Interfaces and Abstract Classes

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Poll: Which of these would make a good superclass / subclass pair?

1. Rectangle / Square
2. Sophomore / Freshman
3. Mammal / Elephant
4. Building / Window

Rectangle and Triangle are both Shapes

```python
class Shape():
    def get_area(self) -> float:
        pass

    def get_perimeter(self) -> float:
        pass
```

But we're unable to implement these methods in `Shape`

```python
class Rectangle(Shape):
    def __init__(self,
        width: float,
        height: float
    ) -> None:
        self.width = width
        self.height = height

    def get_area(self) -> float:
        return self.width * \
            self.height

    def get_perimeter(self) -> float:
        return 2 * \
            (self.width + self.height)
```

Rectangle and Triangle are both Shapes

```python
class Shape():
    def get_area(self) -> float:
        pass

    def get_perimeter(self) -> float:
        pass
```

But we're unable to implement these methods in `Shape`

So we leave them as *abstract methods*.

Abstract method: a method with no implementation

Two abstract methods in `Shape`:

- `get_area()`
- `get_perimeter()`

Implementation is left to the subclasses.

Does leaving methods un-implemented make us uncomfortable?

```
shape = Shape()
print(shape.get_area())  # None
```

What if we instantiate a `Shape` and ask for its (nonexistent) area?

(Or what if we forget to implement the abstract method in its subclass?)

How embarassing. Let's prevent that.

The `ABC` module

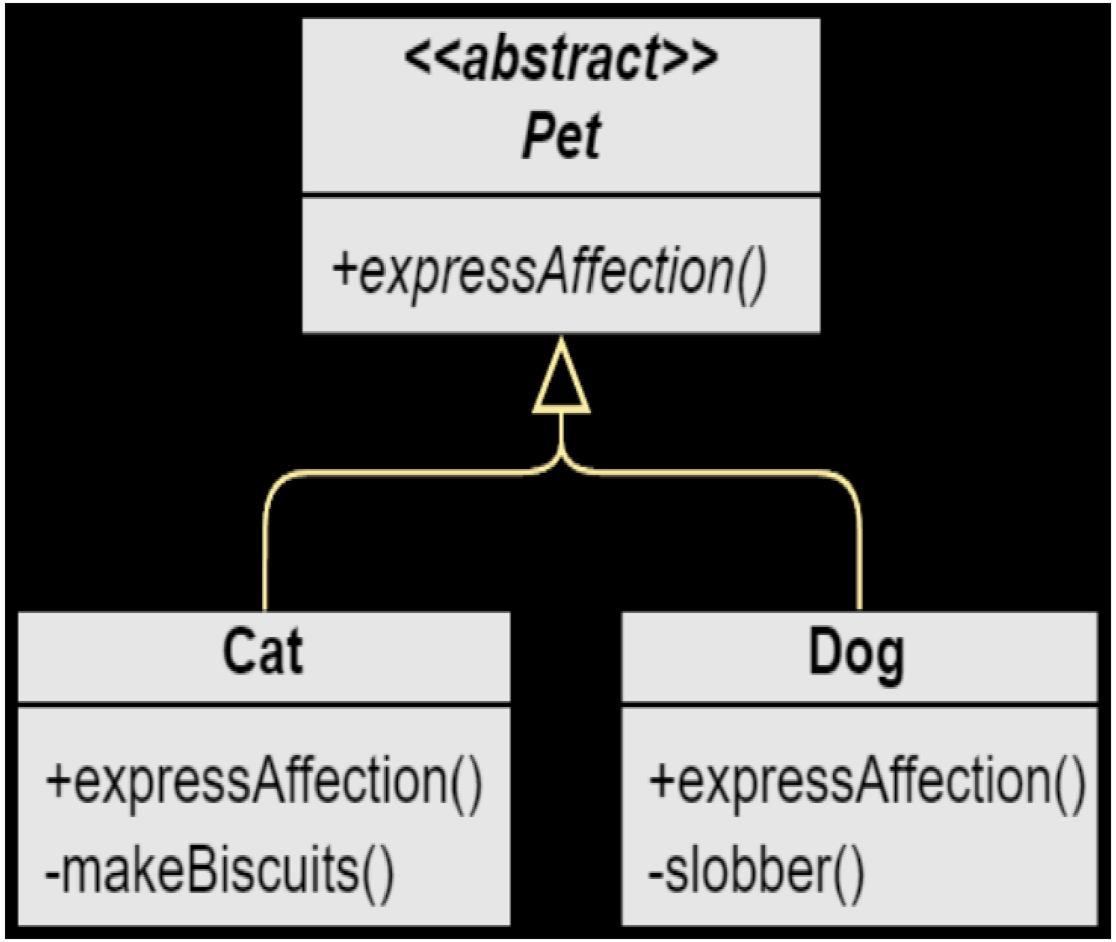
Prevents instantiating a class that has an abstract method (even an inherited one)

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def get_area(self) -> float:
        pass

    @abstractmethod
    def get_perimeter(self) -> float:
        pass

shape = Shape()  # TypeError
```

- To instantiate a class that inherits a method decorated with `@abstractmethod`, we must overwrite it with a **concrete** (non-abstract) method.
- If a subclass doesn't implement all abstract methods, the `ABC` module will raise a `TypeError` when you try to instantiate it.

https://giffiles.alphacoders.com/207/207370.gif

Poll: Which ones are legal?



1. pet1: Pet = Pet()
2. cat1: Cat = Cat()
3. dog1: Dog = Dog()
4. pet2: Pet = Cat()
5. cat2: Cat = Dog()
6. dog2: Dog = Pet()

Poll: Does this work?

```python
for pet in [Cat(), Dog(), Cat()]:
    pet.express_affection()
```

1. Yes
2. No
3. I don't know
4. I looked ahead in the online lecture notes and found the answer

```python
from abc import ABC, abstractmethod

class Pet(ABC):
    @abstractmethod
    def express_affection(self) -> None:
        pass

class Cat(Pet):
    def express_affection(self) -> None:
        self.make_biscuits()

    def make_biscuits(self) -> None:
        print('Making biscuits')

class Dog(Pet):
    def express_affection(self) -> None:
        self.slobber()

    def slobber(self) -> None:
        print('Slobbering')

for pet in [Cat(), Dog(), Cat()]:
    pet.express_affection()
```

It works. Here's the output.

```
Making biscuits
Slobbering
Making biscuits
```

Let's visualize it in pythontutor.com

Interfaces

User interface: describes the behavior without telling you how it's implemented

Interface: describes the behavior of a class without implementing its methods

In Python, an interface is an abstract class (`ABC`) where all methods are `@abstractmethod`

An interface is a contract: if a class wants to "implement" the interface, that class must implement each specified method.

- Different classes can implement the same methods in different ways
- Classes can also have additional methods which are not specified in the interface

Poll: (Designing an interface) What should all classes which implement the interface `Cat` be able to do?

1. Sleep
2. Roar
3. Meow
4. Bark
5. Knead

```python
class Cat(ABC): pass

class Roarable(ABC):
    @abstractmethod
    def roar(self) -> None:
        pass

class Lion(Cat, Roarable):
    def roar(self) -> None:
        print('ROAR')

class AsiaticLion(Lion): pass

class HouseCat(Cat): pass

class Dragon(Roarable):
    def roar(self) -> None:
        print('GRRRR')

cacophony: list[Roarable] = list()
```

Poll: Which types can be instantiated and put into the list `cacophony`?

1. Lion
2. AsiaticLion
3. HouseCat
4. Dragon
5. Roarable

Interfaces vs abstract classes

Interface: a "contract" that specifies what a class should be able to do.

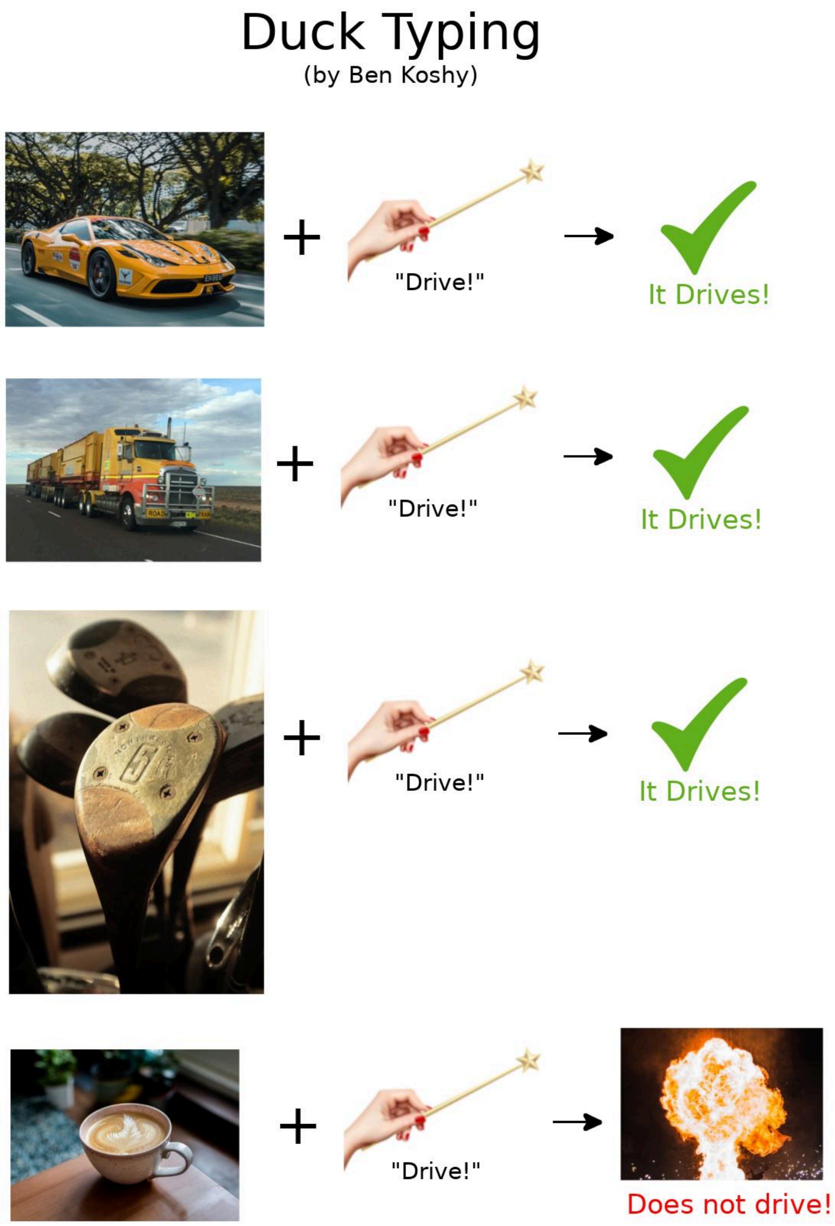
Abstract class: a class that happens to need abstract methods (because it is too non-specific).

Interfaces vs abstract classes: controversy with the `ABC` module

- `ABC` module was originally designed to help with abstraction:
  - inheritance hierarchies where we happen to need abstract methods
- Using the `ABC` module to design interfaces:
  - is commonplace in modern Python
  - but some argue that that is not what it was originally designed for

The controversy:

- Interfaces serve different purposes than abstract superclasses.
- The `ABC` module was created for abstract superclasses, not interfaces.



More controversy: using interfaces when Python uses duck typing

Duck Test: "If it walks like a duck and it quacks like a duck, then it must be a duck."

- Python's types are not enforced
- We can pass a variable of any type to a function expecting arg of any type
- If the variable has the necessary methods / attributes to work in that context (to quack), great! **It's a duck.**

Why are we teaching interfaces when they're controversial?

- It helps us to **detect errors early**, not while running the program
- We prioritize **readability**, and making contracts explicit through interfaces helps with **"self-documentation"**
- It helps us to **keep track of types' capabilities**, especially in large codebases
- When designing APIs for others to use, it helps **ensure that implementers implement all required methods**
- It **prepares students for future courses** where types and interfaces are fundamental concepts

Python's beautiful alternative to interfaces: Contracts

- Interface using `ABC` is an explicit contract: classes must follow the rules (enforced)
  - Early error detection, readability, easier to follow, requires other implementors to follow our rules, teaches fundamental concepts
- Python's built-in contracts are followed by convention but not enforced
  - Includes things that interfaces cannot include (like specifying *what the methods should do*, rather than simply listing the methods that need to be implemented)

A Python contract: `len()`

"Length protocol" / "size protocol":

- `def __len__(self) -> int` which returns a non-negative `int`
- this is what is returned by the `len()` function

```python
class Cat:
    def __len__(self) -> int:
        return 900

print(len(Cat()))  # 900
```

A Python contract: `len()`

There is an interface in `ABC` which enforces that we implement `__len__()`:

```python
from collections.abc import Sized

class Cat(Sized):
    def __len__(self) -> int:
        return 900

print(len(Cat()))  # 900
```

Neglecting to implement `__len__()` (or having it return a negative number) will cause an error.

A Python contract: the `in` operator

"Membership test protocol" / "containment protocol":

- When you use `in`, Python calls `__contains__()`
- Protocol works on its own, but often enforced using `collections.abc.Container`

```python
from collections.abc import Container

class Document(Container[str]):
    def __init__(self, text: str):
        self.words = text.split()

    def __contains__(self, word: object) -> bool:
        if not isinstance(word, str):
            raise TypeError
        return word in self.words

print('hi' in Document('hi this is mini'))  # True
print('cat' in Document('hi this is mini'))  # False
```

Poll:

1. What is your main takeaway from today?

2. What would you like to revisit next time?