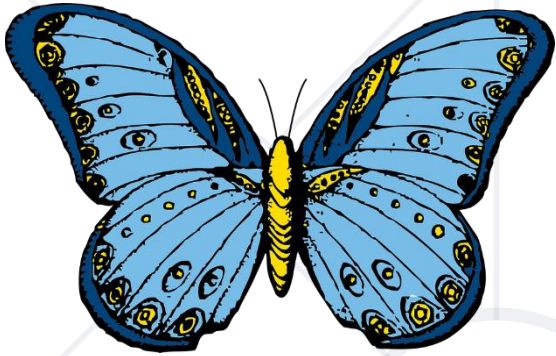


# Polymorphism and Abstraction

Having Multiple Forms



**SoftUni Team**  
Technical Trainers



**SoftUni**



Software University

<https://softuni.bg>

**sli.do**

**#python-advanced**

1. What is a Polymorphism?
2. Overloading Built-in Methods
3. Duck Typing
4. What is an Abstraction?
  - **abc** module
  - **@abstractmethod**





# **What is a Polymorphism**

Definition and Examples

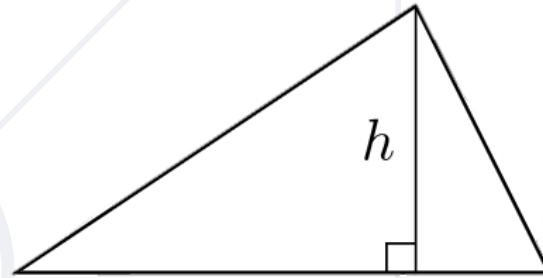
# Polymorphism Definition

- Polymorphism is based on the Greek words "poly" (**many**) and "morphism" (**forms**)
- It is the ability to present the **same interface** for **differing underlying forms** through the interface of **their base class**
- e. g., **Square**, and **Triangle inherit Shape**, so their instances can be used from an instance of type Shape

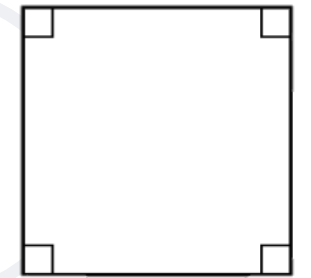


# Run-Time Polymorphism

- A subclass can **override** a method of its superclass
- e. g., **both** triangle and square are **shapes** and have **area**



$$A = \frac{b \cdot h}{2}$$



$$A = s^2$$

# Example: Run-Time Polymorphism

```
class Shape:
    def calculate_area(self):
        return None

class Square(Shape):
    side_length = 2
    def calculate_area(self):
        return self.side_length * 2

class Triangle(Shape):
    base_length = 4
    height = 3
    def calculate_area(self):
        return 0.5 * self.base_length * self.height
```

Overriding  
calculate\_area  
method

# Without Polymorphism

- A **type check** may be required before performing an action on an object to **determine the correct method** to call


```
for obj in Square(), Triangle():  
    if isinstance(obj, Square):  
        area = obj.calculate_square_area()  
    elif isinstance(obj, Triangle):  
        area = obj.calculate_triangle_area()  
    print(area)
```

If not overriding  
calculate\_area  
method





# Compile-Time Polymorphism

- 
- Python **does not** support compile-time polymorphism or **method overloading**
  - If a class has multiple methods **with the same name**, the method defined in the **last will override** the earlier one

```
class Person:  
    def say_hello():  
        return "Hi!"  
    def say_hello():  
        return "Hello"  
  
print(Person.say_hello()) # Hello
```

- **Refactor** the provided code, so we do not need to do **any type-checking**. The classes should implement the method, so it returns the number of sensors for **each type** of robot.



```
class MedicalRobot(Robot):  
    @staticmethod  
    def sensors_amount():  
        return 6
```

```
class ChefRobot(Robot):  
    @staticmethod  
    def sensors_amount():  
        return 4
```

```
class WarRobot(Robot):  
    @staticmethod  
    def sensors_amount():  
        return 12
```

```
def number_of_robot_sensors(robot):  
    print(robot.sensors_amount())
```

```
basic_robot = Robot('Robo')  
da_vinci = MedicalRobot('Da Vinci')  
moley = ChefRobot('Moley')  
griffin = WarRobot('Griffin')
```

```
number_of_robot_sensors(basic_robot)  
number_of_robot_sensors(da_vinci)  
number_of_robot_sensors(moley)  
number_of_robot_sensors(griffin)
```



# **Overloading Built-in Methods**

# Overloading Built-in Methods

- **Change the behavior** of functions such as `len`, `abs`, `str`, `repr`, and so on
- To do this, you only need to **define** the corresponding **special method in your class**



```
class MyClass:
    def __init__(self, name, size):
        self.name = name
        self.size = size

    def __len__(self):
        return self.size

my_class = MyClass("Class Name", 3)
print(len(my_class)) # 3
```

# Operator Overloading

- An operator behaves **differently** based on the **type of the operands**
  - e.g., operator "+" is used to add two **integers** as well as join two **strings** and merge two **lists**
  - It is **overloaded** by **int** class, **str** class and **list** class

```
integer = 1 + 1  
string = "Hello, " + "SoftUni"  
list = ["1", "2"] + ["3", "4"]
```



# Operator Magic Methods



Magic Methods	Get Called Using
<code>__add__(self, other)</code>	<code>+</code>
<code>__sub__(self, other)</code>	<code>-</code>
<code>__mul__(self, other)</code>	<code>*</code>
<code>__floordiv__(self, other)</code>	<code>//</code>
<code>__truediv__(self, other)</code>	<code>/</code>
<code>__pow__(self, other[, modulo])</code>	<code>**</code>

# Example: Overloading `__add__()`

- If we have a **class Purchase** and we want to **sum** all expenses using the **+** operator, we can override the **`__add__`** method

```
class Purchase:
    def __init__(self, product_name, cost):
        self.product_name = product_name
        self.cost = cost

    def __add__(self, other):
        name = f'{self.product_name}, {other.product_name}'
        cost = self.cost + other.cost
        return Purchase(name, cost)

first_purchase = Purchase('sofa', 650)
second_purchase = Purchase('table', 150)
print(first_purchase + second_purchase)  # sofa, table; 800
```



# "Rich Comparison" Magic Methods



Magic Methods	Get Called Using
<code>__lt__(self, other)</code>	<code>&lt;</code>
<code>__le__(self, other)</code>	<code>&lt;=</code>
<code>__eq__(self, other)</code>	<code>==</code>
<code>__ne__(self, other)</code>	<code>!=</code>
<code>__gt__(self, other)</code>	<code>&gt;</code>
<code>__ge__(self, other)</code>	<code>&gt;=</code>

# Example: Overloading `__gt__()`

- If we have a **class Person** and we want to **compare** them by their salary using the `>` operator, we can override the **`__gt__`** method

```
class Person:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def __gt__(self, other):
        return self.salary > other.salary

person_one = Person('John', 20)
person_two = Person('Natasha', 36)
print(person_one > person_two)  # False
```

# Problem: ImageArea

- Create a class called **ImageArea**
- It stores the **width** and the **height** of an image
- Create a method called **get\_area()** which returns the area of the image
- Implement all the **magic methods** for **comparison** of two image areas (**>**, **>=**, **<**, **<=**, **==**, **!=**) which will compare their areas

# Solution: ImageArea

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

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

    def __eq__(self, other):
        return self.get_area() == other.get_area()

# TODO: Implement the other comparison methods
```



**Duck Typing**

# Duck Typing Definition

- Duck Typing is a **type system** used in dynamic languages
- "If it **looks like a duck** and **quacks like a duck**, it's **a duck**"
  - i.e., we don't care about **objects' types**, but whether **they have the methods** we need



- We **can create a method** that calls the sound method, **no matter** what the object which makes the sound is

```
class Cat:
    def sound(self):
        print("Meow!")

class Train:
    def sound(self):
        print("Sound from wheels slipping!")

for any_type in Cat(), Train():
    any_type.sound()
```

Works for both  
classes

- Create a method called **start\_playing** which will receive an instance and will print its **play()** method if it has one

## *# Test Code*

```
class Guitar:
    def play(self):
        return "Playing the guitar"
```

```
guitar = Guitar()
start_playing(guitar)
# Playing the guitar
```

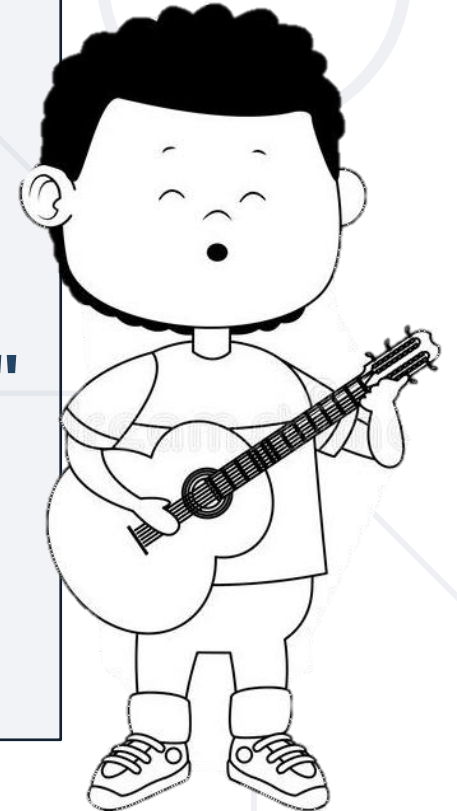
## *# Test Code*

```
class Children:
    def play(self):
        return "Children are playing"
```

```
piano = Children()
start_playing(piano)
# Children are playing
```



```
def start_playing(obj):  
    return obj.play()  
  
# Test Code  
class Guitar:  
    def play(self):  
        return "Playing the guitar"  
  
guitar = Guitar()  
start_playing(guitar)
```





# **What is an Abstraction**

Definition and Examples

# A Word about Abstraction



- In object-oriented programming, abstraction is one of the **four central principles**
- Through abstraction, we hide all but the relevant data about an object to **reduce complexity** and **increase efficiency**
- Abstraction can be achieved by:
  - Functions and methods
  - **Abstract classes**

- Abstract classes are classes that contain one or more **abstract methods**
  - An abstract method is a method that is **declared** but contains **no implementation**
- Abstract classes may not be instantiated and require **subclasses** to provide implementations for the abstract methods

- It could be achieved using **exceptions**, but it **not a good practice**

```
class Shape:
    def __init__(self):
        if type(self) is Shape:
            raise Exception('This is an abstract class')

    def area(self):
        raise Exception('This is an abstract class')

    def perimeter(self):
        raise Exception('This is an abstract class')
```

- Abstract base classes (ABCs) **enforce** derived classes to implement particular methods from the base class
  - We implement it using the **abc** module

```
class Shape:
    def __init__(self):
        if type(self) == Shape:
            raise Exception('...')
    def area(self):
        raise Exception('...')
    def perimeter(self):
        raise Exception('...')
```



```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimeter(self):
        pass
```

# Example: Abstract classes

```
from abc import ABC, abstractmethod
```

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

```
@abstractmethod
```

```
def sound(self):  
    raise NotImplementedError("Subclass must implement")
```

```
# Continues on next slide
```

Defining an Abstract Class

Makes a Method Abstract

# Example: Abstract classes

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

Inherit the  
Abstract Class

```
    def sound(self):  
        print("Bark!")
```

Implement the  
Abstract method

```
class Cat(Animal):  
    def __init__(self, name):  
        super().__init__(name)  
  
    def sound(self):  
        print("Meow!")
```

```
cat = Cat("Willy")  
cat.sound()  
dog = Dog("Willy")  
dog.sound()  
animal = Animal("Willy")  
animal.sound()  
# Meow!  
# Bark!  
# Error!
```



- Create an abstract class **Shape** with abstract methods **calculate\_area** and **calculate\_perimeter**
- Create classes that implement the methods:
  - **Circle** - receives **radius** upon initialization
  - **Rectangle** - receives **height** and **width** upon initialization
  - The **fields** of Circle and Rectangle should be **private**

```
from abc import ABC, abstractmethod
from math import pi

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

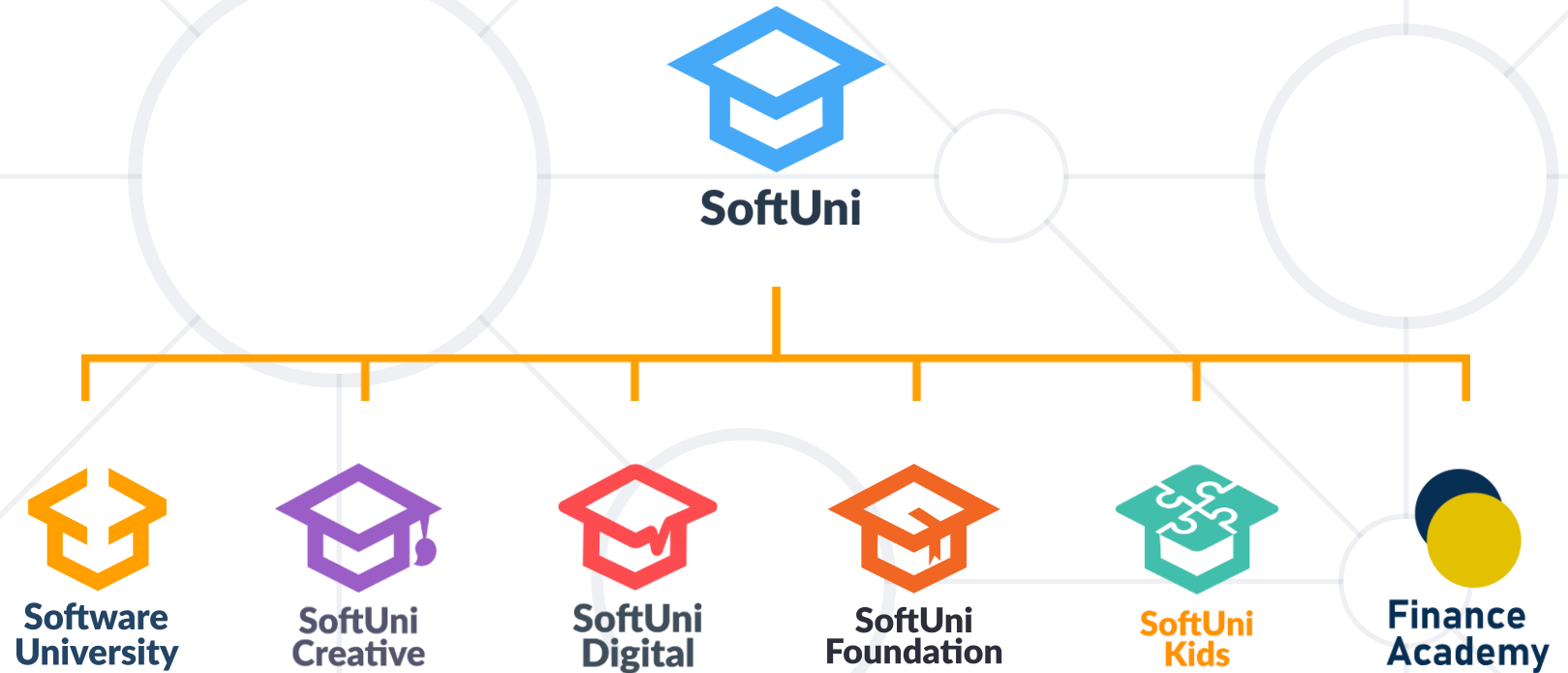
    @abstractmethod
    def calculate_area(self):
        pass

# TODO: Implement Circle and Rectangle
```

- **Polymorphism** means the same function name is used for different types
- Through **abstraction**, we hide all but the relevant data about an object
- Abstract classes may **not be instantiated**, and require subclasses



# Questions?



# SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [softuni.org](http://softuni.org)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)



- This course (slides, examples, demos, exercises, homework, documents, videos, and other assets) is **copyrighted content**
- Unauthorized copy, reproduction, or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>

