

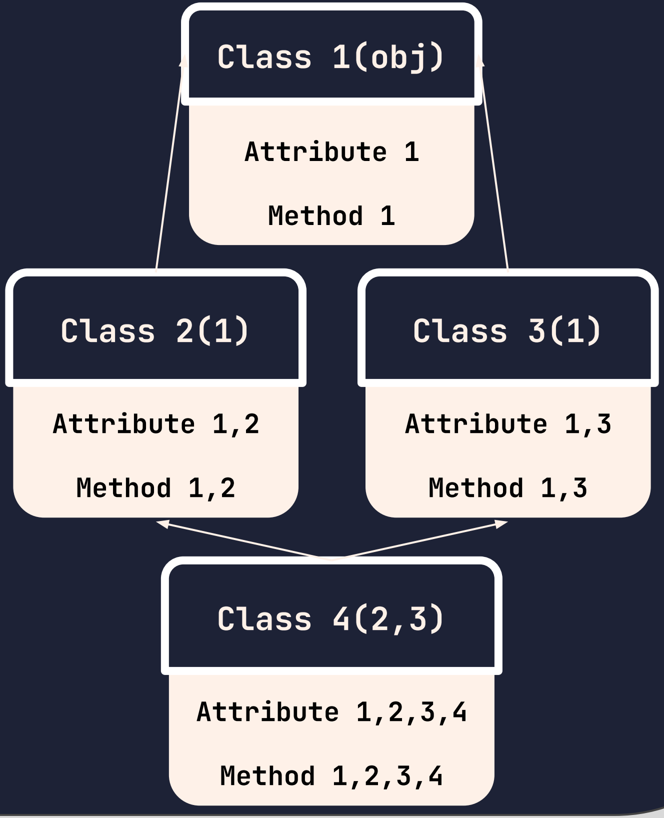
- ## Diamond Inheritance

Class order: **4, 2, 3, 1, object**

It very important concept! It will be tested in final very often.

However, Diamond Inheritance is something unique to Python (most of the other language prohibit Diamond Inheritance.

Always use class.mro() to check the order!!

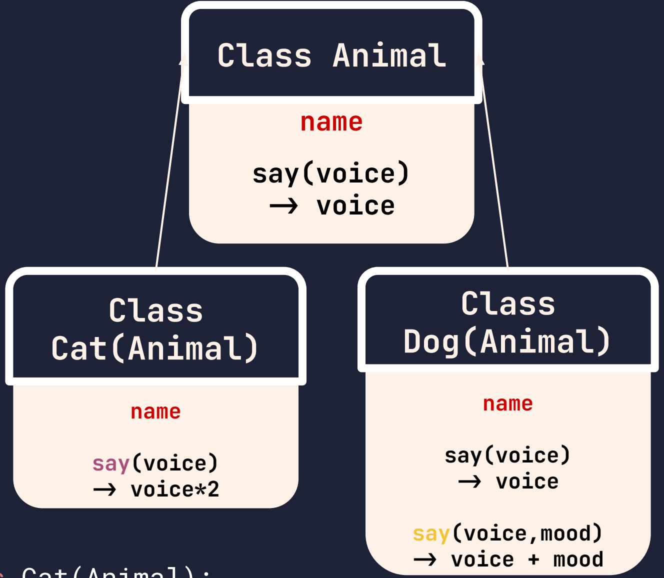


- **Polymorphism**

Overriding: Identical methods (say)

Overloading: Same method name, different arguments (say)

```
class Animal:
   def __init__(self, name): self.name

   def say(self, voice):
      return f’{voice}’

 class Dog(Animal):
    def __init__(self, name):
       super().__init__(name)

    def say(self, voice):
       return f’{voice}’

    def say(self, voice, mood):
       return f’{voice} + {mood}’
```



```
class Cat(Animal):
   def __init__(self, name):
      super().__init__(name)

   def say(self, voice):
      return voice*2
```

Optional / Default argument

```python
def foo(*args):
```

- args will be packed into a tuple for processing in the function body
- args[0] / for arg in args
- can be any number of arguments, even no argument!

```python
def foo(arg1, arg2, arg3 = None):
```

- if arg3 is not supplied, will default to None
- can default to anything you want

BY: Zhu_Ming

Any Questions?

Question_1.tut08

The essential properties of a Thing are as follows:

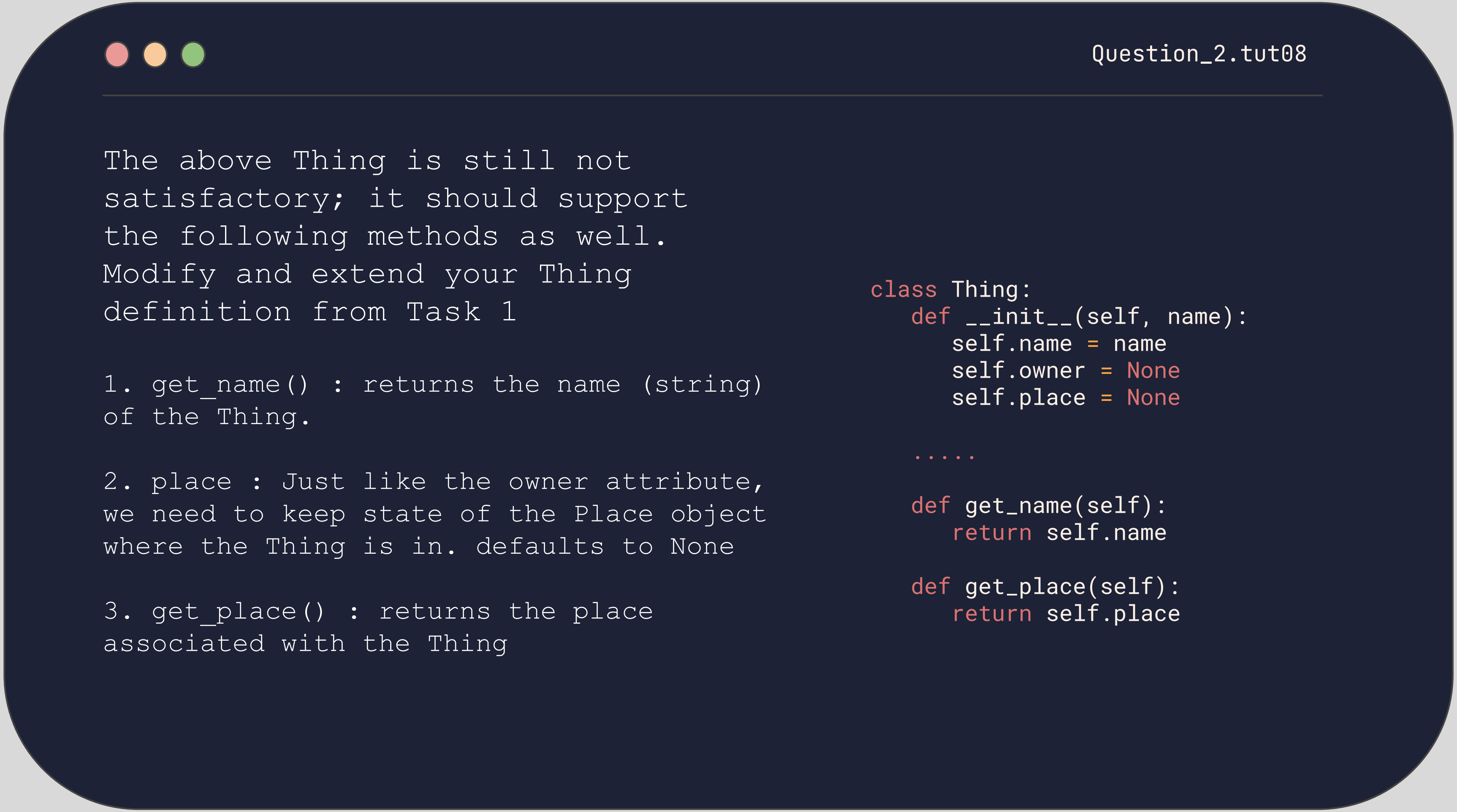
1. The constructor should take in 1 parameter, the name of the Thing.

2. owner : an attribute that stores the owner object of the Thing, usually a Person object. Defaults to None

3. is_owned() : returns True if the thing is “owned”, False otherwise.

4. get_owner() : returns the Person object who owns the Thing object

```
class Thing:
    def __init__(self, name):
        self.name = name
        self.owner = None

    def is_owned(self):
        return self.owner is not None

    def get_owner(self):
        return self.owner
```



Question_2.tut08

The above Thing is still not satisfactory; it should support the following methods as well. Modify and extend your Thing definition from Task 1

1. get_name() : returns the name (string) of the Thing.

2. place : Just like the owner attribute, we need to keep state of the Place object where the Thing is in. defaults to None

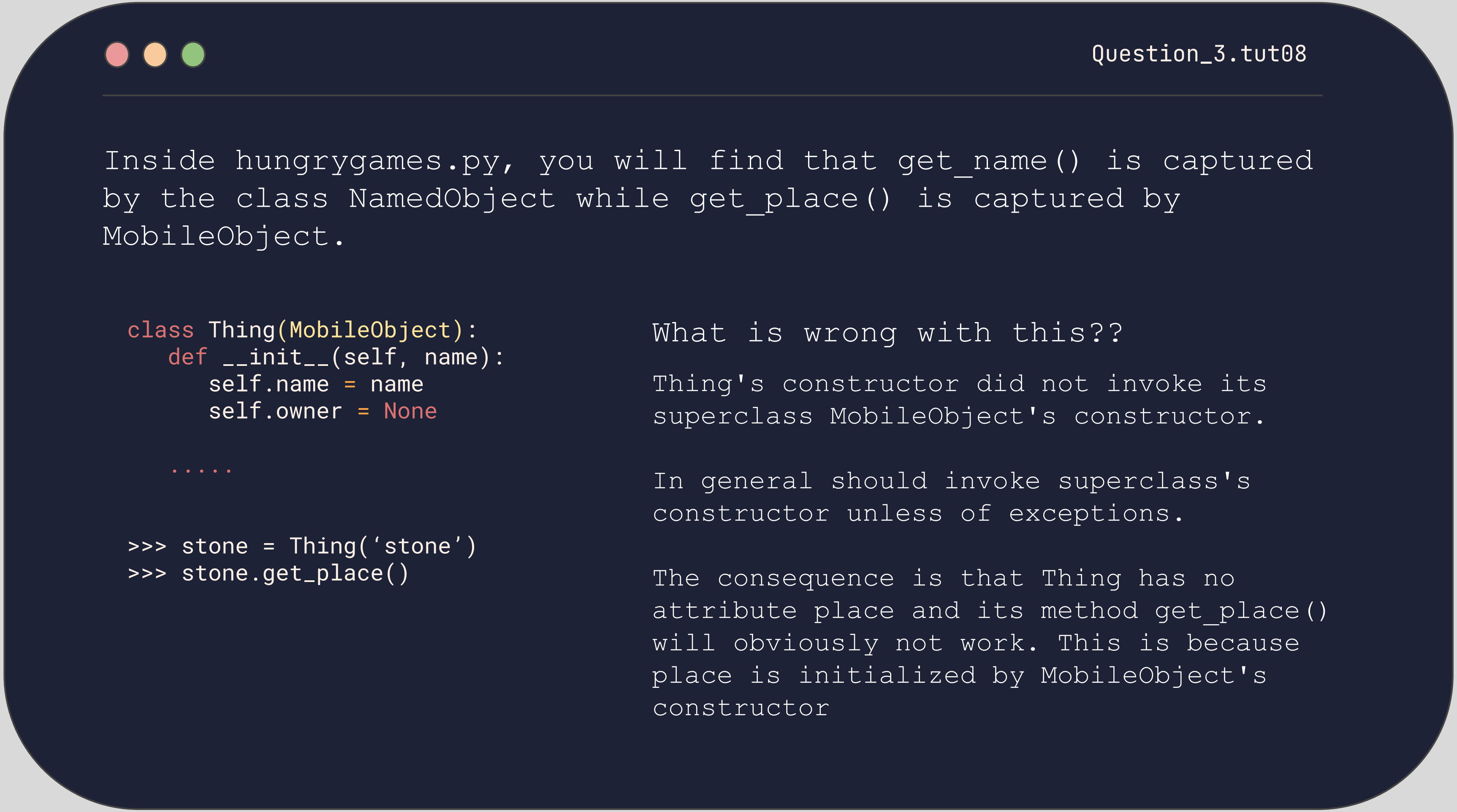
3. get_place() : returns the place associated with the Thing

```
class Thing:
    def __init__(self, name):
        self.name = name
        self.owner = None
        self.place = None

    .....

    def get_name(self):
        return self.name

    def get_place(self):
        return self.place
```

Question_3.tut08

Inside hungrygames.py, you will find that get_name() is captured by the class NamedObject while get_place() is captured by MobileObject.

```
class Thing:
   def __init__(self, name):
      self.name = name
      self.owner = None
      self.place = None

   .....
```



Question_3.tut08

Inside hungrygames.py, you will find that get_name() is captured by the class NamedObject while get_place() is captured by MobileObject.

```
class Thing(MobileObject):
   def __init__(self, name):
      self.name = name
      self.owner = None

   .....


>>> stone = Thing(‘stone’)
>>> stone.get_place()
```

What is wrong with this??

Thing's constructor did not invoke its superclass MobileObject's constructor.

In general should invoke superclass's constructor unless of exceptions.

The consequence is that Thing has no attribute place and its method get_place() will obviously not work. This is because place is initialized by MobileObject's constructor

Question_3.tut08

Inside hungrygames.py, you will find that get_name() is captured by the class NamedObject while get_place() is captured by MobileObject.

```
class Thing(MobileObject):
   def __init__(self, name):
      super().__init__(None)
      self.name = name
      self.owner = None

   .....

>>> stone = Thing(‘stone’)
>>> stone.get_place() # None
```

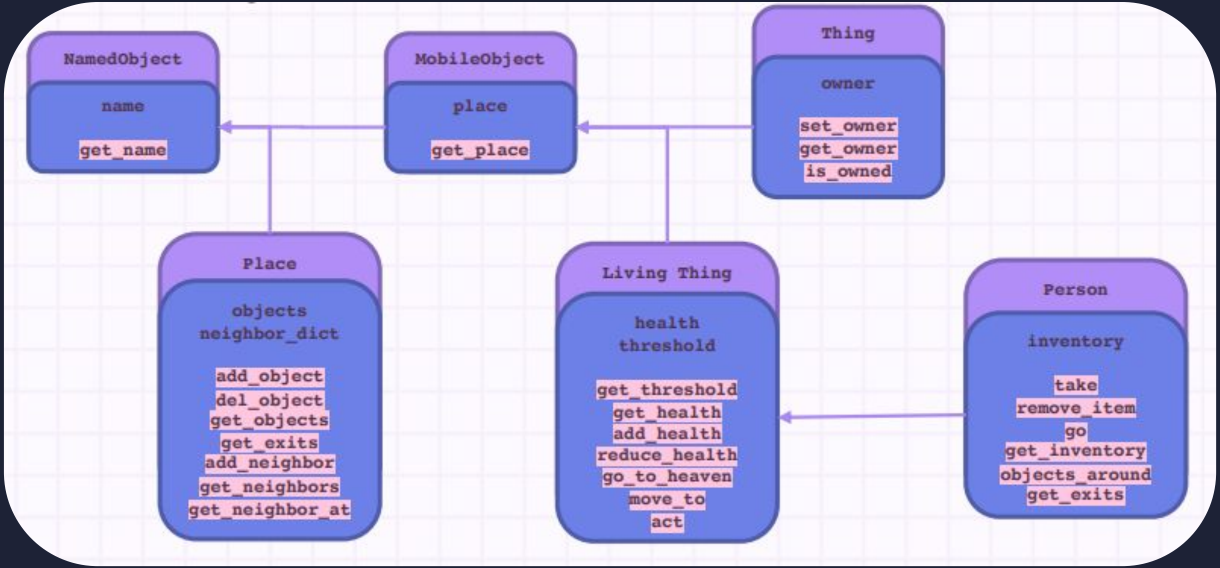
What is wrong with this??

Thing's constructor did not invoke its superclass MobileObject's constructor.

In general should invoke superclass's constructor unless of exceptions.

The consequence is that Thing has no attribute place and its method get_place() will obviously not work. This is because place is initialized by MobileObject's constructor

Inheritance diagram



Question_5.tut08

Suppose we evaluate the following statements:

```
beng = Person("beng", 100, 1)
ice_cream = Thing(“ice cream”)
ice_cream.owner = beng
```

Come up with statements whose evaluation will reveal all the properties of ice_cream and verify that its (new) owner is indeed beng.

```
print(ice_cream.get_owner() is beng) # True

print(ice_cream.__dict__) # print all the attributes in form of dictionary
# {'name': 'ice_cream',
# 'place': None,
# 'owner': <__main__.Person object at 0x000001D75C6E9F10>}
```

Beng’s reference ID,
because beng point to Person object

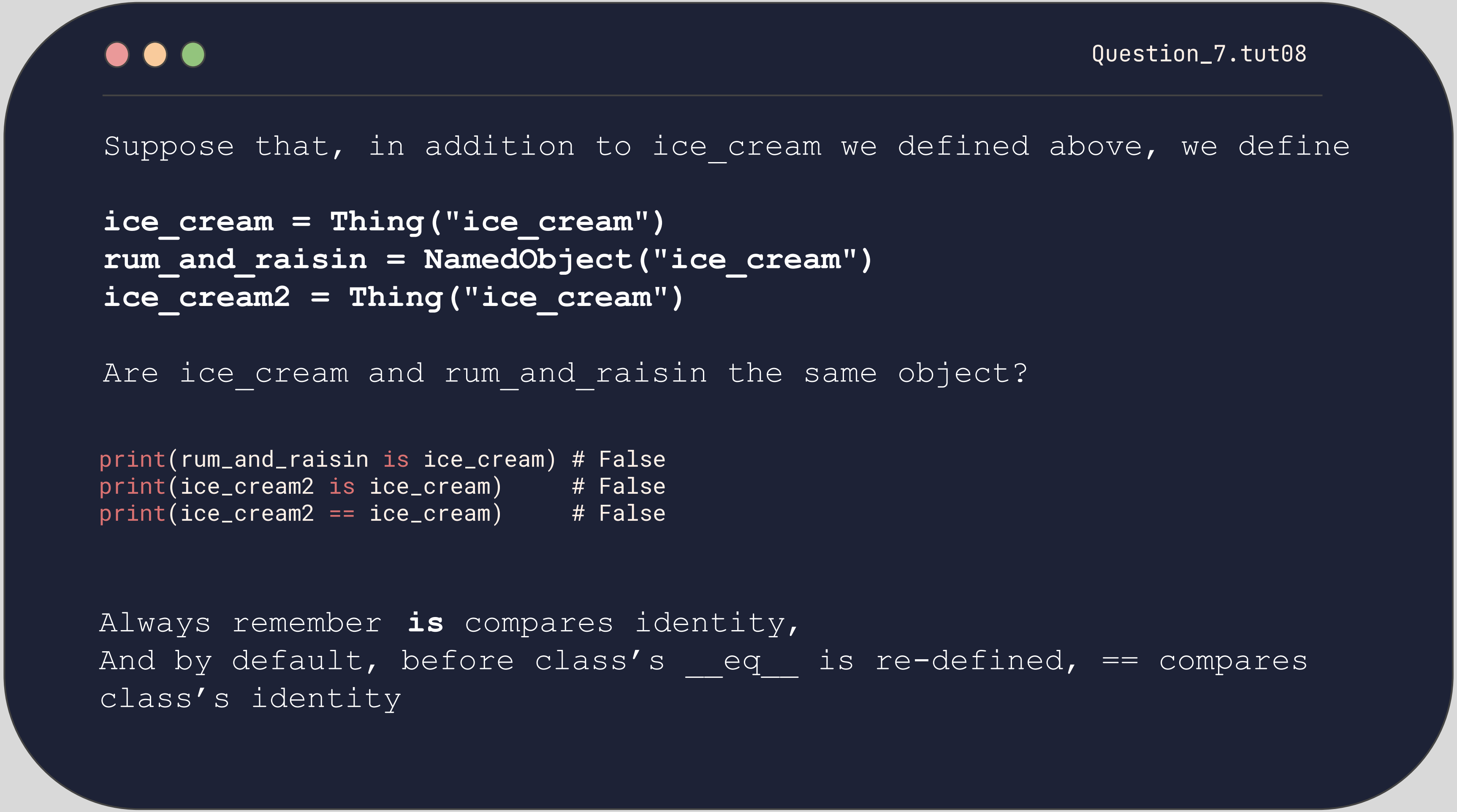
Question_6.tut08

Now suppose we evaluate the following statements:

```
ice_cream = Thing(“ice cream”)
ice_cream.owner = beng
beng.ice_cream = ice_cream
```

Is there anything wrong with the last two statements? What’s the moral of the story?

It works!! Perfectly fine!
But it is a bad practices...



Question_7.tut08

Suppose that, in addition to ice_cream we defined above, we define

```
ice_cream = Thing("ice_cream")
rum_and_raisin = NamedObject("ice_cream")
ice_cream2 = Thing("ice_cream")
```

Are ice_cream and rum_and_raisin the same object?

```
print(rum_and_raisin is ice_cream) # False
print(ice_cream2 is ice_cream)     # False
print(ice_cream2 == ice_cream)     # False
```

Always remember **is** compares identity,
And by default, before class’s __eq__ is re-defined, == compares class’s identity

Question_7.tut08

Now let’s make two similar objects in our world.

```
burger1 = Thing(“burger”)
burger2 = Thing(“burger”)
```

Are burger1 and burger2 the same object? Would burger1 == burger2 evaluate to True?

```
print(burger1 is burger2) # False
print(burger1 == burger2) # False
```

Question_8.tut08

Every object has an __eq__ method. We can override it to make it work the way we want it to!

```
class Thing(MobileObject):
   def __eq__(self, other):
      return isinstance(other, Thing) and \
             self.get_name() == other.get_name() and \
             self.get_place() == other.get_place()
```

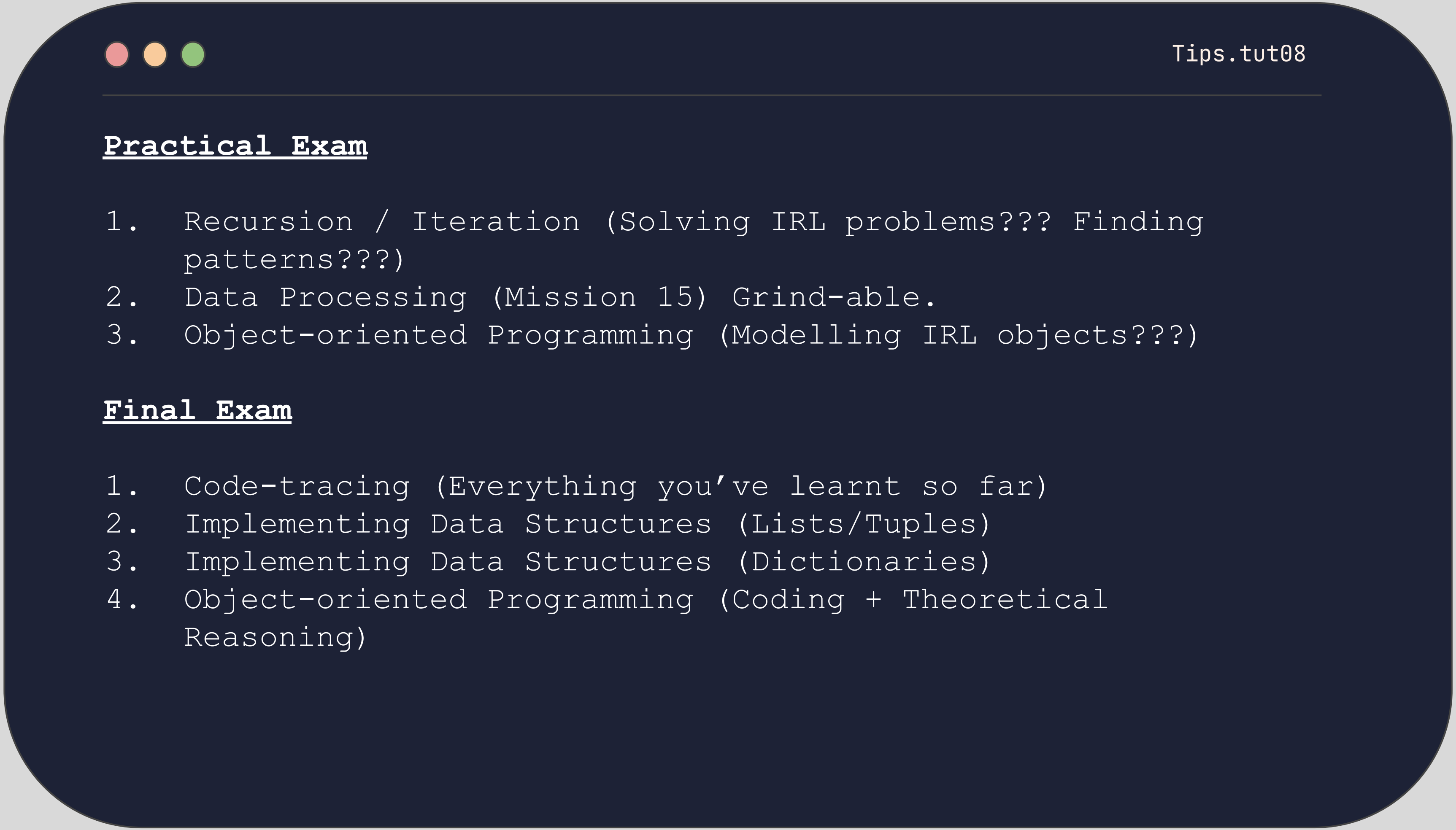
type(x) → returns the Class of x

isinstance(x, Class) → returns a boolean: True if x is an instance of Class or a subclass of Class.

Thank You!!

The End

See you next lesson

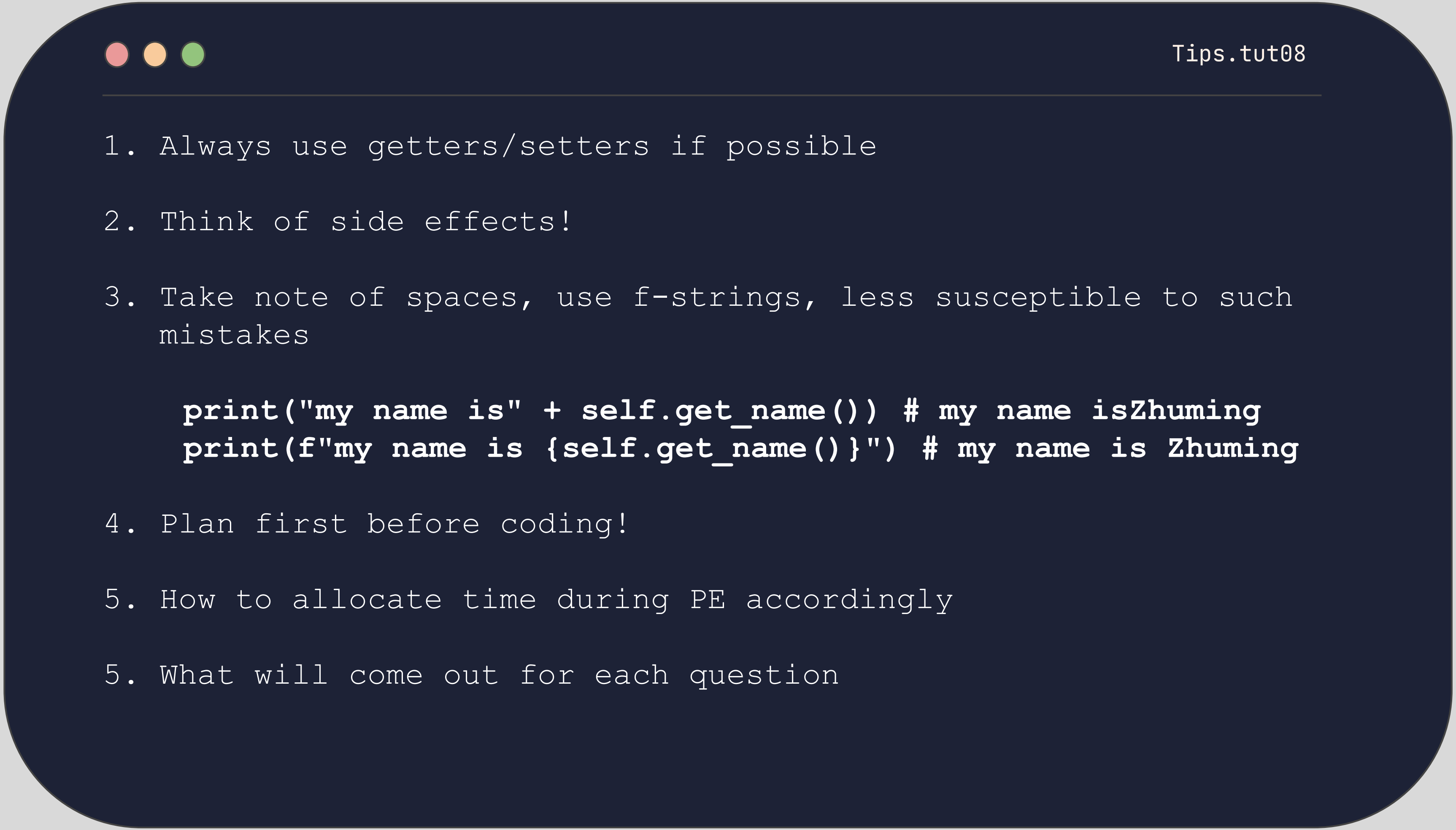


Practical Exam

1. Recursion / Iteration (Solving IRL problems??? Finding patterns???)
2. Data Processing (Mission 15) Grind-able.
3. Object-oriented Programming (Modelling IRL objects???)

Final Exam

1. Code-tracing (Everything you've learnt so far)
2. Implementing Data Structures (Lists/Tuples)
3. Implementing Data Structures (Dictionaries)
4. Object-oriented Programming (Coding + Theoretical Reasoning)



1. Always use getters/setters if possible

2. Think of side effects!

3. Take note of spaces, use f-strings, less susceptible to such mistakes

```
print("my name is" + self.get_name()) # my name isZhuming
print(f"my name is {self.get_name()}") # my name is Zhuming
```

4. Plan first before coding!

5. How to allocate time during PE accordingly

5. What will come out for each question

BY: Zhu_Ming

Any Questions?