Programming 2

Dries Decuyper
Serhat Erdogan
Janne Gilis
Bart Thumas

Abstract Methods
Abstract Properties

AGENDA

- Abstract Methods
- Abstract Properties

Abstract Methods

- Abstract Classes
- Abstract Methods
- Abstract Class vs. Concrete Class
- Abstract Class Inheritance
- Abstract Methods and Polymorphism

ABSTRACT CLASSES

```
from abc import ABC

class Shape(ABC):
    ...
```

- Abstract classes are classes that cannot be instantiated and may contain abstract methods
- Use of the ABC (Abstract Base Class) module in Python

ABSTRACT METHODS

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass
```

- Abstract methods are methods declared in an abstract class, without providing an implementation
- Use of the @abstractmethod decorator to declare abstract methods

ABSTRACT CLASS VS. CONCRETE CLASS

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

obj = Shape() # Results in Error
obj.calculate_area() # Results in Error
```

- Cannot be instantiated directly. It exists to be subclassed
- Its main purpose is to serve as a blueprint for other classes

ABSTRACT CLASS VS. CONCRETE CLASS

```
class Rectangle():
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

rectangle = Rectangle(width=5, height=10)
print("Rectangle Area:", rectangle.calculate_area())
```

- Can be instantiated directly and used to create objects
- Inheritance: provides implementations for all the abstract methods defined in its abstract parent class

ABSTRACT CLASS INHERITANCE

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass
```

```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius * self.radius

circle = Circle(radius=7)
print("Circle Area:", circle.calculate_area())
```

- Why? **BLUEPRINT!!!**

ABSTRACT METHODS AND POLYMORPHISM

- Abstract methods enable dynamic binding, where the appropriate method implementation is selected at runtime based on the actual type of the object
- Each subclass can provide its own unique implementation for the abstract method while adhering to the contract specified by the abstract class
- The ability to have different implementations of the same method across various subclasses results in runtime polymorphism. This means that the appropriate method is determined at runtime, providing flexibility and extensibility in the code

EXERCISE

- Try the following exercises

- 02-OO
  > 07-abstract-methods

Abstract Properties

- Abstract Properties
- Use Cases for Abstract Properties

ABSTRACT PROPERTIES

Python 2:

```
from abc import ABC, abstractproperty

class Shape(ABC):
    @abstractproperty
    def color(self):
        pass
```

Python 3.3+:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @property
    @abstractmethod
    def color(self):
        pass
```

- Abstract properties are attributes that are declared but not implemented in an abstract class
- Order matters, you have to use @property above @abstractmethod

GETTER METHOD FOR ABSTRACT PROPERTY

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @property
    @abstractmethod
    def color(self):
        pass
```

- Read-only abstract properties by omitting the setter method
- Order matters, you have to use @property above @abstractmethod

SETTER METHOD FOR ABSTRACT PROPERTY

```
from abc import ABC, abstractmethod

class Shape(ABC):
    ...

    @color.setter
    @abstractmethod
    def color(self, value):
        pass
```

- Order matters, you have to use @attribute.setter above @abstractmethod

ABSTRACT PROPERTY IN CONCRETE SUBCLASS

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @property
    @abstractmethod
    def color(self):
        pass

    @color.setter
    @abstractmethod
    def color(self, value):
        pass

    @abstractmethod
    def calculate_area(self):
        pass
```

```python
class Rectangle(Shape):
    def __init__(self, width, height, color):
        self.width = width
        self.height = height
        self.color = color

    @property
    def color(self):
        return self._color

    @color.setter
    def color(self, value):
        if not isinstance(value, str):
            raise ValueError("Color must be a string")
        self._color = value

    def calculate_area(self):
        return self.width * self.height
```

```python
rectangle = Rectangle(width=5, height=10, color="Blue")

print("Rectangle Area:", rectangle.calculate_area())
print("Rectangle Color:", rectangle.color)

rectangle.color = 42 # Resutls in error due to color validation
```

- Validation is done in the implementation of the abstract setter property
- In Documentation you might encounter @parent.attribute.setter, this only works if you have an actual implementation (concrete getter) in the parent class, not an abstract one

USE CASES FOR ABSTRACT PROPERTIES

- Enforce presence and ensure implementations for attributes by concrete subclasses
- Common interface-like structure across multiple classes
- Control access level in subclasses by creating only readable or writable properties
- Subclasses can implement the same abstract property in their own way, polymorphism
- Provide hooks for users to extend or customize functionality when designing a framework
- …

Revision

- Abstract properties vs. abstract methods

ABSTRACT PROPERTIES VS. ABSTRACT METHODS

Properties

- Defining attributes without providing the implementation for their getter and setter methods
- Use the @property decorator for the getter and @attribute.setter for the setter, on top of the @abstractmethod decorator
- Want to ensure that all subclasses provide a specific set of attributes with getter and setter methods

Methods

- Defining a method signature without providing any implementation
- Use the @abstractmethod decorator directly on the method
- Want to ensure that all subclasses provide a specific method implementation

EXERCISE

- Try the following exercises

- 02-OO
  > 08-abstract-properties