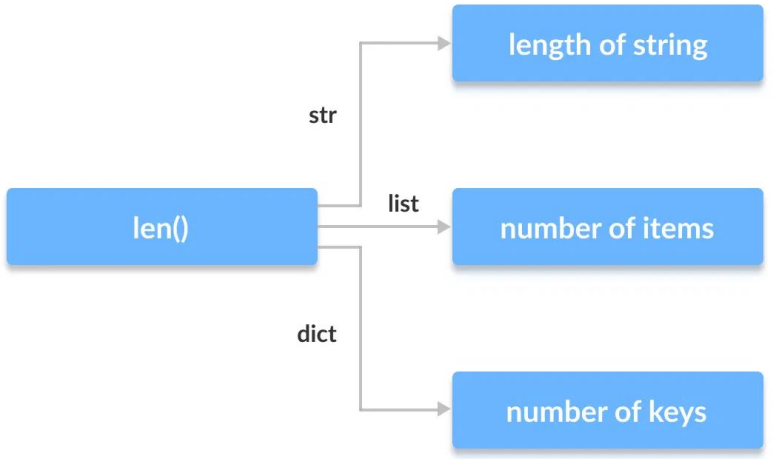
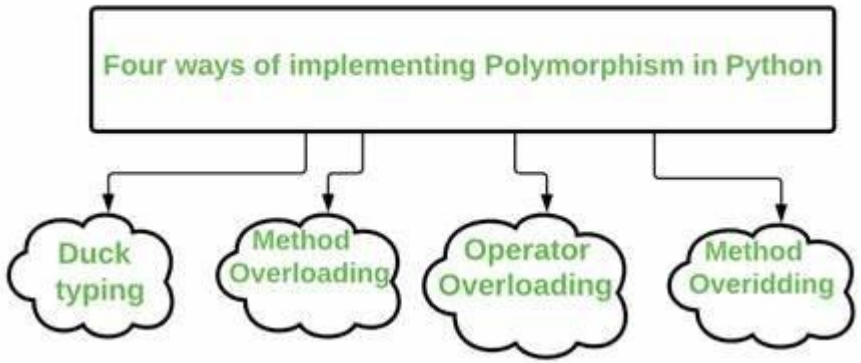
Object-Oriented Programming (OOP)

Polymorphism - Poly means “many” morph means “forms”

- Polymorphism represents the ability of different objects to respond to the same method call.
- It allows for flexible code design, by creating methods that can work with multiple object types.
- It allows us to write classes with methods that operate on abstractions, which promotes code reuse for specific object types.
- It is closely tied to other OOP concepts like inheritance, encapsulation, and abstraction.

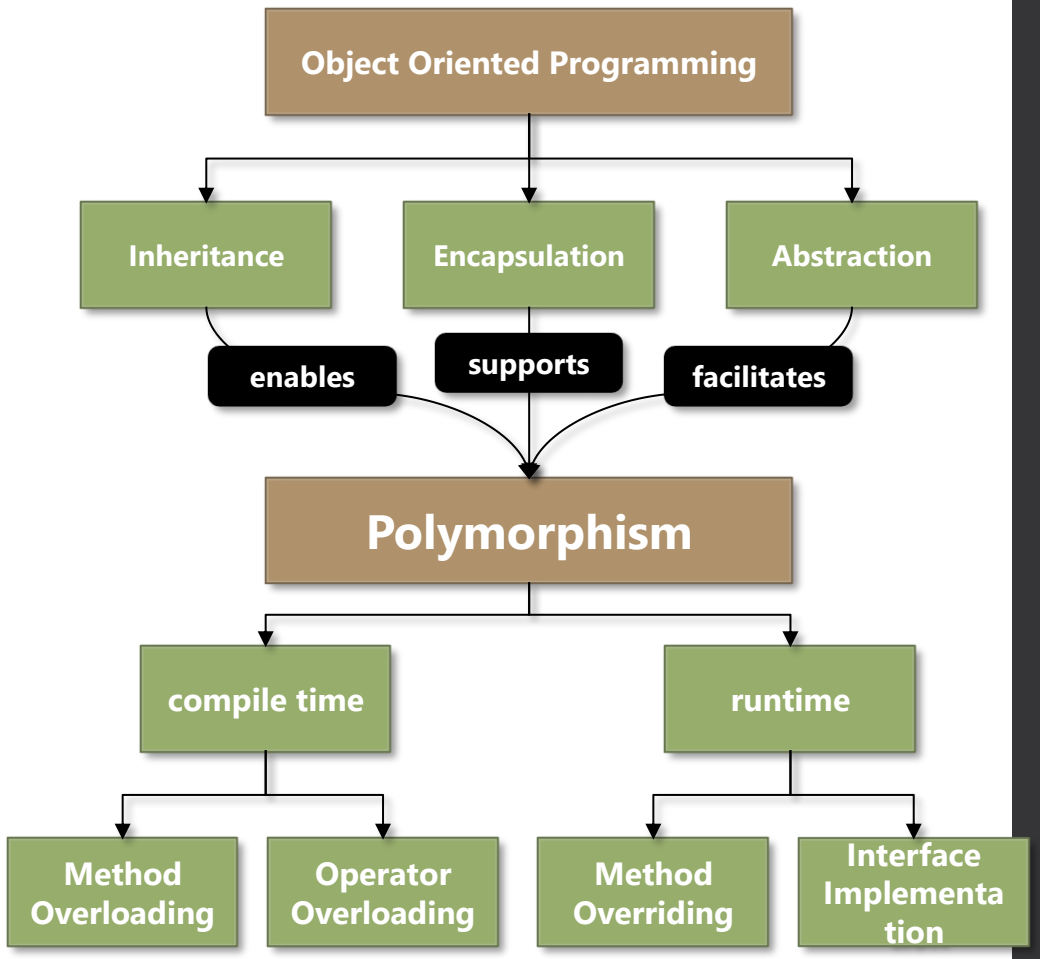


len() function *returns different things based on its arguments*

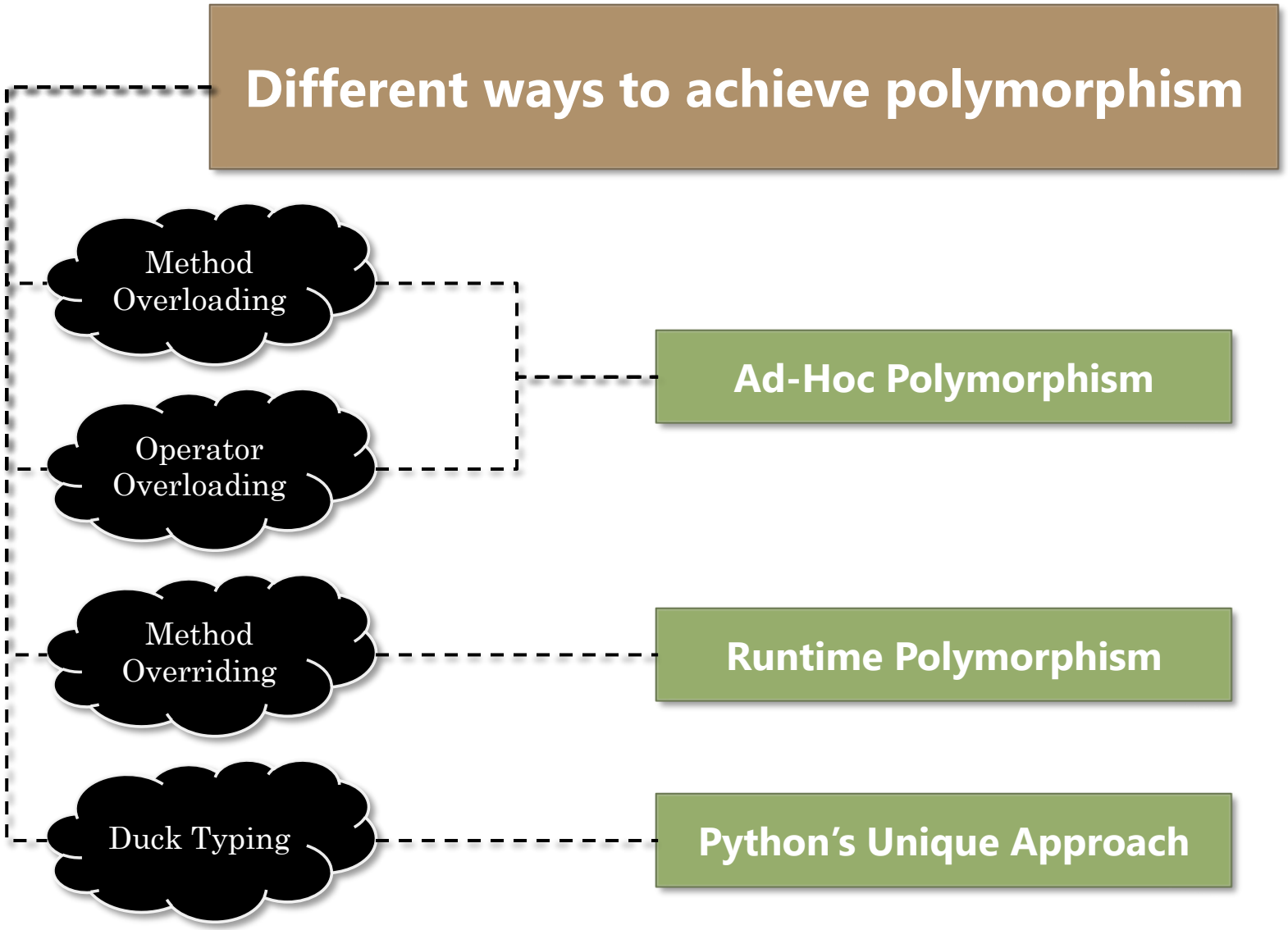


Polymorphism – One of the Pillars of OOP

- **OOP** provides the concepts of classes and objects that make polymorphism possible.
- **Inheritance** enables polymorphism where base classes keep their own implementation of methods while derived classes override them.
- **Encapsulation** supports polymorphism by hiding the implementation details and providing same interface for different classes.
- **Abstraction** facilitates polymorphism by defining abstract classes and interfaces that multiple classes must implement for different behaviors.



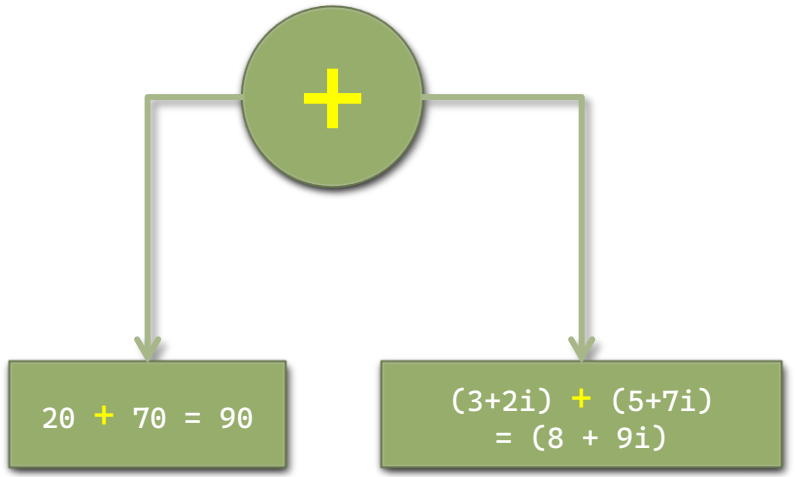
Polymorphism in Python



Types of Polymorphism in Python

Ad-hoc Polymorphism – Operator/function overloading

- This type of polymorphism allow functions or operators to behave differently based on the type or number of arguments.
- It is implemented through the use of dunder methods like, **`__add__`**, **`__sub__`**, **`__str__`**, **`__getitem__`**, etc.
- This allows custom objects to behave like built-in types.
- For example, '+' operator adds numbers and concatenates strings and lists. '*' operator multiplies numbers and repeats strings and lists.

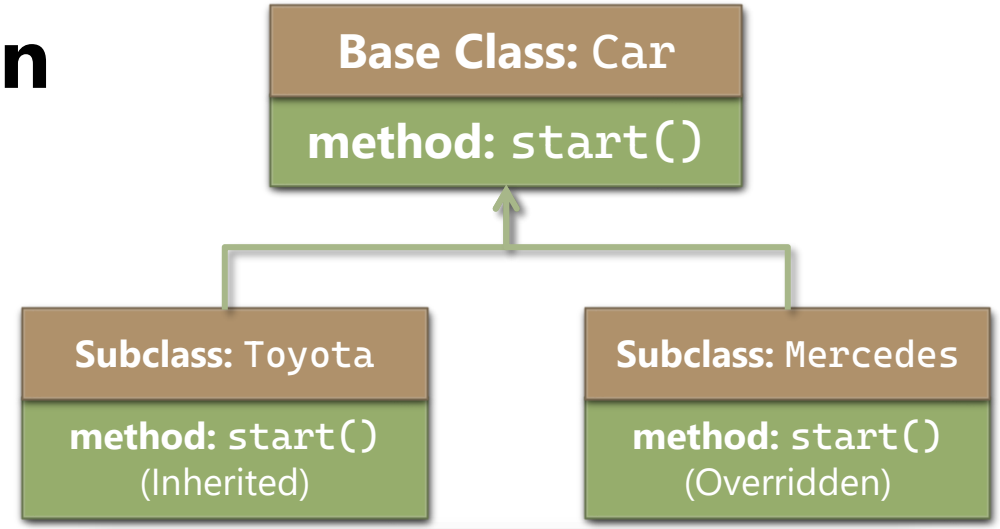



```python
class Complex:
    """
    A class to represent complex numbers
    """
    def __init__(self, real, imag):
        self.real = real # real part
        self.imag = imag # imaginary part

    def __add__(self, other):
        """
        Overlaoding this method to support addition of
        two complex numbers using + operator
        """
        if isinstance(other, Complex):
            return Complex(self.real + other.real,
                           self.imag + other.imag)
        elif isinstance(other, (int, float)):
            return Complex(self.real + other, self.imag)
```

Types of Polymorphism in Python

Runtime Polymorphism – Subtype Polymorphism

- Objects of different classes related by inheritance respond to the same method call.
- Code that works with a superclass also works with all its subclasses.
- Python automatically calls the correct method based on the object's type through method overriding.
- Abstract Base Classes (ABC) define interfaces for subclasses to implement.

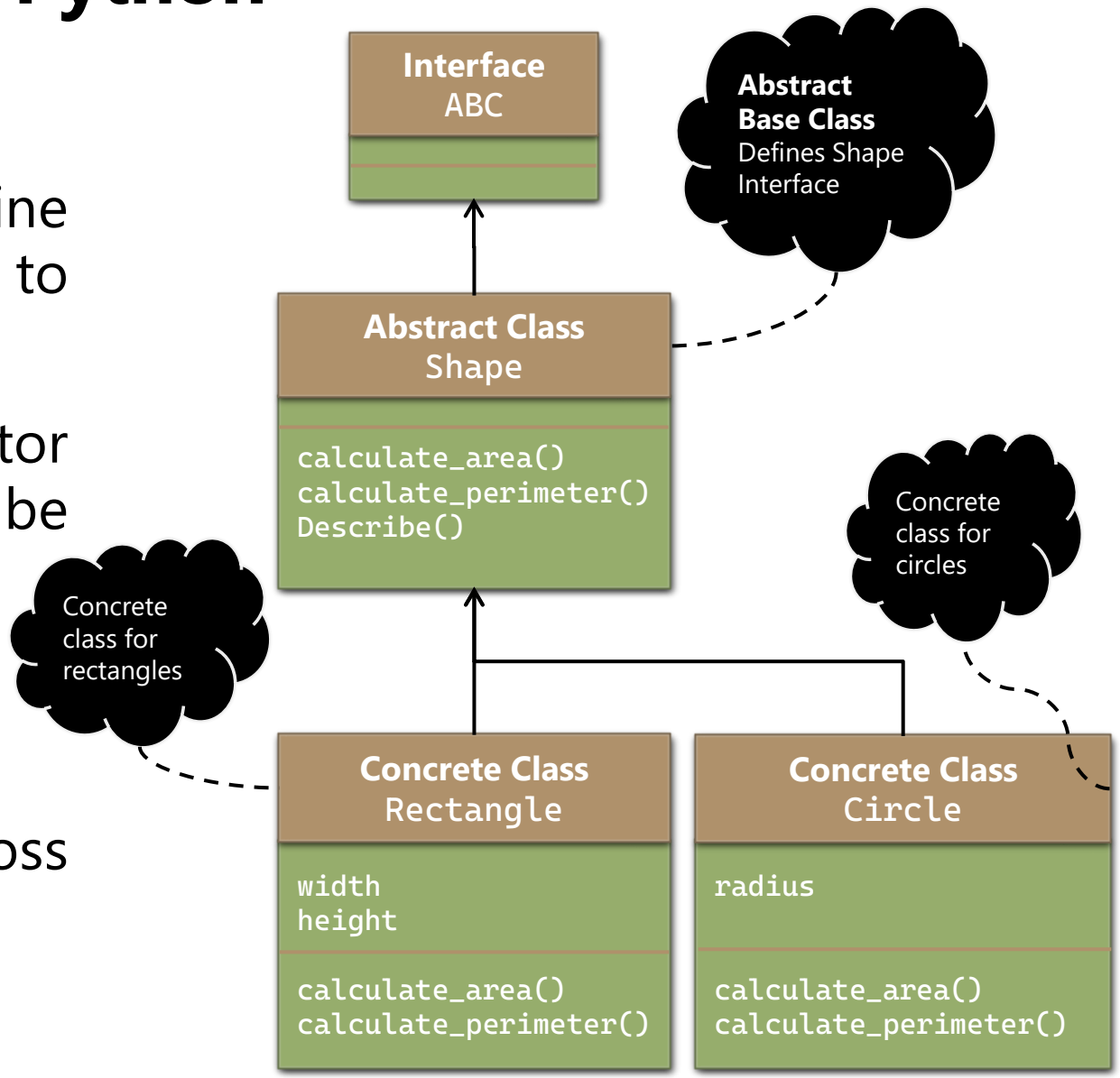



```python
class Car:
    def start(self):
        print("Car has started...")

class Toyota(Car):
    pass

class Mercedes(Car):
    def start(self):
        print("Mercedes has started...")
# example usage
car = Car()
toyota = Toyota()
mercedes = Mercedes()

car.start()       # Car has started...
toyota.start()    # Car has started...
mercedes.start()  # Mercedes has started...
```

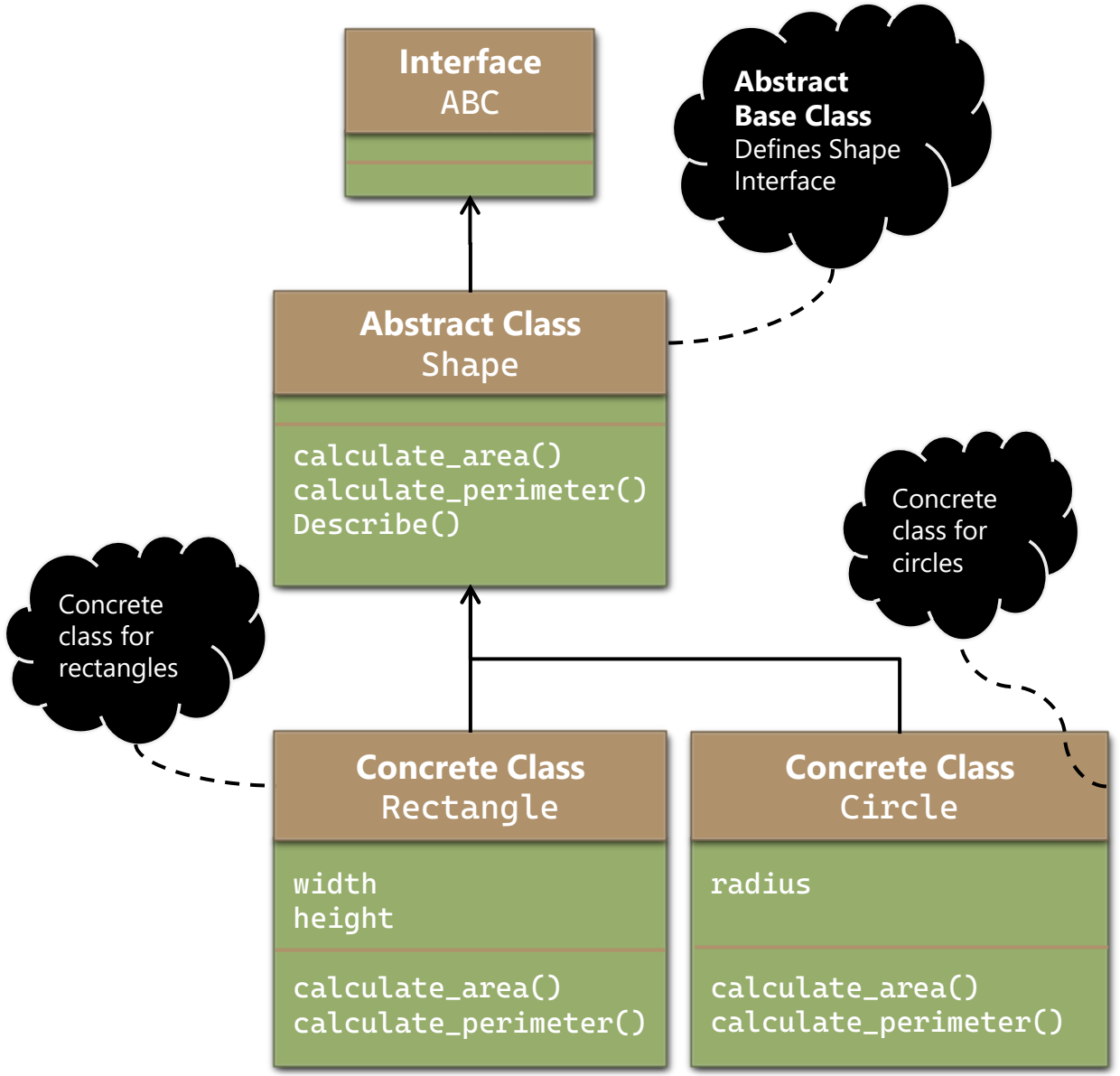
Types of Polymorphism in Python

Abstract Base Classes - ABC

- Abstract Base Classes (ABC) define interfaces for subclasses to implement.
- Use `@abstractmethod` decorator to declare methods that must be overridden.
- Cannot instantiate ABC directly must create concrete subclasses.
- Ensures consistent behavior across different implementations.



```python
from abc import ABC, abstractmethod

# Abstract base class defining the interface
class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        """Calculate the area of the shape."""
        pass

    @abstractmethod
    def calculate_perimeter(self):
        """Calculate the perimeter of the shape."""
        pass

    def describe(self):
        """Non-abstract method that can be inherited as-is."""
        return f"Shape area {self.calculate_area()} and " \
                f"Shape Perimeter {self.calculate_perimeter()}"

# Concrete Classes
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

    def calculate_perimeter(self):
        return 2 * (self.width + self.height)

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

    def calculate_area(self):
        return 3.14 * self.radius**2

    def calculate_perimeter(self):
        return 2 * 3.14 * self.radius
```

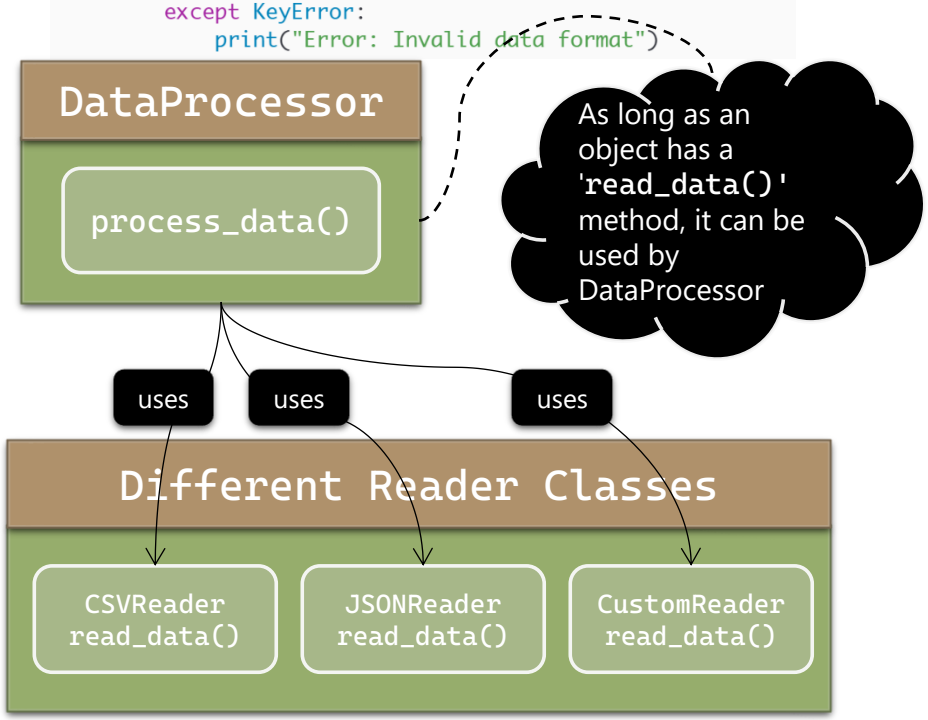


Types of Polymorphism in Python

Duck Typing – Python's Unique Approach

- This type of polymorphism focus on object's behavior(method) rather than its type or class.
- Objects are compatible if they support the same methods being used.
- No need for explicit interface declarations or inheritance.
- It reduces the need for creating complex class hierarchies.

```python
# Three different classes, no common base class
class CSVReader:
    def read_data(self, file_path):
        print(f"Reading CSV file: {file_path}")
        return [{"name": "Alice", "age": 30},
                {"name": "Bob", "age": 25}]

class JSONReader:
    def read_data(self, file_path):
        print(f"Reading JSON file: {file_path}")
        return [{"name": "Charlie", "age": 35},
                {"name": "David", "age": 28}]

# A class that processes data from different sources
class DataProcessor:
    def process_data(self, reader, source):
        try:
            data = reader.read_data(source)
            total_age = sum(item["age"] for item in data)
            average_age = total_age / len(data)
            print(f"Average age: {average_age:.2f}")
        except AttributeError:
            print("Error: Incompatible reader object")
        except KeyError:
            print("Error: Invalid data format")
```



```python
# Three different classes, no common base class
class CSVReader:
    def read_data(self, file_path):
        print(f"Reading CSV file: {file_path}")
        return [{"name": "Alice", "age": 30},
                {"name": "Bob", "age": 25}]

class JSONReader:
    def read_data(self, file_path):
        print(f"Reading JSON file: {file_path}")
        return [{"name": "Charlie", "age": 35},
                {"name": "David", "age": 28}]

# A class that processes data from different sources
class DataProcessor:
    def process_data(self, reader, source):
        try:
            data = reader.read_data(source)
            total_age = sum(item["age"] for item in data)
            average_age = total_age / len(data)
            print(f"Average age: {average_age:.2f}")
        except AttributeError:
            print("Error: Incompatible reader object")
        except KeyError:
            print("Error: Invalid data format")

# Usage
csv_reader = CSVReader()
json_reader = JSONReader()
processor = DataProcessor()
processor.process_data(csv_reader, "data.csv")
processor.process_data(json_reader, "data.json")
```

Duck Typing – Python's Unique Approach