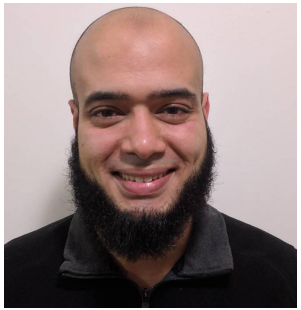
Python Programming

Polymorphism

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



What is Polymorphism?

- Polymorphism = many forms
- The ability to use Functions/Methods/Operators/Objects with different types and potentially behave differently
    - Think + operators **with numbers** sum them, but **with strings** concatenate them (different behaviour)
- In languages like C++, the concept appears strongly when explaining!
- But python is polymorphic by design!

```
# len functions receives many types(forms)
print(len([1, 2, 3]))
print(len(((6, 7), 'hey')))
print(len("mostafa"))

# + * operators can be used with several types
# But behave differently!
print(2 + 3 * 4)
print('Most' + ' Saad' * 4)
```

Duck Typing

- If it walks like a duck, and it quacks like a duck, then it must be a duck
  - Methods matters much more than the object type!
    - Does it support the requested behaviour?
  - To call len(something), object needs to define **__len__**, Regardless the object type!
- Observe: Polymorphism even doesn't need inheritance to exist!

```
2   class Car:
3       def get_name(self):
4           return 'BMW'
5
6   class Person:
7       def get_name(self):
8           return 'Mostafa'
9
10  class Home:
11      pass
12
13  def process(obj):
14      # any object that has
15      # get_name method is good
16      print(obj.get_name())
17
18  process(Car())
19  process(Person())
20
21  # AttributeError: 'Home' object
22  #  has no attribute 'get_name'
23  #process(Home())
```

Shape Example

```
class Shape:
    def __init__(self, name):
        super().__init__()
        self.name = name

    @property
    def area(self):
        raise NotImplementedError
```

```
class Rectangle(Shape):
    def __init__(self, name, wid, height):
        super().__init__(name)
        self.wid = wid
        self.height = height

    @property
    def area(self):
        return self.wid * self.height
```

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

    @property
    def area(self):
        from math import pi
        return 2 * pi * self.radius
```

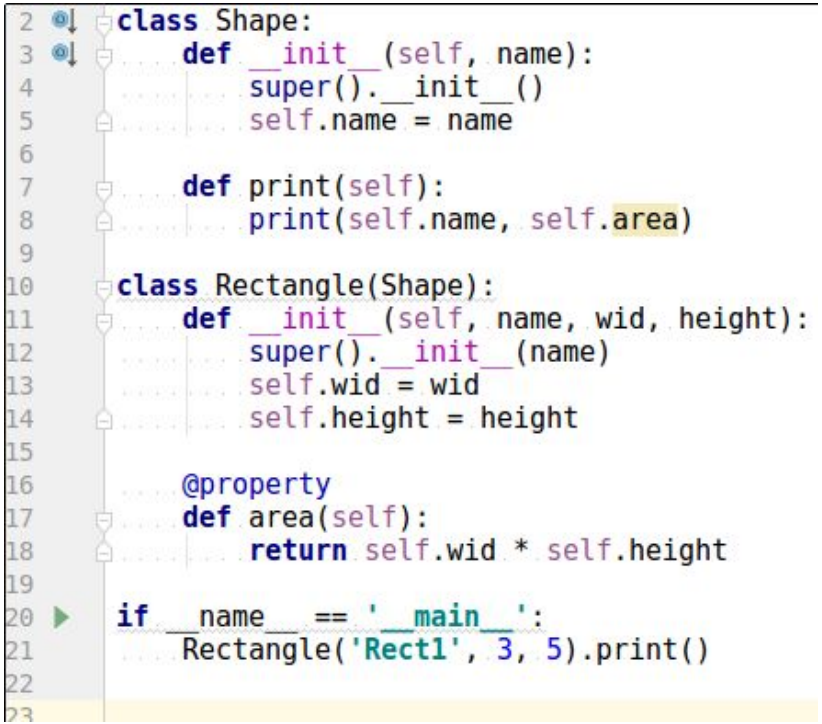
Shape Example

```
31  class Editor:
32      def __init__(self):
33          self.shapes = []
34
35      def process(self):
36          area_sum = 0
37          for shape in self.shapes:
38              print(shape.name, shape.area)
39              area_sum += shape.area
40          return area_sum
41
42  if __name__ == '__main__':
43      editor = Editor()
44      editor.shapes.append(Rectangle('Rect1', 3, 5))
45      editor.shapes.append(Circle('MyCirc', 2))
46      editor.shapes.append(Rectangle('Rect2', 10, 2))
47      print(f'area sum = {editor.process()}')
```

```
Rect1 15
MyCirc 12.566370614359172
Rect2 20
area sum = 47.56637061435917
```

Inverse of control

- Normal flow: child class knows parent class. But what if parent class is waiting for something from children?
- Method print is calling area property
- This is a case where **high-level class** is calling **low-level class**
- Core step in frameworks named **Inverse of control**



```
class Shape:
    def __init__(self, name):
        super().__init__()
        self.name = name

    def print(self):
        print(self.name, self.area)

class Rectangle(Shape):
    def __init__(self, name, wid, height):
        super().__init__(name)
        self.wid = wid
        self.height = height

    @property
    def area(self):
        return self.wid * self.height

if __name__ == '__main__':
    Rectangle('Rect1', 3, 5).print()
```

Inverse of control: Example

```
3  class GraphAlgorithm:
4      def __init__(self):
5          self.algorithms_steps = [self.step1_general,
6                                   self.step2_abstract,
7                                   self.step3_general]
8
9      def run(self):
10         return ''.join([step() for step in self.algorithms_steps])
11
12     def step1_general(self):
13         return 'G1'
14
15     def step3_general(self):
16         return 'G3'
17
18 class Dijkstra(GraphAlgorithm):
19     def __init__(self):
20         super().__init__()
21
22     def step2_abstract(self):
23         return 'APQ'
24
25 print(Dijkstra().run())
```

Composition over Inheritance

- In inheritance: we highlighted several issues in (multiple) inheritance
- Composition is the way to go
  - You avoid the “combination hell” of inheritance (RobotDog)
  - Built-in polymorphism (duck typing) make things intuitive
- Design patterns are typically needed
  - Delegate pattern to switch B inherits A to ⇒ B has delegate of A
  - Switch template pattern to strategy pattern
  - Decorator pattern to avoid the Exploding class hierarchy
- Cons
  - More code to do delegation of calls on composed objects
    - You may need to wrap/delegate some classes to remove irrelevant functions
  - More code to create instances of intermediate classes ⇒ use Factory Pattern

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."