Classes

Solve the following exercises and upload your solutions to Moodle until the specified due date. Make sure to use the *exact filenames* that are specified for each individual exercise. Unless explicitly stated otherwise, you can assume correct user input and correct arguments. You are allowed to implement additional attributes and methods as long as the original interface remains unchanged.

Exercise 1 – Submission: `ex1.py` — 20 Points

Create a class `Complex` that models complex numbers. The class has the following instance attributes:

- `real: float`

  Represents the real part of the complex number.

- `imaginary: float`

  Represent the imaginary part of the complex number.

The class has the following instance methods:

- `__init__(self, real: float, imaginary: float)`

  Sets both instance attributes.

- `print(self)`

  Prints this `Complex` object to the console. Format: `<real> <sign> <imag>i`, where `<real>` is the real part of the complex number, `<imag>` is the imaginary part (followed by the character `i`), and `<sign>` is either the plus character `+` if the imaginary part is positive or the minus character `-` if the imaginary part is negative. Example: `1.2 - 5.4i`

- `abs(self) -> float`

  Returns the absolute value of this `Complex` object. The absolute value of a complex number is defined as $|a + bi| = \sqrt{a^2 + b^2}$, where a and b are the real and imaginary parts, respectively.

Example program execution:

```
c1 = Complex(1.2, -5.4)
c1.print()
c2 = Complex(3.0, 4.0)
c2.print()
print(c2.abs())
```

Example output:

```
1.2 - 5.4i
3.0 + 4.0i
5.0
```

Exercise 2 – Submission: `ex2.py` 20 Points

Implement the following additional instance method in the `Complex` class from above:

- `add(self, other: "Complex")`

  Adds `other` to this `Complex` object in-place (no return value). If `other` is not an instance of class `Complex`, raise a `TypeError`. Adding means adding calculating the sum of the two real parts and the sum of the two imaginary parts, respectively.

Moreover, add the following static method (`@staticmethod`):

- `add_all(comp: "Complex", *comps: "Complex") -> "Complex"`

  Adds `comp` and all numbers in `*comps` together and returns a new `Complex` object containing this sum. None of the input arguments must be changed, i.e., all complex numbers specified by `comp` and `*comps` must remain the same. If any of `comp` or `*comps` are not instances of class `complex`, raise a `TypeError`. Use the `Complex.add` method from above to avoid code duplication.

Example program execution:

```
c1 = Complex(1.0, -2.0)
c1.print()
c2 = Complex(9.0, 100.0)
c1.add(c2)
c1.print()
c_sum = Complex.add_all(c1, c1, c2, Complex(33.75, -14.25))
c_sum.print()
c1.print()
will_fail = Complex.add_all(100)
```

Example output (error message is up to you and may differ):

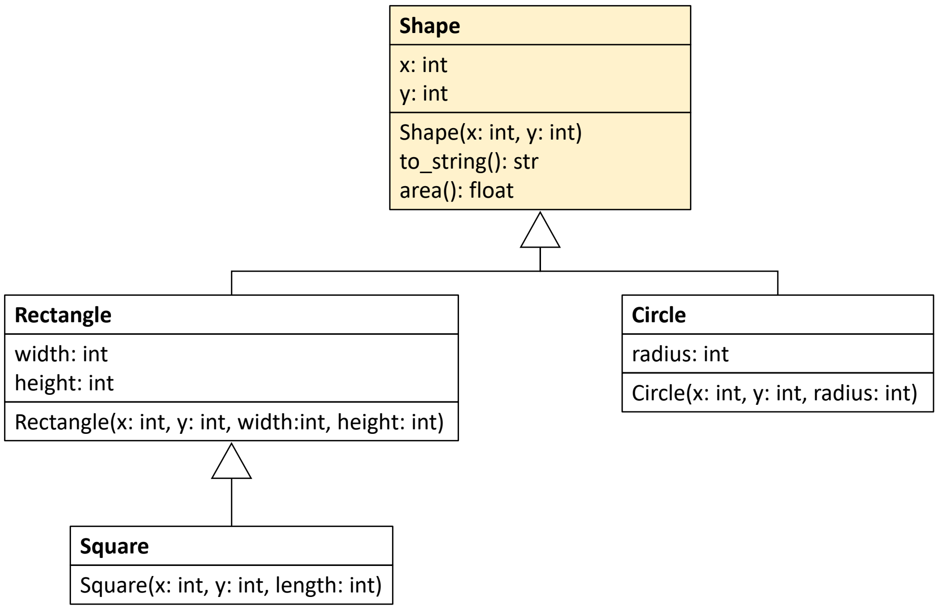
```
1.0 - 2.0i
10.0 + 98.0i
62.75 + 281.75i
10.0 + 98.0i
TypeError: can only add 'Complex', not 'int'
```

Hints:

- In the `add_all` method, you must not change the given complex numbers. Instead, create a new `Complex` object (choose an appropriate initialization for its real and imaginary part) which you can then freely change and return afterwards.

Exercise 3 – Submission: `ex3.py` 15 Points

You are given the following class hierarchy that models shapes in a 2D plane. You will have to implement the classes in this and the following exercises.



In this exercise, you have to implement the class `Shape`, which represents the base class of 2D shapes. The class has the following instance attributes:

- `x: int`

  The x-coordinate of the shape.

- `y: int`

  The y-coordinate of the shape.

The class has the following instance methods:

- `__init__(self, x: int, y: int)`

  Sets both instance attributes.

- `to_string(self) -> str`

  Returns a string representation of the form `"Shape: x=<x_value>, y=<y_value>"`, where `<?_value>` represents the value of the corresponding attribute.

- `area(self) -> float`

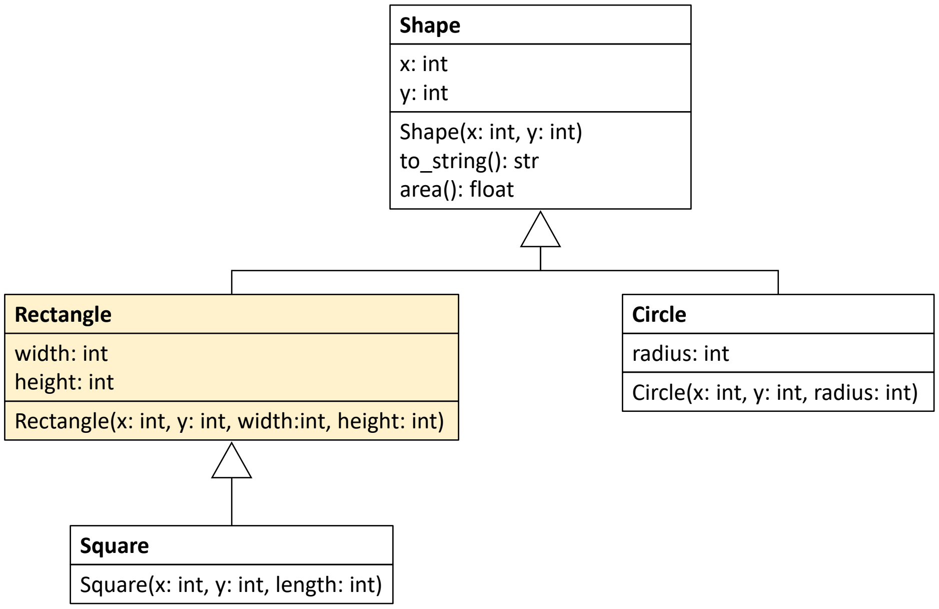
  Returns the area of the shape as a float, which must be implemented by all concrete subclasses. In the `Shape` class, a `NotImplementedError` is raised.

Hints:

- In the `to_string` method, you need the name of the class. While this could be hard-coded, you could also use `type(x).__name__` to get the name of the type/class of some object `x`. This has the benefit that instances of any subclasses will also return their correct names. Alternatively, you can write a helper method that returns the name, and you override it in the sublcasses.

Exercise 4 – Submission: `ex4.py` 15 Points

You are given the same class hierarchy as in the previous exercise that models shapes in a 2D plane.



In this exercise, you have to implement the concrete subclass `Rectangle`, which represents a rectangle shape and extends the base class `Shape`. The class has the following additional instance attributes:

- `width: int`

  The width of the rectangle.

- `height: int`

  The height of the rectangle.

The class has the following instance methods (reuse code from superclasses to avoid unnecessary code duplication):

- `__init__(self, x: int, y: int, width: int, height: int)`

  Sets both instance attributes (in addition to the attributes of the base class `Shape`).

- `to_string(self) -> str`

  Returns a string representation of the form `"Rectangle: x=<x_value>, y=<y_value>, width =<width_value>, height=<height_value>"`, where `<?_value>` represents the value of the corresponding attribute.

- `area(self) -> float`

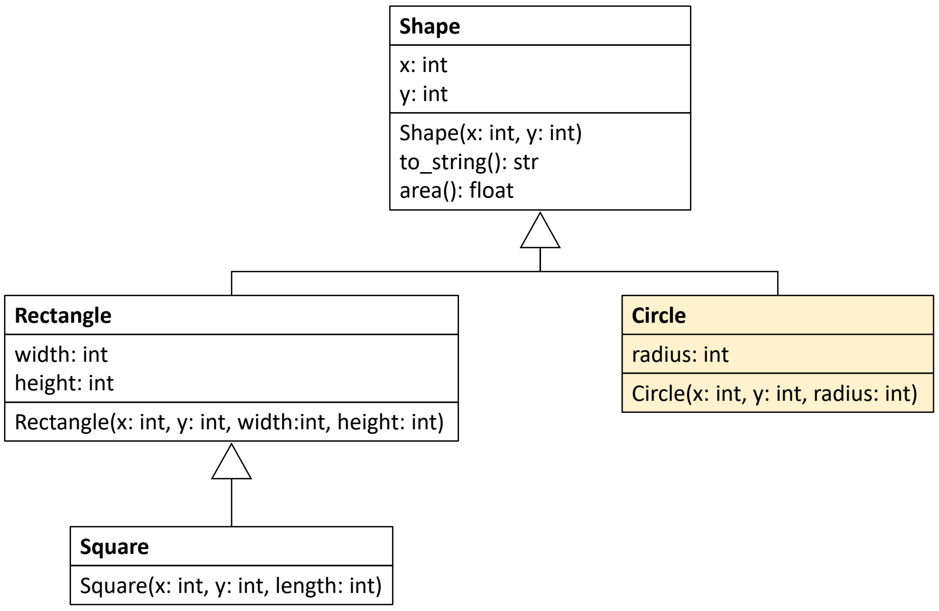
  Returns the area of the rectangle, i.e., `width * height`, as a float.

Hints:

- You can import the previous exercise as module to avoid having to copy the entire class hierarchy. For example, you can write `from ex3 import Shape`.
- Use `super().some_method()` to access the `some_method` implementation of the superclass.

Exercise 5 – Submission: `ex5.py` 15 Points

You are given the same class hierarchy as in the previous exercise that models shapes in a 2D plane.



In this exercise, you have to implement the concrete subclass `Circle`, which represents a circle shape and extends the base class `Shape`. The class has the following additional instance attributes:

- `radius: int`

  The radius of the circle.

The class has the following instance methods (reuse code from superclasses to avoid unnecessary code duplication):

- `__init__(self, x: int, y: int, radius: int)`

  Sets the instance attribute (in addition to the attributes of the base class `Shape`).

- `to_string(self) -> str`

  Returns a string representation of the form `"Circle: x=<x_value>, y=<y_value>, radius =<radius_value>"`, where `<?_value>` represents the value of the corresponding attribute.

- `area(self) -> float`

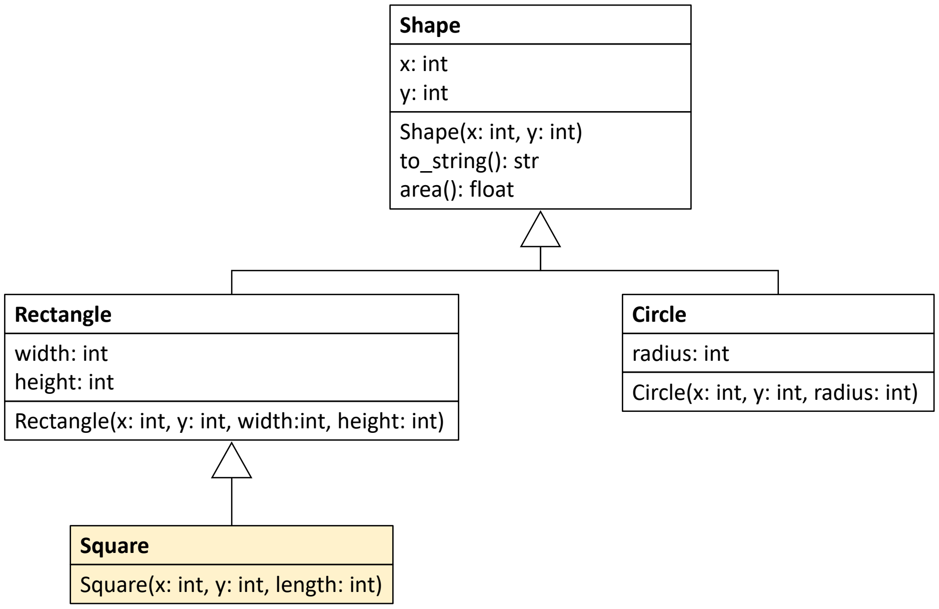
  Returns the area of the circle, i.e., `radius`^{2} `* math.pi`, where `math` is a Python built-in module that must be imported.

Hints:

- You can import the previous exercise as module to avoid having to copy the entire class hierarchy. For example, you can write `from ex3 import Shape`.
- Use `super().some_method()` to access the `some_method` implementation of the superclass.

Exercise 6 – Submission: `ex6.py` 15 Points

You are given the same class hierarchy as in the previous exercise that models shapes in a 2D plane.



In this exercise, you have to implement the concrete subclass `Square`, which represents a square shape and extends the base class `Rectangle`. The class has *no* additional instance attributes.

The class has the following instance methods (reuse code from superclasses to avoid unnecessary code duplication):

- `__init__(self, x: int, y: int, length: int)`

  The `length` parameter only exists for user convenience. Internally, the attributes `width` and `height` of the base class `Rectangle` are used (both attributes will be set to this `length` value).

- `to_string(self) -> str`

  Returns a string representation of the form `"Square: x=<x_value>, y=<y_value>, width =<width_value>, height=<height_value>"`, where `<?_value>` represents the value of the corresponding attribute.

- `area(self) -> float`

  Returns the area of the square, i.e., `width * height`, as a float.

Hints:

- You can import the previous exercise as module to avoid having to copy the entire class hierarchy. For example, you can write `from ex4 import Rectangle`.
- The above requirement "reuse code from superclasses to avoid unnecessary code duplication" is especially relevant in this exercise. Depending on your implementation, it might be that you do not need to override any methods of the superclass `Rectangle`.

Combined Examples for Exercises 3, 4, 5 and 6

Example program execution:

```
s = Shape(4, 9)
print(s.to_string())

r = Rectangle(1, 2, 3, 4)
print(r.to_string())
print("Rectangle area:", r.area())

c = Circle(5, 2, 2)
print(c.to_string())
print("Circle area:", c.area())

s = Square(0, 0, 10)
print(s.to_string())
print("Square area:", s.area())
```

Example output:

```
Shape: x=4, y=9
Rectangle: x=1, y=2, width=3, height=4
Rectangle area: 12.0
Circle: x=5, y=2, radius=2
Circle area: 12.566370614359172
Square: x=0, y=0, width=10, height=10
Square area: 100.0
```