Introduction to programming using Python

Session 8

Matthieu Choplin

matthieu.choplin1@city.ac.uk

http://moodle.city.ac.uk/

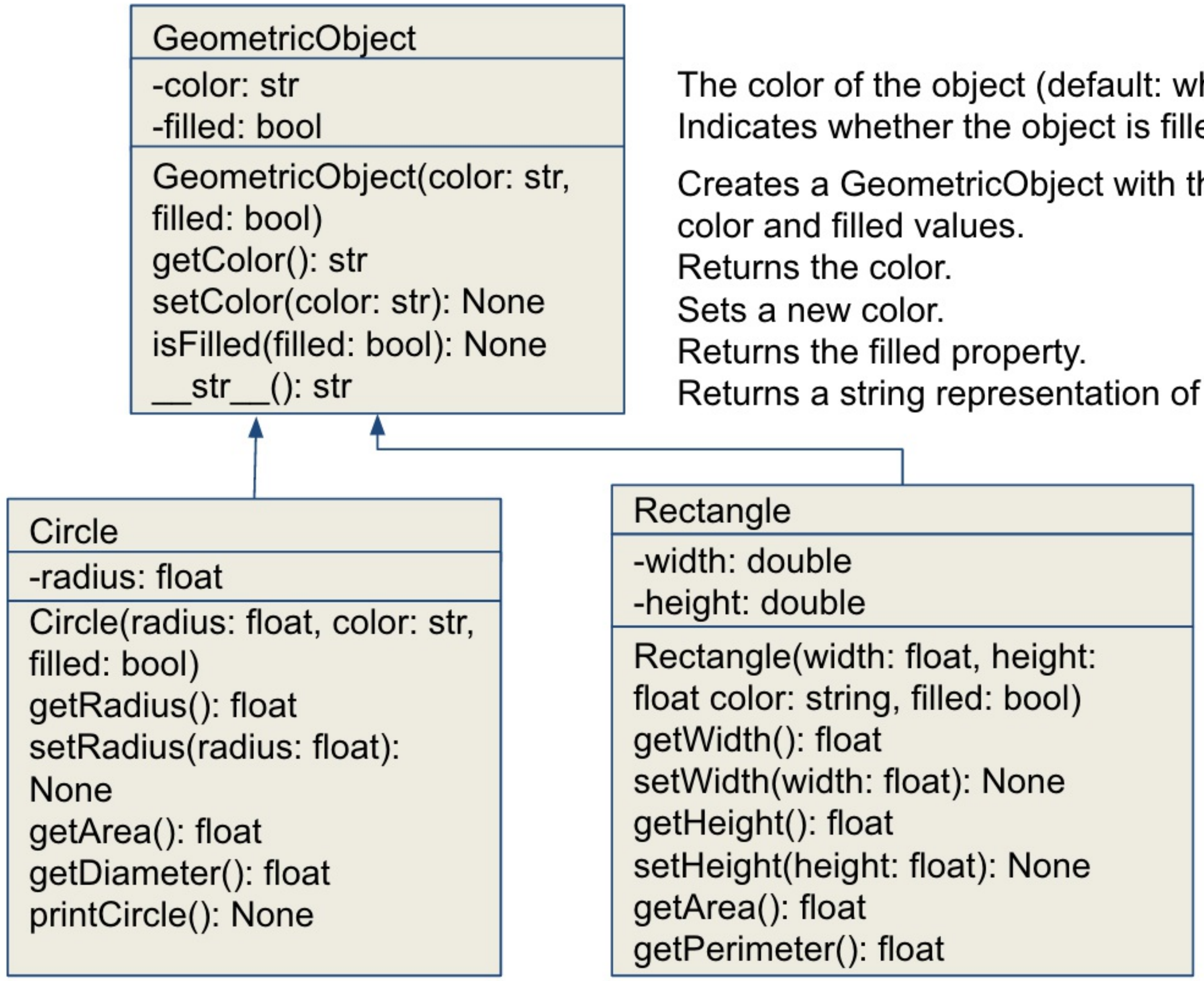
Objectives

- To develop a subclass from a superclass through inheritance
- To override methods in the subclass
- To understand encapsulation in Python
- To explore the object class and its methods
- To understand polymorphism and dynamic binding
- To determine if an object is an instance of a class using the isinstance function
- To discover relationships among classes
- To design classes using composition and inheritance relationships

Definition

- Inheritance enables you to define a general class (a superclass) and later extend it to more specialized classes (subclasses).
- Example: a class Rectangle and a class Circle. They share common attributes and methods such as the attribute color.
- Common attributes and methods can be put in a parent class.
- Using inheritance enables to **avoid redundancy**

UML representation of inheritance



GeometricObject

-color: str
-filled: bool

GeometricObject(color: str, filled: bool)
getColor(): str
setColor(color: str): None
isFilled(filled: bool): None
__str__(): str

The color of the object (default: white).
Indicates whether the object is filled with a color.

Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Returns a string representation of this object.

Circle

-radius: float

Circle(radius: float, color: str, filled: bool)
getRadius(): float
setRadius(radius: float): None
getArea(): float
getDiameter(): float
printCircle(): None

Rectangle

-width: double
-height: double

Rectangle(width: float, height: float color: string, filled: bool)
getWidth(): float
setWidth(width: float): None
getHeight(): float
setHeight(height: float): None
getArea(): float
getPerimeter(): float

Superclasses and Subclasses

- The syntax of inheritance is:

```
class Child(Parent):
   # class body
```

- If you want to call the method of the superclass, use super()
- In particular, call super().__init__() to get the superclass attributes accessible from the subclass

As an example, see the following programs:

- GeometricObject.py
- CircleDerivedFromGeometricObject.py
- RectangleDerivedFromGeometricObject.py
- TestCircleRectangle.py

Try to fix the program

```python
class A:
    def __init__(self, i = 0):
        self.i = i

class B(A):
    def __init__(self, j = 0):
        self.j = j

def main():
    b = B()
    print(b.i)
    print(b.j)
main()
```

👁 Solution

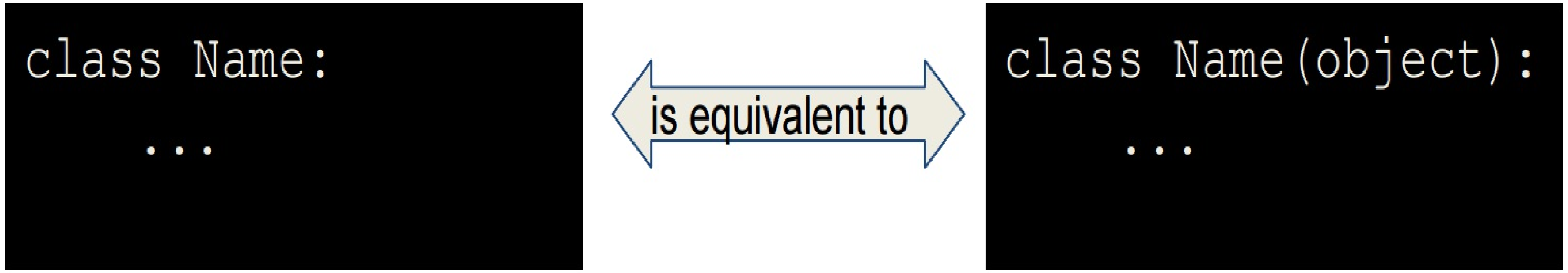
Overriding Methods

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as **method overriding**.

```
class Circle(GeometricObject):
    # Other methods are omitted
    # Override the __str__ method defined in GeometricOb
    def __str__(self):
        return super().__str__() + " radius: " + \
            str(radius)
```

The object Class

- Every class in Python is descended from the **object** class. If no inheritance is specified when a class is defined, the superclass of the class is object by default.



```
class Name:
    ...
```

is equivalent to

```
class Name(object):
    ...
```

- There are more than a dozen methods defined in the object class. We discuss four methods __new__(), __init__(), __str__(), and __eq__(other) here.

The __new__, __init__ Methods

- All methods defined in the object class are special methods with two leading underscores and two trailing underscores.
- The __new__() method is automatically invoked when an object is constructed. This method then invokes the __init__() method to initialize the object. Normally you should only override the __init__() method to initialize the data fields defined in the new class.

The __str__ Method

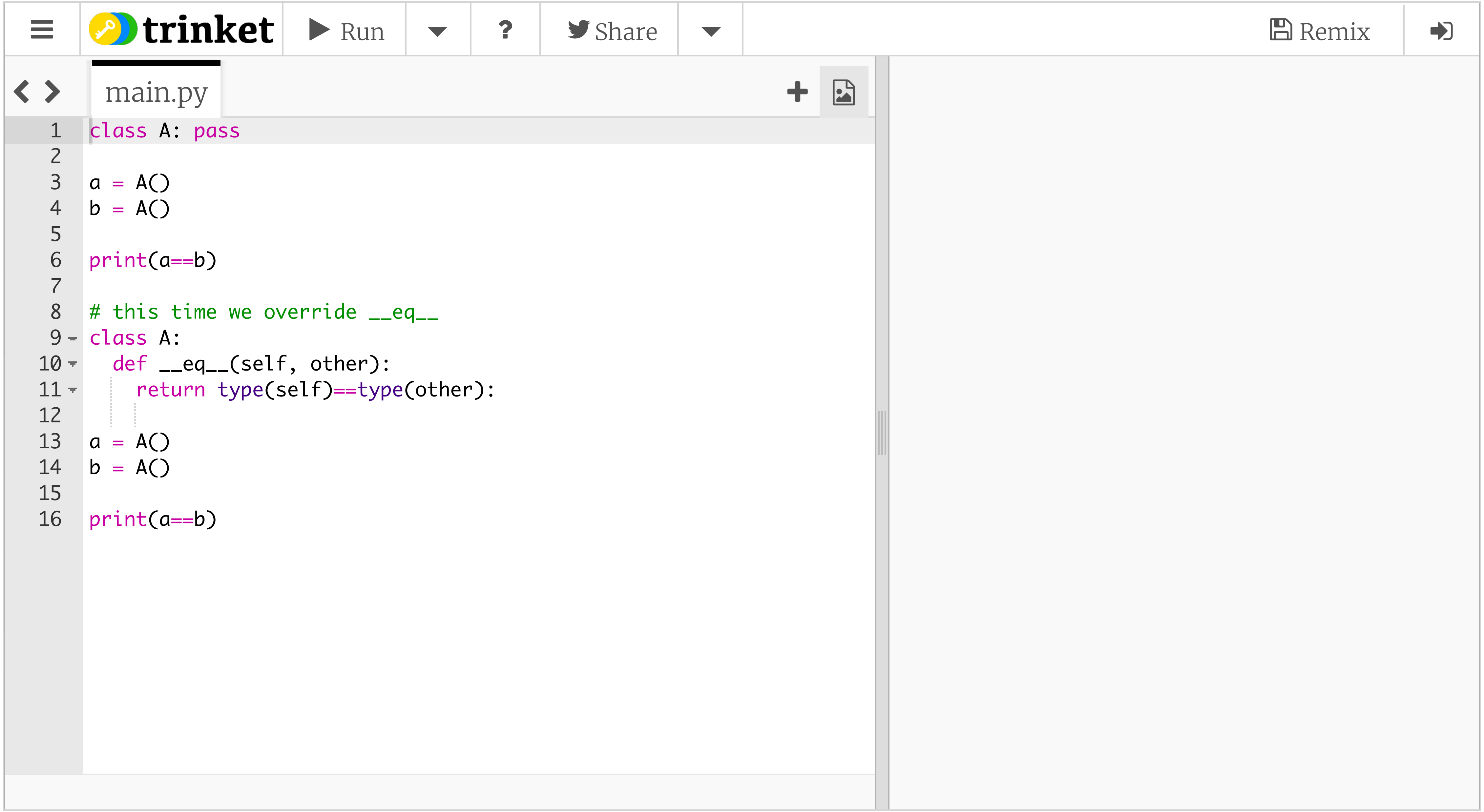
- The __str__() method returns a string representation for the object. By default, it returns a string consisting of a class name of which the object is an instance and the object’s memory address in hexadecimal.

```
def __str__(self):
   return "color: " + self.__color + \
       " and filled: " + str(self.__filled)
```

The __eq__ Method

- The __eq__(other) method returns True if two objects are the same. By default, x.__eq__(y) (i.e., x == y) returns False, but x.__eq__(x) is True. You can override this method to return True if two objects have the same contents.

Override __eq__



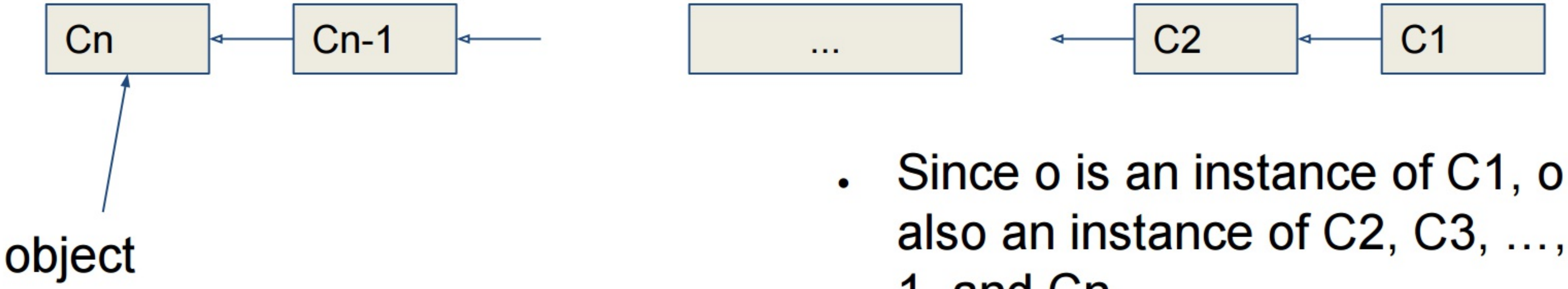
```
class A: pass

a = A()
b = A()

print(a==b)

# this time we override __eq__
class A:
  def __eq__(self, other):
    return type(self)==type(other):

a = A()
b = A()

print(a==b)
```

Polymorphism

- The inheritance relationship enables a subclass to inherit features from its superclass with additional new features.
- A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa. For example, every circle is a geometric object, but not every geometric object is a circle. Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.
- Examples:
  - PolymorphismDemo.py RectangleFromGeometricObject.py CircleFromGeometricObject.py
  - Animals.py

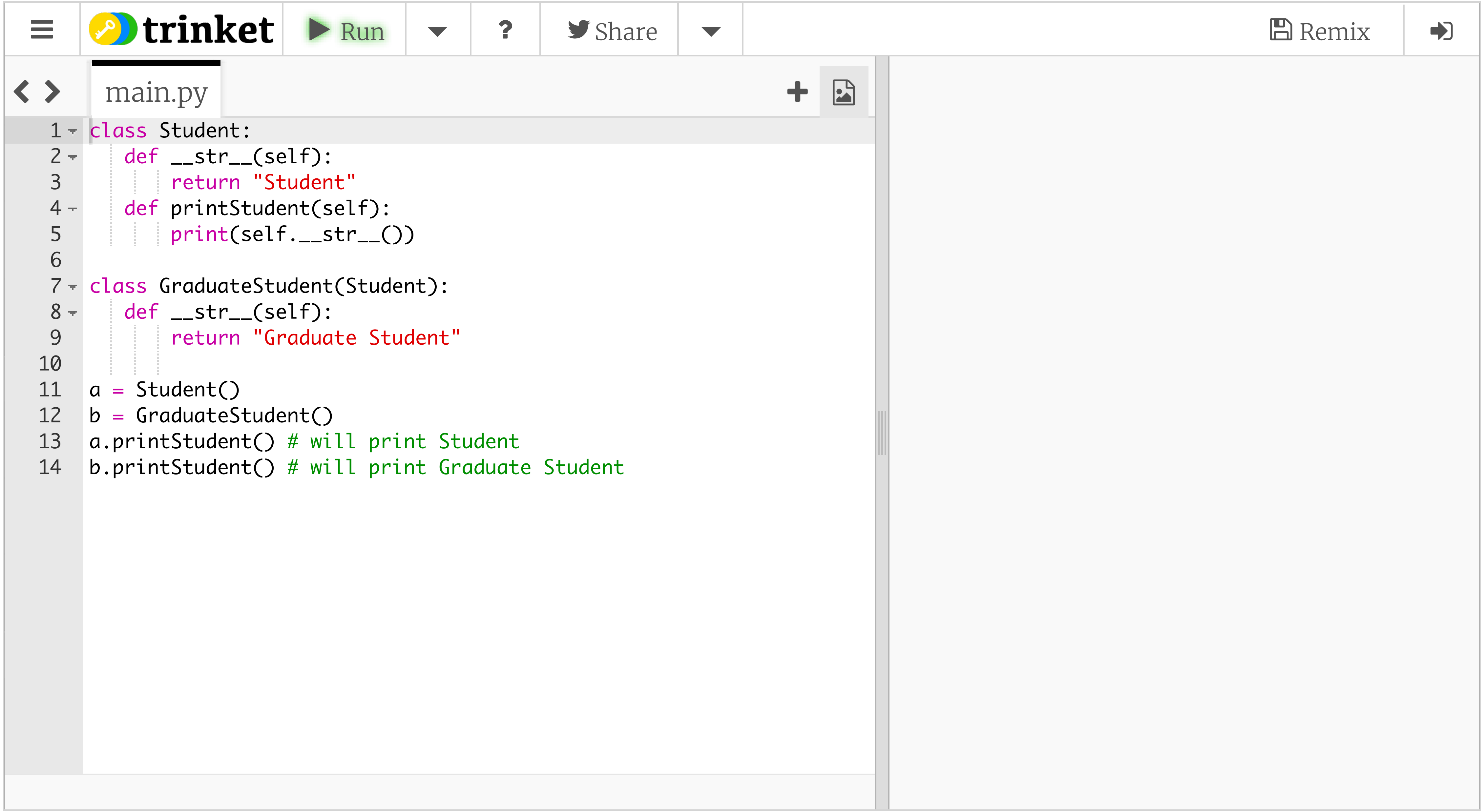
Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C1, C2, ..., Cn-1, and Cn, where C1 is a subclass of C2, C2 is a subclass of C3, ..., and Cn-1 is a subclass of Cn. That is, Cn is the most general class, and C1 is the most specific class. In Python, Cn is the object class. If o invokes a method p, Python searches the implementation for the method p in C1, C2, ..., Cn-1 and Cn, in this order, until it is found. Once an implementation is found, the search stops and **the first-found implementation is invoked**.



- Since o is an instance of C1, o is also an instance of C2, C3, …, Cn-1, and Cn

Dynamic Binding: example



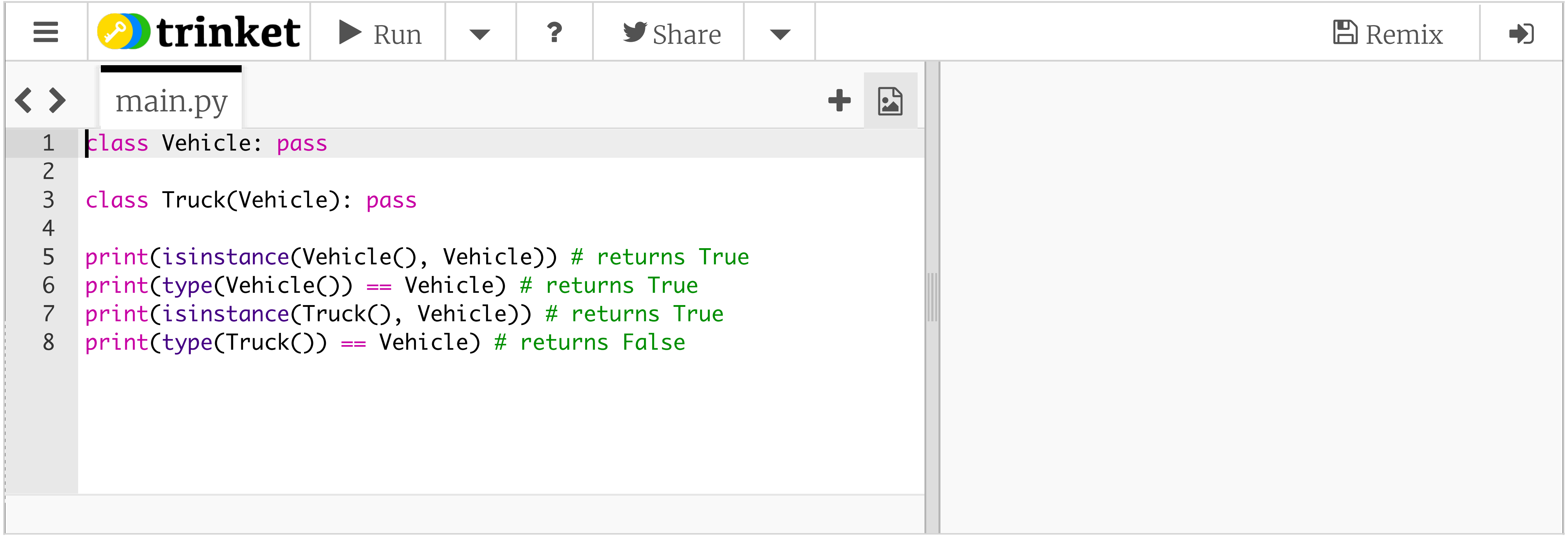
```python
class Student:
    def __str__(self):
        return "Student"
    def printStudent(self):
        print(self.__str__())

class GraduateStudent(Student):
    def __str__(self):
        return "Graduate Student"

a = Student()
b = GraduateStudent()
a.printStudent() # will print Student
b.printStudent() # will print Graduate Student
```

The isinstance Function

- The isinstance function can be used to determine if an object is an instance of a class.
- See the example program IsinstanceDemo.py

isinstance() compared to type()

- isinstance take into account inheritance, an instance of a derived class is an instance of a base class too

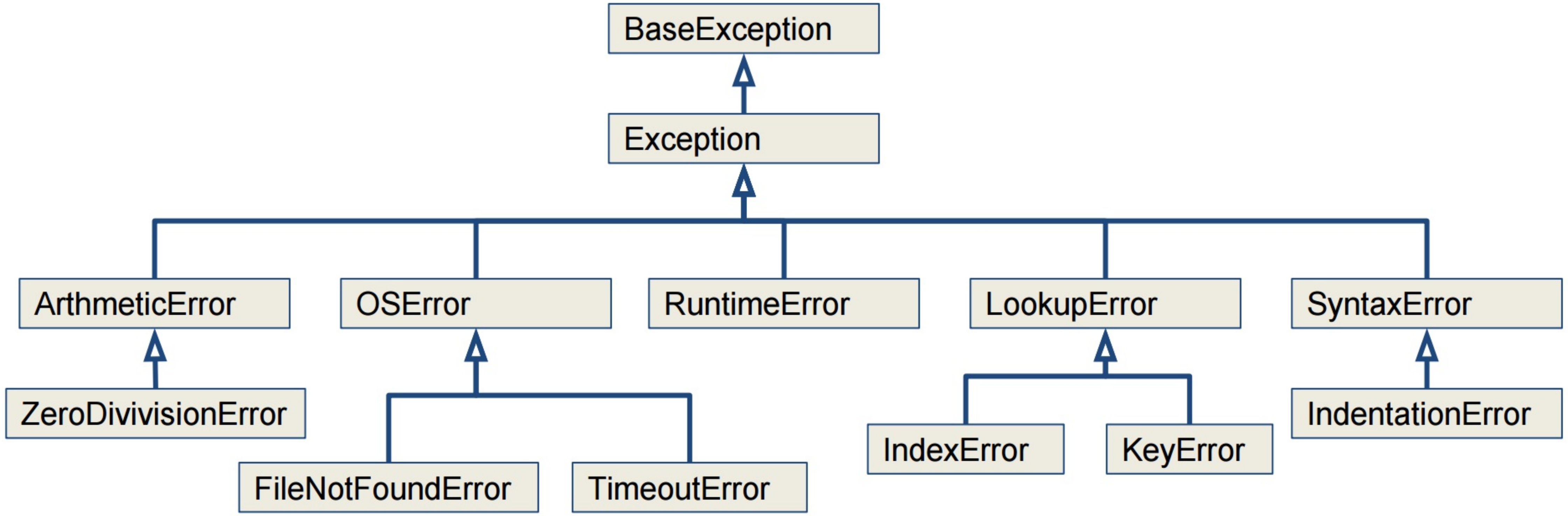


```python
class Vehicle: pass

class Truck(Vehicle): pass

print(isinstance(Vehicle(), Vehicle)) # returns True
print(type(Vehicle()) == Vehicle) # returns True
print(isinstance(Truck(), Vehicle)) # returns True
print(type(Truck()) == Vehicle) # returns False
```

- NB: the instance are created on the fly here, we do not pass them to a variable

The hierarchy of the type of Exceptions

You can find the full hierarchy on https://docs.python.org/3/library/exceptions.html#exception-hierarchy



Defining Custom Exception Classes

See how we inherit from *RunTimeError* in the class InvalidRadiusException in the example CircleWithCustomException.py and how we use it in TestCircleWithCustomException.py

Encapsulation

- The syntax we have seen so far for data encapsulation is to use 2 underscore in front of the attribute we want to hide, which forces us to use getter and setter to access and modify the field.

```
class C:
    def __init__(self,x):
        self.__x = x

    def getX(self):
        return self.__x

    def setX(self, x):
        self.__x = x
```

Encapsulation and data mangling

- The use of double leading underscores causes the name to be **mangled** to something else. Specifically, the private attributes in the preceding class get renamed to _C__x. At this point, you might ask what purpose such name mangling serves. The answer is inheritance - such attributes cannot be overridden via inheritance. For example:

```
class C:
    def __init__(self,x):
        self.__x = x

class A(C):
    def __init__(self):
        super().__init__(2)
        # Does not override C.__x
        self.__x = 1

a = A()
print(a._A__x)
print(a._C__x)
```

Encapsulation in a more pythonic way

- We can use **property** to customize access to an attribute

```
class C:
    def __init__(self,x):
        self.setX(x)

    def getX(self):
        return self.__x

    def setX(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x

    x = property(getX, setX)
```

Equivalent using decorators

```
class P:
    def __init__(self,x):
        self.x = x

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x
```

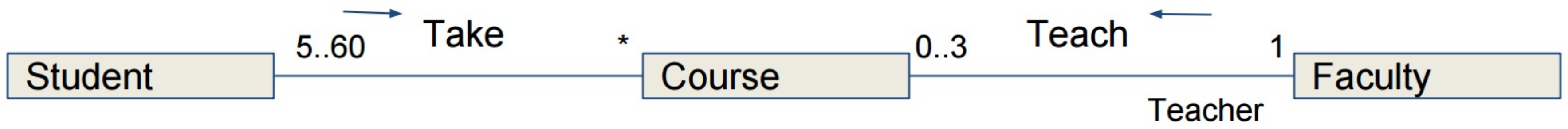
Properties should only be used in cases where you actually need to perform extra processing on attribute access

Relationships among Classes

- Association
- Aggregation
- Composition
- Inheritance

Association

- Association represents a general binary relationship that describes an activity between two classes.



```
class Student:
    def addCourse(self,
        courses):
       # add course
       # to a list
```

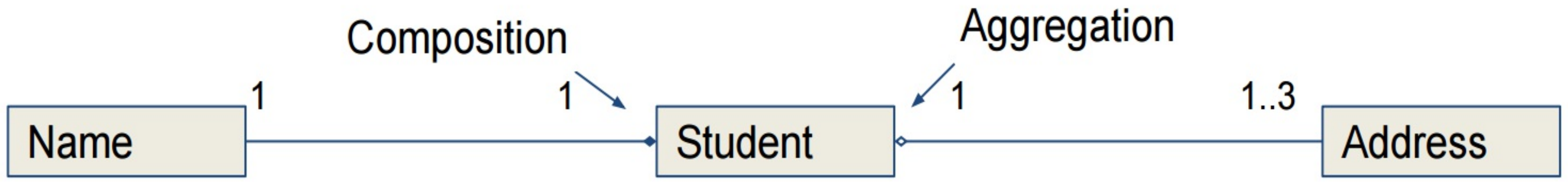
```
class Course:
    def addStudent(self,
        student):
       # add student
       # to a list
    def setFaculty(self,
        faculty):
```

```
class Faculty:
    def addCourse(self,
        course):
       # add course
       # to a list
```

- The association relations are implemented using data fields and methods in classes.

Aggregation and Composition

- Aggregation is a special form of association, which represents an ownership relationship between two classes. Aggregation models the has-a relationship. If an object is exclusively owned by an aggregated object, the relationship between the object and its aggregated object is referred to as composition.



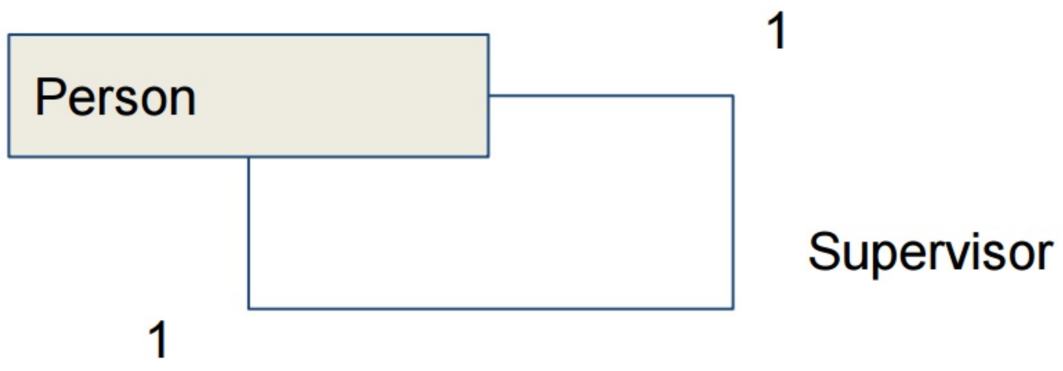
```
class Name:
    ...
```

```
class Student:
    def __init__(self,name, addresses):
        self.name = name
        self.addresses = addresses
```

```
class Addresses:
    ...
```

Aggregation Between Same Class objects

- Aggregation may exist between objects of the same class. For example, a person may have a supervisor.



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

is-a relationship vs has-a relationship

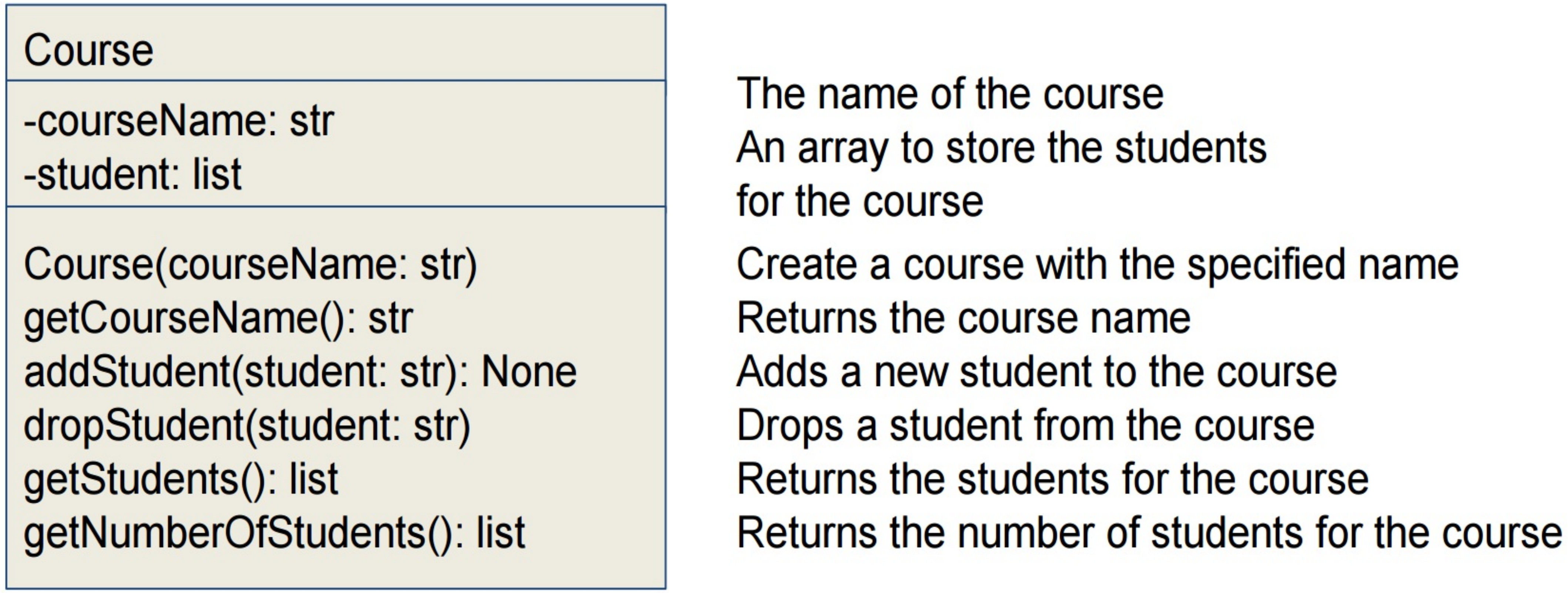
- Inheritance is for the **is-a** relationship
- Composition and aggregation is for the **has-a** relationship

Multiple inheritance

- Syntax for multiple inheritance:

```
class Child(ParentA, ParentB):
    # rest of the class
```

The Course Class



Course	
-courseName: str	The name of the course
-student: list	An array to store the students for the course
Course(courseName: str)	Create a course with the specified name
getCourseName(): str	Returns the course name
addStudent(student: str): None	Adds a new student to the course
dropStudent(student: str)	Drops a student from the course
getStudents(): list	Returns the students for the course
getNumberOfStudents(): list	Returns the number of students for the course

- See the programs:
  - TestCourse.py to see how it is used
  - Course.py to see how it is implemented