



Introduction to programming using Python

Session 8

Matthieu Choplin

matthieu.choplin@city.ac.uk

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



Objectives

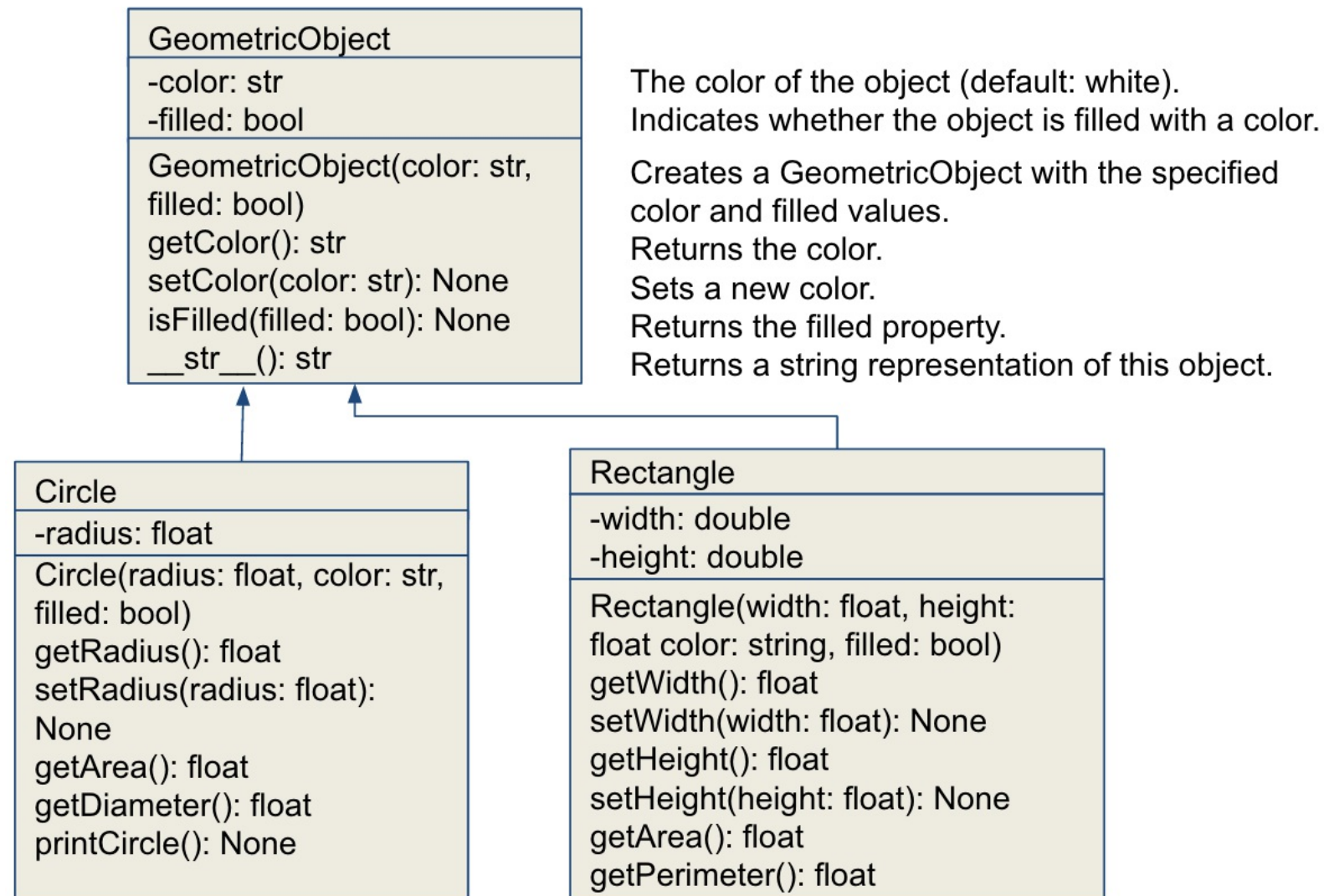
- To develop a subclass from a superclass through inheritance
- To override methods in the subclass
- To understand encapsulation in Python
- To explore the object class and its methods
- To understand polymorphism and dynamic binding
- To determine if an object is an instance of a class using the `isinstance`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





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](#)
- [TestCircle.py](#)

Try to fix the program

```
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



Exercise: create a Rectangle Class inheriting from GeometricObject

- We already have the [class diagram](#)
 - We want to be able to run this script:
[TestCircleRectangle.py](#)
 - Create the file `RectangleDerivedFromGeometricObject.py` in which you will create the class **Rectangle** inheriting from **GeometricObject**
- 👁 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**.

`__str__` is a special method used to represent the object

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



Exercise: override the `__str__` method

For the class `rectangle`, override the class `__str__` so that when I print a `rectangle` object it says "Rectangle of area 4 and perimeter 8"

- 👁 Solution using "+" concatenation
- 👁 Solution using `format()`

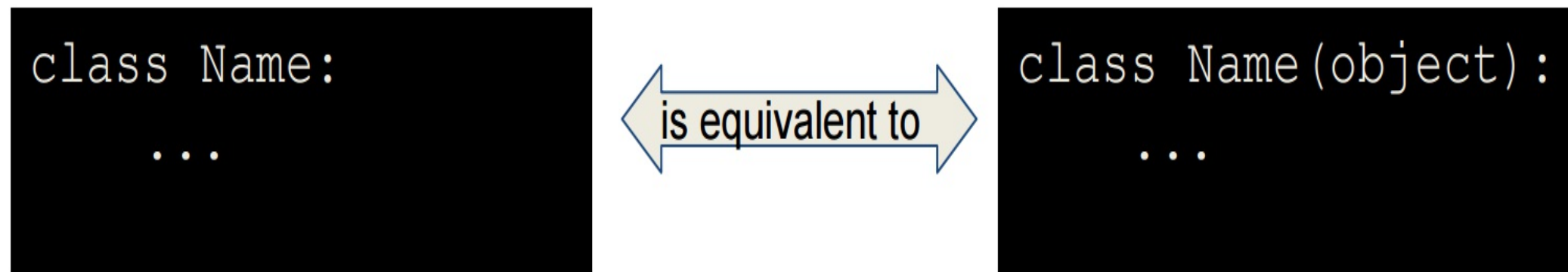


Learn more about string formatting

Many ways to format strings in python, see this website:
<https://pyformat.info/>

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.



- 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.

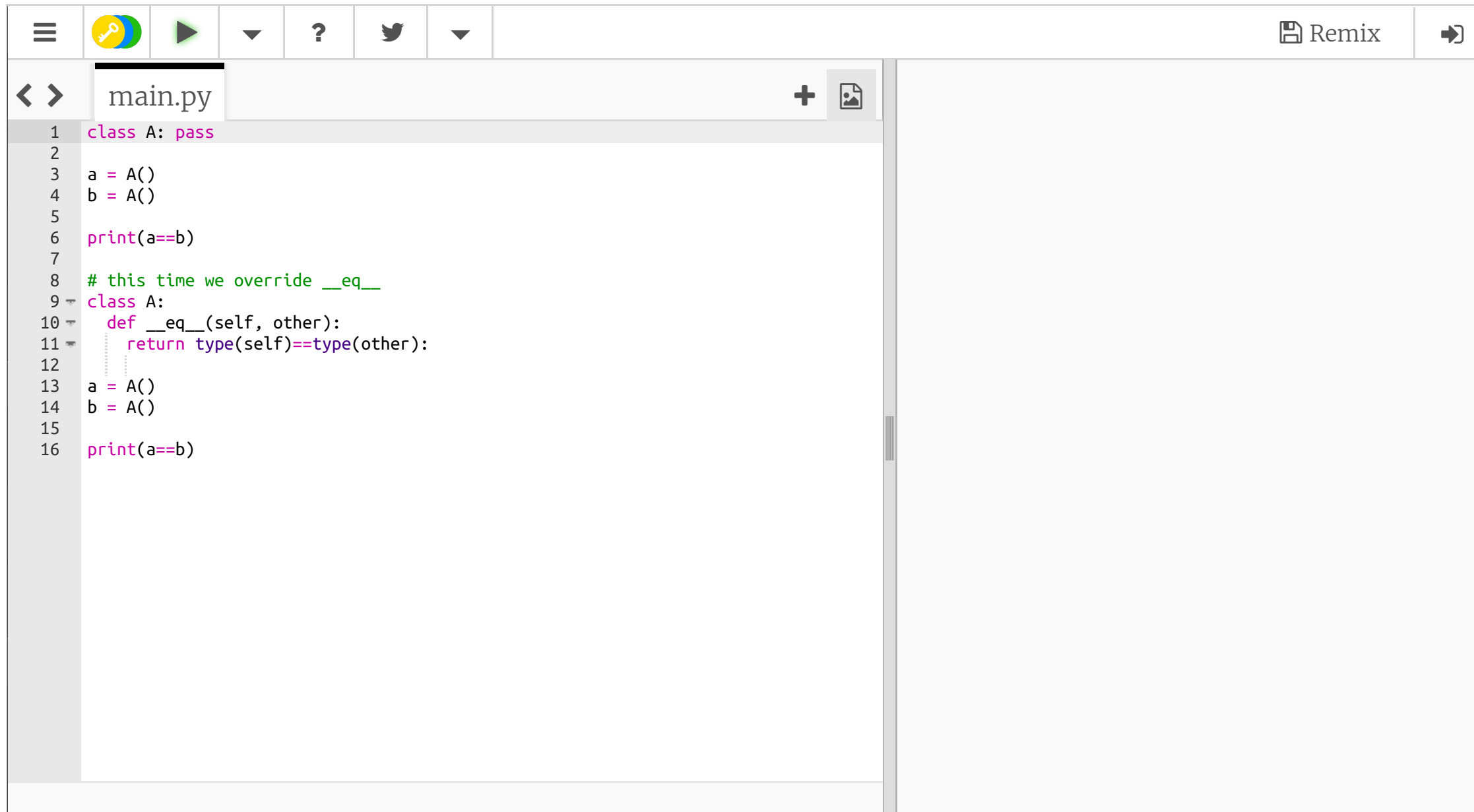
```
def __str__(self):  
    return "color: " + self.__color + \  
        " and filled: " + str(self.__filled)
```



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__`



```
1 class A: pass
2
3 a = A()
4 b = A()
5
6 print(a==b)
7
8 # this time we override __eq__
9 class A:
10     def __eq__(self, other):
11         return type(self)==type(other):
12
13 a = A()
14 b = A()
15
16 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](#)



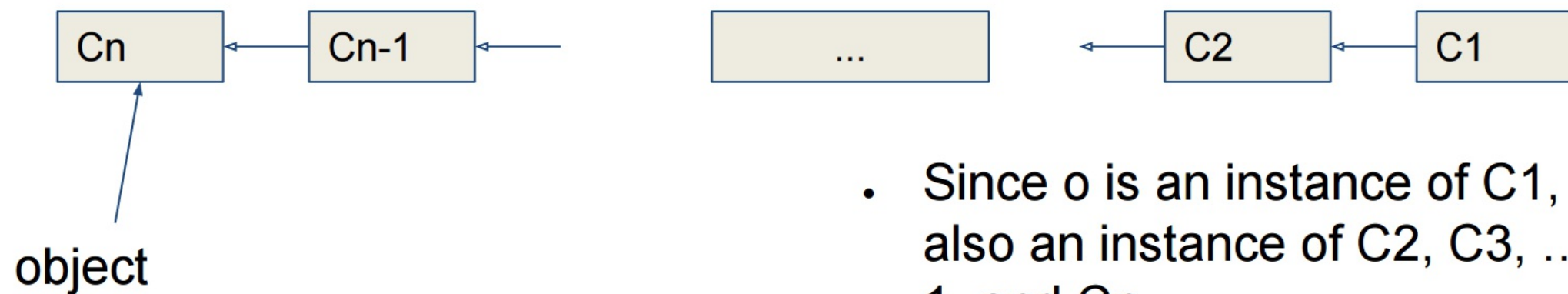
Exercise: Polymorphism

- Create 3 classes in a polymorphic way
- Complete the following script:
 - [Animals_incomplete.py](#)

 Solution

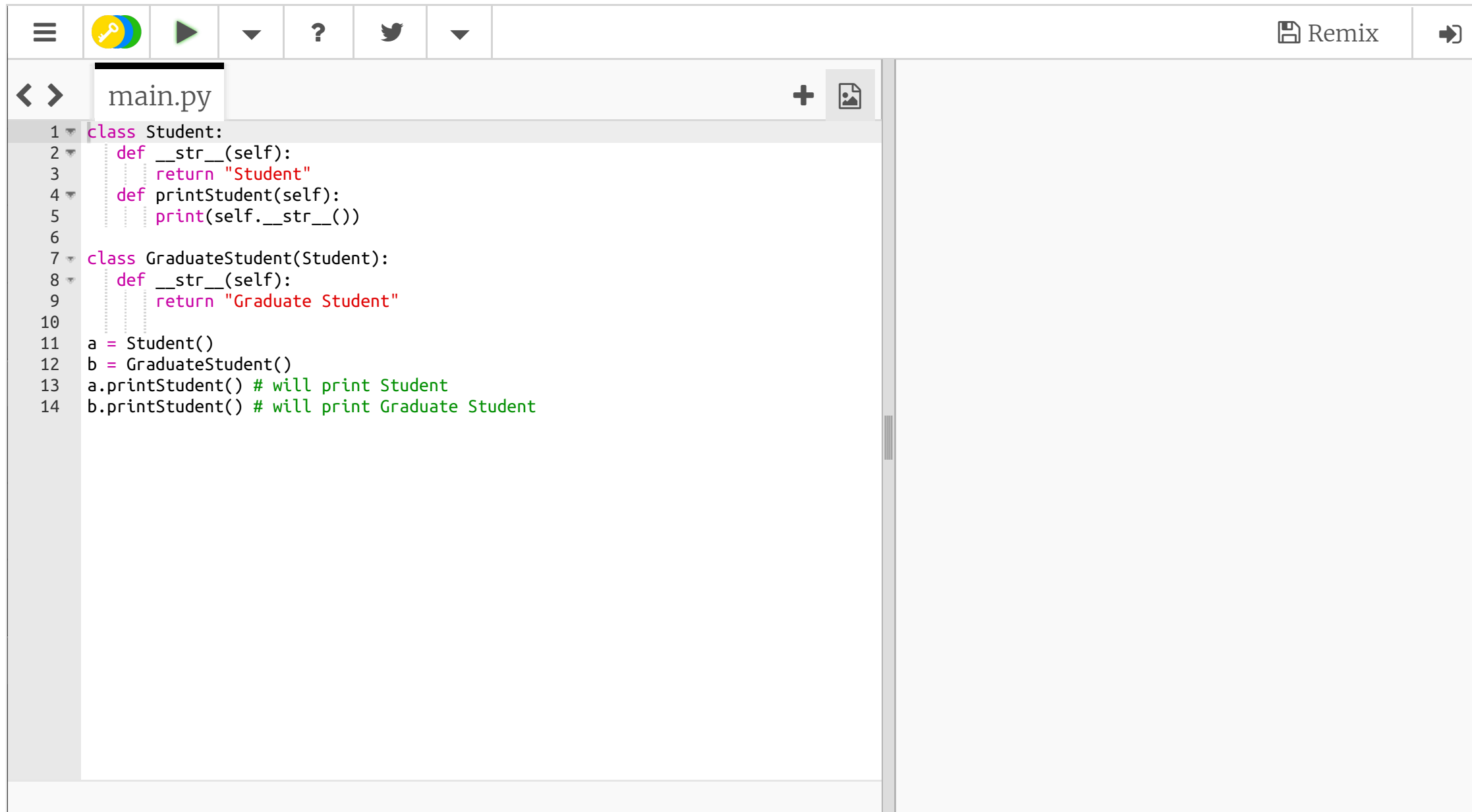
Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, \dots, C_{n-1} , and C_n , where C_1 is a subclass of C_2 , C_2 is a subclass of C_3 , ..., and C_{n-1} is a subclass of C_n . That is, C_n is the most general class, and C_1 is the most specific class. In Python, C_n is the object class. If o invokes a method p , Python searches the implementation for the method p in C_1, C_2, \dots, C_{n-1} and C_n , 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 C_1 , o is also an instance of C_2, C_3, \dots, C_{n-1} , and C_n

Dynamic Binding: example



```
1 class Student:
2     def __str__(self):
3         return "Student"
4     def printStudent(self):
5         print(self.__str__())
6
7 class GraduateStudent(Student):
8     def __str__(self):
9         return "Graduate Student"
10
11 a = Student()
12 b = GraduateStudent()
13 a.printStudent() # will print Student
14 b.printStudent() # will print Graduate Student
```

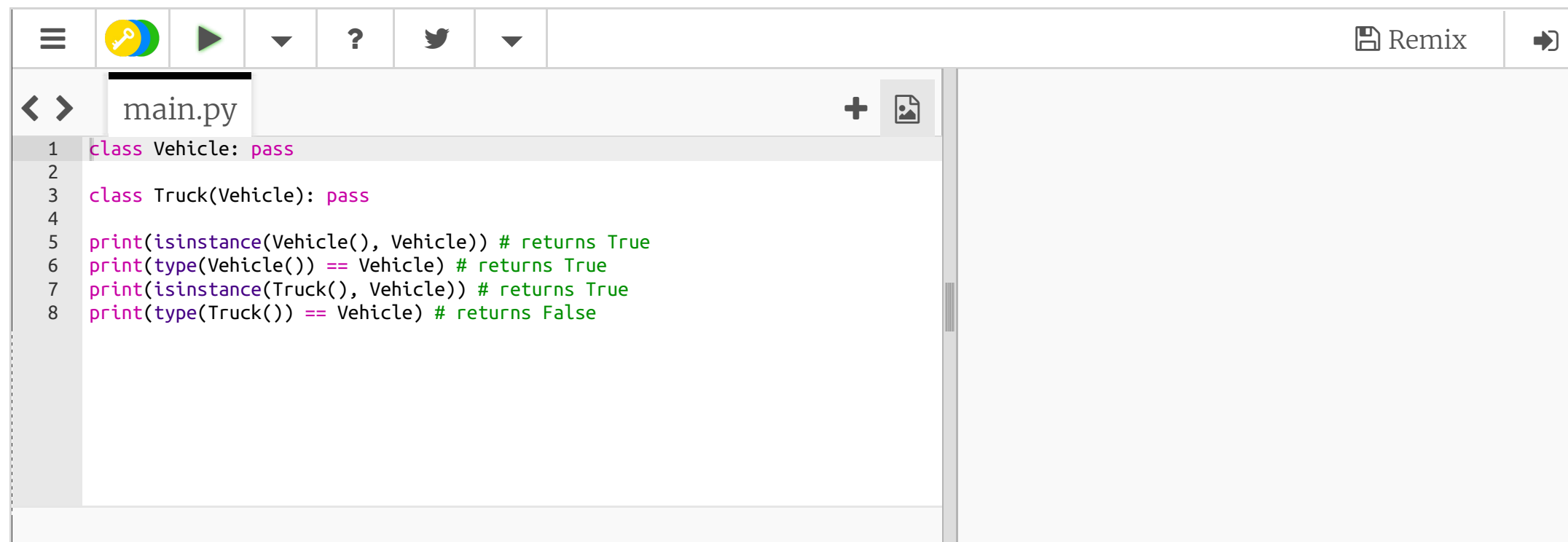


The isinstance Function

- 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



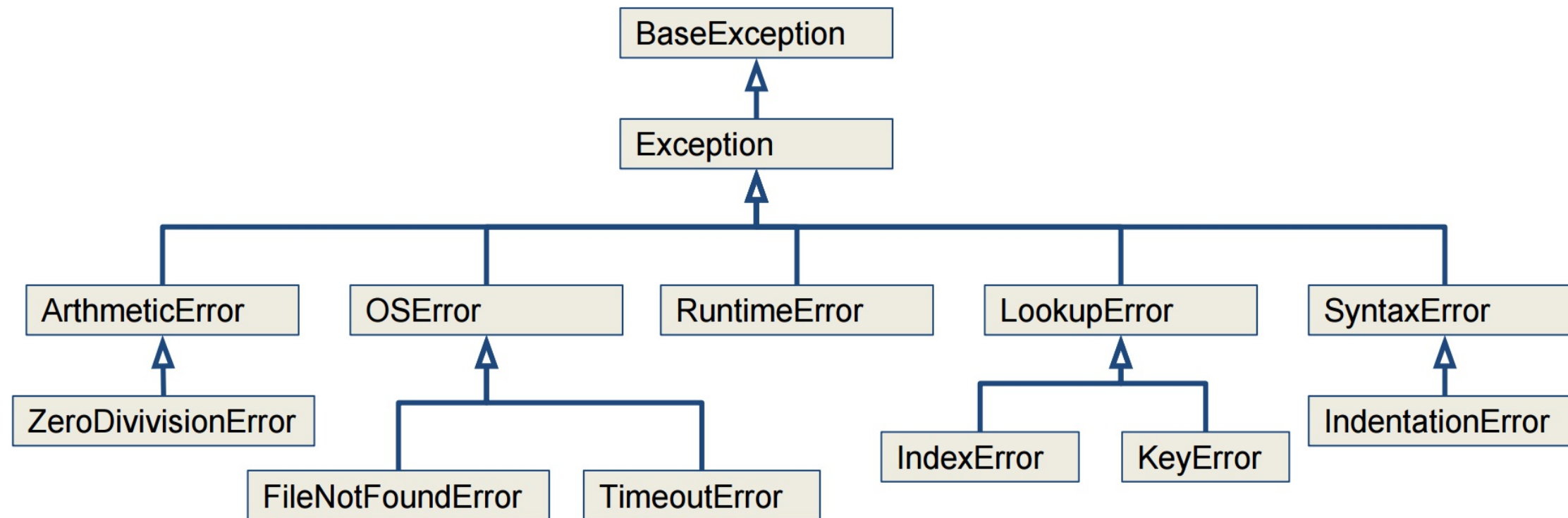
```
1 class Vehicle: pass
2
3 class Truck(Vehicle): pass
4
5 print(isinstance(Vehicle(), Vehicle)) # returns True
6 print(type(Vehicle()) == Vehicle) # returns True
7 print(isinstance(Truck(), Vehicle)) # returns True
8 print(type(Truck()) == Vehicle) # returns False
```

- 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
```

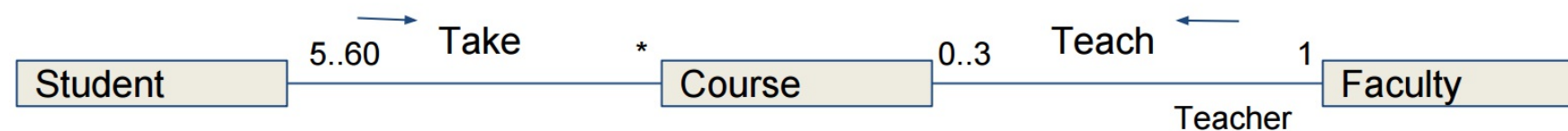
Properties should only be used in cases where you actually need to perform extra processing on attribute access

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
```

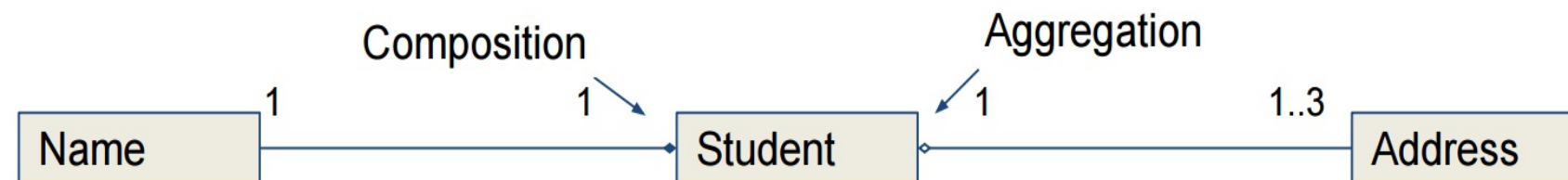
```
class Course:
    def addStudent(self,
        student):
        # add student
        # to a list
    def setFaculty(self,
        faculty):
```

```
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.



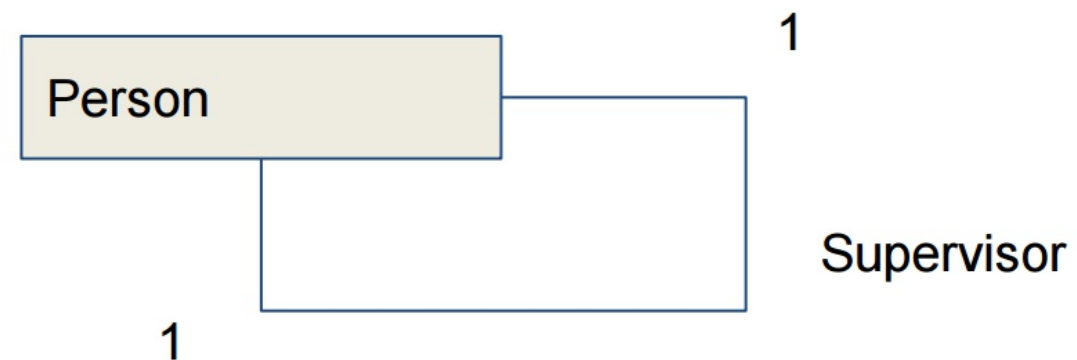
```
class Name:  
    ...
```

```
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
```

- Note: as soon as we explicitly inherits from a class, we are in the case of multiple inheritance because in python all classes inherits from **object** by default

Exercises on Inheritance and OOP

- The Account class
- Game: ATM Machine



Exercise: The Account class (1)

Design a class named Account that contains:

- A private int data field named id for the account.
- A private float data field named balance for the account.
- A private float data field named annualInterestRate that stores the current interest rate.
- A constructor that creates an account with the specified id (default 0), initial balance (default 100), and annual interest rate (default 0).
- The accessor and mutator methods for id, balance, and annualInterestRate.
- A method named getMonthlyInterestRate() that returns the monthly interest rate.
- A method named getMonthlyInterest() that returns the monthly interest.
- A method named withdraw that withdraws a specified amount from the account.
- A method named deposit that deposits a specified amount to the account.

NB: for making fields private, use [this](#), remember that you can also use [properties](#)



Exercise: The Account class (2)

Draw the UML diagram for the class, and then implement the class. (Hint: The method `getMonthlyInterest()` is to return the monthly interest amount, not the interest rate. Use this formula to calculate the monthly interest: $\text{balance} * \text{monthlyInterestRate}$.)

`monthlyInterestRate` is $\text{annualInterestRate} / 12$. Note that `annualInterestRate` is a percent (like 4.5%). You need to divide it by 100.)

Write a test program that creates an `Account` object with an account id of 1122, a balance of £20,000, and an annual interest rate of 4.5%. Use the `withdraw` method to withdraw £2,500, use the `deposit` method to deposit £3,000, and print the id, balance, monthly interest rate, and monthly interest.



Exercise on Inheritance and OOP

Use the Account class created in the previous exercise to simulate an ATM machine. Create ten accounts in a list with the ids 0, 1, ..., 9, and an initial balance of £100. The system prompts the user to enter an id. If the id is entered incorrectly, ask the user to enter a correct id. Once an id is accepted, the main menu is displayed as shown in the sample run. You can enter a choice of 1 for viewing the current balance, 2 for withdrawing money, 3 for depositing money, and 4 for exiting the main menu. Once you exit, the system will prompt for an id again. So, once the system starts, it won't stop.

Solution OOP

👁 Show solution

You can write a better program 😊