Info1 Python Tutorial - Part 5

Alex Wolf

October 24, 2023

1 Info 1: Python Tutorial - Part 5

2 Quick recap

2.1 Scope

Scope determines where in a program a given variable or function can be accessed and modified.

For example, `y` is *in scope* both inside and outside of the `power2` function:

[1]:
```
y = 10
def power2(y):
    y = 6           # reassigning y just overwrites the y passed to the␣
 ↪function on line 2!
    print(y)        # value of y which was assigned on line 3
    return y ** 2
print(y)            # still the value of y defined on line 1, reassignment␣
 ↪inside power2 had no impact on this!
print(power2(4))
print(y)
print(power2(4))
```

```
10
6
36
10
6
36
```

2.2 Class

A class defines the attributes and behavior of a thing. It is a blueprint for creating objects and encapsulates data for the objects and methods to manipulate that data.

2.2.1 Python example:

[2]:
```
class MyClass:
    def __init__(self, param):
        self.param = param
```

```
    def my_method(self):
        print("This is a method")
```

`__init__` is the constructor in python

`MyClass` is the identifier of the class

`my_method` is a method defined inside the class

2.2.2 Creating objects:

Once a class is defined, you can create objects (instances) of that class:

[3]:
```
obj = MyClass("Hello")
obj.my_method()
print(obj.param)
```

```
This is a method
Hello
```

3 Working with multiple classes

Your task is to design a system to handle various geometric shapes. This system must enable precise area calculations for each shape and facilitate their visualization on a canvas using specified x- and y-coordinates. Furthermore, the system should also incorporate functionality for coloring the shapes.

Specifications:

Area calculation

Drawing

Coloring

[4]:
```
from math import pi

class Circle:
    def __init__(self, cord_x, cord_y, color, radius):
        self.x = cord_x
        self.y = cord_y
        self.color = color
        self.radius = radius

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

class Rectangle:
    def __init__(self, cord_x, cord_y, color, width, length):
        self.x = cord_x
```

```
        self.y = cord_y
        self.color = color
        self.width = width
        self.length = length

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


class Triangle:
    def __init__(self, cord_x, cord_y, color, base, height):
        self.x = cord_x
        self.y = cord_y
        self.color = color
        self.base = base
        self.height = height

    def calculate_area(self):
        return 0.5*self.base*self.height
```

[5]:
```
circle = Circle(10, 10, "blue", 1.784)
rectangle = Rectangle(15, 10, "red", 5 , 8)
triangle = Triangle(20, 10, "green", 4, 3)
shapes = [circle, rectangle, triangle]
[ x.calculate_area() for x in shapes]
```

[5]:
```
[9.998608708503477, 40, 6.0]
```

4 The Problem

In all classes, the fundamental task involves calculating the area of a shape and recording its x and y coordinates for future rendering on a canvas. However, a glaring issue arises: redundancy plagues our code. The same logic is duplicated across multiple classes, creating an impractical and error-prone system. Any necessary changes mandate alterations in multiple places, leading to inefficiency and maintenance nightmares.

This lack of a unified foundation for all shapes poses several pressing problems:

Code Redundancy: Each of the three shape classes redundantly houses identical draw methods and variables. This repetition not only wastes precious development time but also increases the likelihood of inconsistencies.

Difficulty in Maintenance: When modifications are necessary, especially in the draw method, the task becomes formidable. With three classes to edit, the probability of overlooking a crucial detail skyrockets, making maintenance a cumbersome challenge.

Lack of Code Reusability: Regrettably, our current design lacks code reusability. The valuable logic responsible for drawing and area calculations is trapped within individual silos, preventing efficient

utilization across various shapes. This missed opportunity for code reuse hampers our productivity and limits the system's flexibility.

Difficulty in External Use: Beyond our immediate development environment, the challenges persist. External users cannot confidently assume uniformity across the shape classes. Each class possesses its unique methods and attributes, demanding meticulous scrutiny. This lack of consistency complicates the integration of our shapes into external applications, hindering seamless collaboration and interoperability.

To address these issues and foster a more robust, efficient, and maintainable system, we must transition to a unified approach. By establishing a common foundation shared by all shapes, we can eliminate redundancy, simplify maintenance, enhance code reusability, and facilitate external use. Embracing a standardized structure will not only streamline our current development efforts but also pave the way for future scalability and collaboration.

5 The Solution - Inheritance

Inheritance is one of the fundamental pillars of OOP. It allows you to create a new class (the child class or subclass) that inherits properties and behaviors from an existing class (the parent class or superclass). This promotes code reuse and maintains a clear hierarchy of objects.

In Python, inheritance enables you to create a new class based on an existing class, inheriting its attributes and methods, and allowing you to override or extend them as needed.

We will go through the following example for the basic syntax:

[6]:
```
class Shape:
    def __init__(self, cord_x, cord_y, color):
        self.cord_x = cord_x
        self.cord_y = cord_y
        self.color = color

    def calculate_area(self):
        pass


class Circle(Shape):
    def __init__(self, cord_x, cord_y, color, radius):
        super().__init__(cord_x, cord_y, color)
        self.radius = radius

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

class Rectangle(Shape):
    def __init__(self, cord_x, cord_y, color, width, length):
        super().__init__(cord_x, cord_y, color)
        self.width = width
        self.length=length
```

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

[7]:
```
class Parent:
    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2

    def cool_function(self):
        print(f"Attribute1: {self.attr1}\nAttribute2: {self.attr2}")


class Child(Parent):
    def __init__(self, attr1, attr2, attr3):
        super().__init__(attr1, attr2)
        self.attr3 = attr3
```

Parent is the existing class, and Child is the new class that inherits from Parent.

Child is declared with the name of the parent-/superclass inside parentheses, indicating that Child inherits from Parent

This allows us to reuse methods and properties from the superclass.

When a method is called on an object of the `Child` class, Python first looks for that method in the `Child` class. If the method is not found in the `Child` class, Python looks for it in the `Parent` class. This process continues up the inheritance chain until the method is found or the top of the class hierarchy is reached.

[8]:
```
# Example usage
parent_obj = Parent("A", "B")
child_obj = Child("A", "B", "C")
print("Parent output:")
parent_obj.cool_function()  # Output: Cool function in Parent class
print("Child output:")
child_obj.cool_function()  # Output: Cool function in Parent class
```

```
Parent output:
Attribute1: A
Attribute2: B
Child output:
Attribute1: A
Attribute2: B
```

[9]:
```
class Child(Parent):
    def __init__(self, attr1, attr2, attr3):
        super().__init__(attr1, attr2)
        self.attr3 = attr3
```

```
    # def cool_function(self):
    #     super().cool_function()
    #     print(f"Attribute3: {self.attr3}")

child_obj = Child("A", "B", "C")
child_obj.cool_function()
```

```
Attribute1: A
Attribute2: B
```

[10]:
```
# Further hierarchy: GrandChild
# Code new class!
# class GrandChild(Child):
#      def __init__(self, attr1, attr2, attr3, attr4):
#          super().__init__(attr1, attr2, attr3)
#          self.attr4 = attr4

# grandKid_obj = GrandChild("A", "B", "C", "D")
# grandKid_obj.cool_function()
```

6 Unified Modelling Language (UML)

UML, is a standardized modeling language consisting of an integrated set of diagrams, developed to help software engineers specify, visualize, and document artifacts of software systems.

6.1 Class

Visibility:

- (public)
  # (protected)
  – (private)
  Attributes are given as: visibility name: type = defaultValue
  Operations are given as: visibility name (parameterlist) : return-type

6.2 Relationships

Relationships	Description
Exactly Once:	1
One or more:	1+
Optional:	1?
Bounded:	1..10
Unbounded:	* (0..n)

6.3 UML class diagram of our original Shape example

We will use UML diagrams to model our original Shape example, illustrating the relationships and structures between classes, methods, and attributes.

6.3.1 Python implementation

[11]:
```python
class Shape:
    def __init__(self, cord_x, cord_y, color):
        self.cord_x = cord_x
        self.cord_y = cord_y
        self.color = color

    def calculate_area(self):
        pass

    def draw(self, canvas):
        pass
```

[12]:
```python
class Circle(Shape):
    def __init__(self, cord_x, cord_y, color, radius):
        super().__init__(cord_x, cord_y, color)
        self.radius = radius

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

class Rectangle(Shape):
    def __init__(self, cord_x, cord_y, color, width, length):
        super().__init__(cord_x, cord_y, color)
        self.width = width
        self.length=length


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

class Triangle(Shape):
    def __init__(self, cord_x, cord_y, color, base, height):
        super().__init__(cord_x, cord_y, color)
        self.base = base
        self.height = height

    def calculate_area(self):
        return 0.5*self.base*self.height
```

[13]:
```
circle = Circle(10, 10, "blue", 1.784)
rectangle = Rectangle(15, 10, "red", 5 , 8)
triangle = Triangle(20, 10, "green", 4, 3)
triangle2 = Triangle(25, 10, "brown", 4, 4)

shapes = [circle, rectangle, triangle, triangle2]
print("-"*30)
[ print(f"The area of the {x.color} {type(x).__name__} is {round(x.
  ↪calculate_area())}") for x in shapes]
print("-"*30)
```

```
------------------------------
The area of the blue Circle is 10
The area of the red Rectangle is 40
The area of the green Triangle is 6
The area of the brown Triangle is 8
------------------------------
```

6.4 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single class to represent different underlying forms (types). Polymorphism ensures that the correct method is called based on the object's actual type.

6.4.1 Behavior:

Polymorphism enables the same method or property name to behave differently based on the object's actual class. This behavior is achieved through method overriding, where a subclass provides a specific implementation for a method that is already defined in its superclass.

6.4.2 Interface:

Polymorphism allows the use of a common link or base class to work with different types of objects. This concept is vital for creating modular and extensible code. A class can be polymorphic if it inherits from a common base class or implements a specific interface, ensuring a consistent interface across different subclasses.

`Circle` and `Rectangle` are both substitutable for `Shape`. You can use them interchangeably wherever a `Shape` is expected, and the behavior will again be consistent with the Liskov Substitution Principle.

6.4.3 Benefits of Polymorphism:

Flexibility and Extensibility: Polymorphism allows for easy addition of new classes without modifying existing code, enhancing the code's flexibility and extensibility.

Modularity: Polymorphic interfaces promote modularity by allowing classes to interact through well-defined interfaces, reducing dependencies between classes.

Code Reusability: Polymorphism encourages the reuse of code. Methods defined in base classes can be reused across multiple subclasses, promoting efficient code reuse.

Simplified Code: Polymorphism simplifies code by enabling the use of generic interfaces, making it easier to understand and maintain complex systems.

In summary, polymorphism is a powerful OOP concept that promotes flexibility, modularity, and code reuse by allowing objects of different types to be treated uniformly through a common interface or base class. Understanding and leveraging polymorphism are essential skills for effective object-oriented software design.

6.4.4 Method overriding overview:

Overriding

Definition

Enables a subclass to provide a specific implementation for a method already defined in its superclass.

Purpose

Promote code extensibility and flexibility, allowing subclasses to express unique behaviour while retaining the superclass structure.

Inheritance

Tightly coupled to inheritance. Subclasses provide a specialized implementation of the superclass.

Execution

Subclass takes precedence over superclass when called.

```
  <tr>
<td><b>Scope</b></td>
<td>Involves the superclass, as it provides the overridden method</td>
```

6.5 Liskov Substitution Principle (LSP)

Let $\Phi(x)$ be a property provable about objects x of type T. Then $\Phi(y)$ should be true for objects y of type S where S is a subtype of T.

The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass.

Behavior Preservation: Subclasses must preserve the behavior of the superclass

No Stronger Preconditions: Subclasses should not require stronger preconditions than the superclass.

No Weaker Postconditions: Subclasses should not provide weaker postconditions than the superclass.

Invariant Preservation: Invariants, which are conditions that are always true for an object, must be preserved by subclasses.

7 Example:

Consider our `Shape` example involving geometric shapes. If you have a superclass `Shape` with a method `calculate_area()`, any subclass like `Circle` or `Rectangle` should be able to provide its own implementation of `calculate_area()` without altering the code that uses the `Shape` class.

In this example, both `Circle` and `Rectangle` are substitutable for `Shape`. You can use them interchangeably wherever a Shape is expected, and the behavior will be consistent with the Liskov Substitution Principle.

By adhering to the Liskov Substitution Principle, you ensure that your inheritance hierarchy is well-structured and that changes in one part of the hierarchy do not adversely affect other parts, leading to more maintainable and robust code.

[14]:
```python
class Shape:
    def __init__(self, cord_x, cord_y, color):
        self.cord_x = cord_x
        self.cord_y = cord_y
        self.color = color

    def calculate_area(self):
        # weaker post-cond.
        # return round(0)
        pass

    def draw(self, canvas):
        pass

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

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

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

    def calculate_area(self):
        # stronger precondition
        # if self.width == 0:
            # raise Exception("Not doing anything")
        return self.width * self.height

# Using polymorphism
shapes = [Circle(5), Rectangle(5, 6)]
```

```
# weaker post-condition -> float instead of int
# shapes = [Circle(5), Rectangle(0.5, 6)]

for shape in shapes:
    print(f"Area: {shape.calculate_area()}")
```

```
Area: 78.5
Area: 30
```

7.1 Composition and Aggregation

Composition is another crucial OOP concept that involves creating objects of one class within another class to reuse functionality. Unlike inheritance, it doesn't establish a parent-child relationship but allows you to assemble objects to create more complex behaviors.

Aggregation is a specific form of composition where one class contains a reference to another class, but the child object can exist independently of the parent object. It represents a ``has-a'' relationship.

7.1.1 Task

In our enhanced shape example, we'll introduced a `Canvas`, serving as a visual platform to showcase various geometric shapes. Each shape is encapsulated within a `BoundingBox`, a rectangular frame specifically designed to accommodate the shape's dimensions. By utilizing this `BoundingBox`, we ensure that every shape fits neatly within its designated space on the `Canvas`. This integration not only enhances the aesthetic presentation but also allows for precise rendering of shapes, offering a visually appealing and organized representation of the geometric elements.

ToDo

Create BoundingBox:

defines the encapsulating width and length needed for each shape

Create a Canvas:

Works with Shapes

```
<li>Can calculate the total area of all shapes on the canvas</li>
<li>Can calculate the needed width and length for all the shapes</li></ul></li>
<li>Model the resulting classes using UML</li>
```

[15]:
```
class Canvas:
    def __init__(self):
        self.shapes = []

    def add_shape(self, shape):
        self.shapes.append(shape)

    def calculate_total_area(self):
        return sum(shape.calculate_area() for shape in self.shapes)
```

```
    def calculate_size(self):
        total_width = 0
        total_height = 0
        for shape in self.shapes:
            width, height = shape.get_bounds()
            total_width = max(total_width, shape.cord_x + width)
            total_height = max(total_height, shape.cord_y + height)
        return total_width, total_height
```

[16]:
```
class BoundingBox:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_dimensions(self):
        return self.width, self.height

class Shape:
    def __init__(self, cord_x, cord_y, color, boundedBox):
        self.cord_x = cord_x
        self.cord_y = cord_y
        self.color = color
        self.boundedBox = boundedBox

    def calculate_area(self):
        pass

    def draw(self, canvas):
        pass

    def get_bounds(self):
        return self.boundedBox.get_dimensions()
```

[17]:
```
class Circle(Shape):
    def __init__(self, cord_x, cord_y, color, radius):
        self.radius = radius
        self.diameter = 2* radius
        bBox = BoundingBox(self.diameter, self.diameter)
        super().__init__(cord_x, cord_y, color, bBox)

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

class Rectangle(Shape):
    def __init__(self, cord_x, cord_y, color, width, length):
        self.width = width
        self.length=length
```

```
        super().__init__(cord_x, cord_y, color, BoundingBox(width, length))

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

class Triangle(Shape):
    @staticmethod
    def compute_min_values(x, y, base, height):
        min_x = min(x, x + base)
        max_x = max(x, x + base)
        min_y = min(y, y + height)
        max_y = max(y, y + height)
        return max_x - min_x, max_y - min_y

    def __init__(self, cord_x, cord_y, color, base, height):
        width, height = self.compute_min_values(cord_x, cord_y, base, height)
        super().__init__(cord_x, cord_y, color, BoundingBox(width, height))
        self.base = base
        self.height = height

    def calculate_area(self):
        return 0.5*self.base*self.height
```

[18]:
```
rectangle = Rectangle(0, 0, "red", 5 , 8)
rectangle2 = Rectangle(5, 0, "blue", 5 , 8)
circle = Circle(10, 0, "blue", 1.784)
triangle = Triangle(15, 0, "green", 4, 3)
triangle2 = Triangle(20, 0, "brown", 4, 4)
```

[19]:
```
def create_canvas(shape_list):
    canvas = Canvas()
    [canvas.add_shape(x) for x in shape_list]
    total_area = canvas.calculate_total_area()
    print("Total area of the shapes inside the canvas: ", total_area)
    canvas_width, canvas_height = canvas.calculate_size()
    print("Canvas size needed: Width =", canvas_width, ", Height =",␣
 ↪canvas_height)
```

[20]:
```
print("Rectangles only:")
rectangles_only = [rectangle, rectangle2]
create_canvas(rectangles_only)
print(f"Each rectangle has a widht of 5 and both have the x axis set to 0."
      f"\nThus, we get a width of 10 (5+5)\n")

print("All shapes:")
shapes = [rectangle, rectangle2, circle, triangle, triangle2]
create_canvas(shapes)
```

```
Rectangles only:
Total area of the shapes inside the canvas:  80
Canvas size needed: Width = 10 , Height = 8
Each rectangle has a widht of 5 and both have the x axis set to 0.
Thus, we get a width of 10 (5+5)

All shapes:
Total area of the shapes inside the canvas:  103.99860870850348
Canvas size needed: Width = 24 , Height = 8
```

8 Additional examples

8.1 Example - Car composition

Specification:

The car must have an engine

The car needs four tires

If any of your tires has less than 33 psi pressure throw a warning when starting the car

Add a fuel tank with a load capacity of 55 liters

When the car is driving it consumes fuel and the tank content decreases

Add a warning when the tank has less than 10 liters left

[21]:
```python
class Engine:
    def __init__(self, model, brand):
        self.model = model
        self.brand = brand

    def start(self):
        print("Engine started")
```

[22]:
```python
class Tire:
    def __init__(self, pressure, size):
        self.pressure = pressure
        self.size = size
        self.model = "Awesome tire"

    def inflate(value):
        pass

    def deflate(value):
        pass
```

[23]:
```python
class Car:
    def __init__(self, model, brand, consumption):
        self.model = model
```

```
        self.brand = brand
        self.consumption = consumption
        self.started = False
        self.engine = Engine("Tundra", "Toyota")
        self.tires = [Tire(33, 15) for _ in range(4)]

    def start(self):
        self.engine.start()
        self.started = True

    def check_tires():
        pass

    def drive(self, distance):
        if not self.started:
            raise Exception("Car not started")
        print("Car is moving")
```

[24]:
```
my_car = Car("carina", "Toyota", 2)
try:
    my_car.drive(10)
except Exception as e:
    print(f"Driving not possible because '{e}'")

my_car.start()
my_car.drive(10)
```

```
Driving not possible because 'Car not started'
Engine started
Car is moving
```

8.2 Example - Class room aggregation

Define classes for `Student`, `Teacher`, and `Classroom`. Both `Teacher` and `Student` extend the `Person`class which holds common attributes. The `Teacher` can mark attendance based on the students present inside a `Classroom`.

[25]:
```
class Classroom:
    def __init__(self):
        self.students = []

    def add_student(self, student):
        self.students.append(student)
```

[26]:
```
class Person:
    def __init__(self, name, surname, age):
        self.name = name
        self.surname = surname
```

```
        self.age = age

    def do_something(thing):
        print(thing)
```

[27]:
```
class Student(Person):
    def __init__(self, name, surname, age, id, gpa):
        super().__init__(name, surname, age)
        self.student_id = id
        self.gpa = gpa
```

[28]:
```
class Teacher(Person):
    def __init__(self,  name, surname, age, id):
        super().__init__(name, surname, age)
        self.employee_id = id

    def markAttendance(self, classroom):
        attending_students = ', '.join(student.name for student in classroom.
 ↪students)
        print(f"Students attending this class: {attending_students}")
```

[29]:
```
classroom = Classroom()
frank = Teacher("Victor", "Frankenstein", 205, 44)
frank.markAttendance(classroom)

steve = Student("Stephen", "King", 76,  1, 1)
shelly = Student("Mary", "Shelley", 226, 2, 1)

classroom.add_student(steve)
classroom.add_student(shelly)

frank.markAttendance(classroom)
```

```
Students attending this class:
Students attending this class: Stephen, Mary
```

9 Additional concepts

9.1 Method Overloading and Operator Overloading:

Polymorphism also extends to method overloading and operator overloading. Method overloading allows multiple methods with the same name but different parameters, while operator overloading enables custom behavior for operators like `+`, `-`, `*`, `etc.`, based on the object's type.

[30]:
```
class MethodOverloading:
    # first definition of function add
    def add(a, b):
```

```
        return a + b

    # second definition of function add
    def add(a, b, c):
        return a + b + c

print(MethodOverloading.add(1, 2, 3))
# print(MethodOverloading.add(1, 2)) # Exception due to redefinition of add␣
 ↪method
```

```
6
```

9.1.1 Python doesn't support method overloading

We can define the methods, but python will only support the last definition of the method.

However, Python does support methods with multiple arguments as a work-around:

[31]:
```
class MethodOverloading:
    # first definition of function add
    def add(a, b, *args):
        answer = a + b
        for x in args:
            answer += x
        return answer

print(MethodOverloading.add(1, 2))
print(MethodOverloading.add(1, 2, 3))
print(MethodOverloading.add(1, 2, 3, 5))
```

```
3
6
11
```

9.2 Operator overloading

If we wish to do addition of two instances of a class python will throw a type error as it is unknown how those two objects should be handled.

[32]:
```
class OperatorOverloading:
    def __init__(self, variable):
        self.var = variable

obj1 = OperatorOverloading(1)
obj2 = OperatorOverloading(2)
#print(obj1 + obj2) # TypeError
```

We can adjust how operators are handled for a class. Similar to what you saw in the previous lecture with the `__str__` method

[33]:
```python
class OperatorOverloading:
    def __init__(self, variable):
        self.var = variable

    # add definition of binary operator
    def __add__(self, o):
        return self.var + o.var

obj1 = OperatorOverloading(1)
obj2 = OperatorOverloading(2)

print(obj1 + obj2)
```

```
3
```

9.2.1 Difference between Method overriding and overloading:

Topic

Overloading

Overriding

Definition

Allows a class to define multiple methods with the same name but varied parameters.

Enables a subclass to provide a specific implementation for a method already defined in its superclass.

Purpose

Group related functionalities under the same method name.

Promote code extensibility and flexibility, allowing subclasses to express unique behaviour while retaining the superclass structure.

Inheritance

Not directly related to inheritance.

Tightly coupled to inheritance. Subclasses provide a specialized implementation of the superclass.

Execution

Decides which version to execute based on the number and types of arguments during a function call.

Subclass takes precedence over superclass when called.

```
  <tr>
<td><b>Scope</b></td>
<td>Limited to the class in which the methods are defined</td>
<td>Involves the superclass, as it provides the overridden method</td>
```