Inheritance

Inheritance

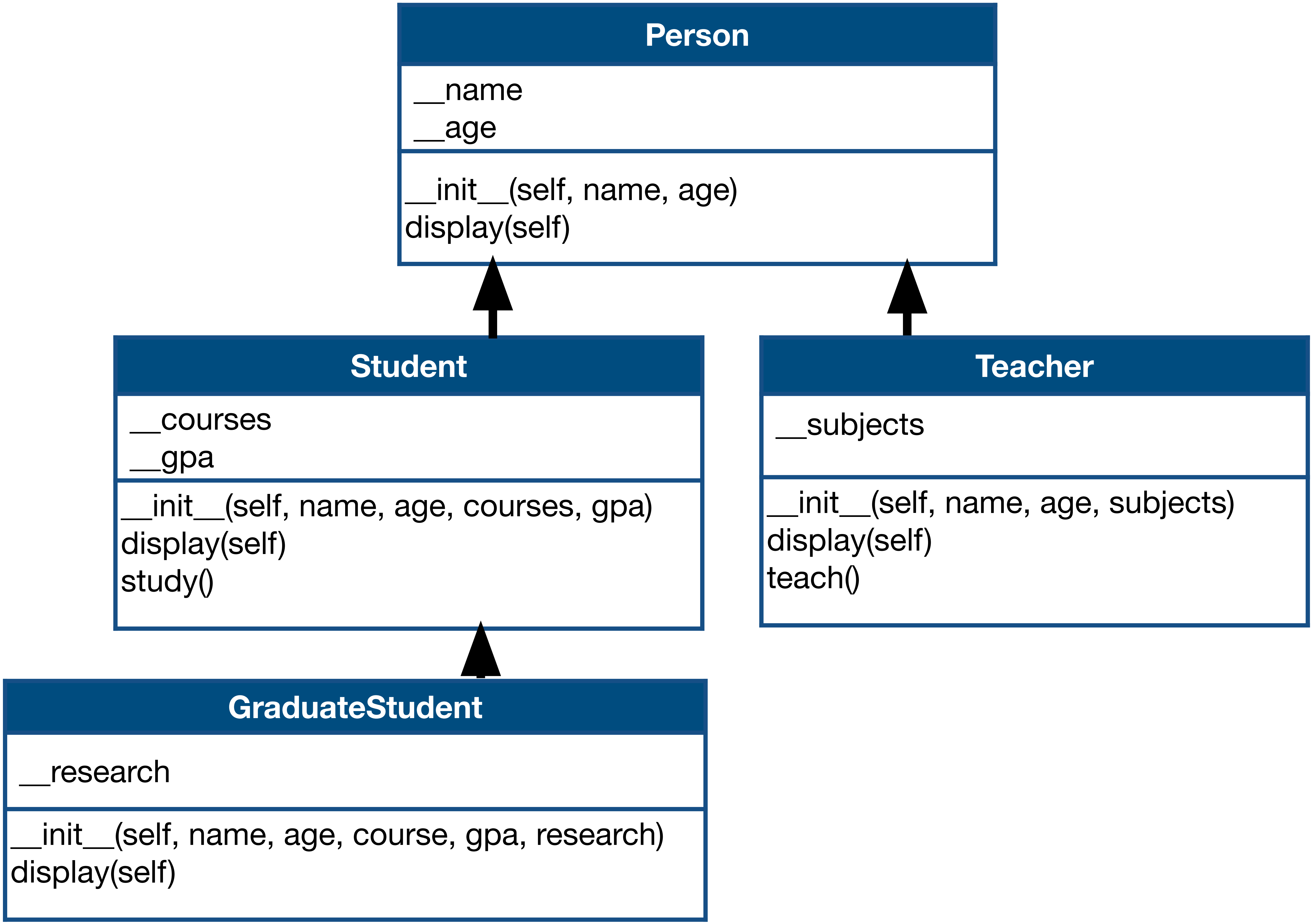
- Inheritance allows you to create a new class based on an existing class. The new class inherits the attributes and methods of the existing class.

- A base class known as parent class or superclass is the class that another class inherits.

- A derived class known as child class or subclass that inherits another class.

- A subclass can add new attributes and methods to the superclasss. It can also override a method from the superclass by providing its own implementation.

Inheritance



Person

__name
__age

__init__(self, name, age)
display(self)

Student

__courses
__gpa

__init__(self, name, age, courses, gpa)
display(self)
study()

Teacher

__subjects

__init__(self, name, age, subjects)
display(self)
teach()

GraduateStudent

__research

__init__(self, name, age, course, gpa, research)
display(self)

Defining A Subclass

- A subclass can directly access public attributes of the superclass
- You can add new attributes and methods that aren't in the superclass
- You can call methods of the superclass including constructors and properties by coding the name of the superclass, the dot operator, and the name of the methods.
- You can override existing methods in the superclass by coding methods that have the same name.
- The syntax for working with subclasses
  - To define a subclass

    class SubClassName(SuperClassName)
  - To call a method or constructor of the superclass

    SuperClassName.methodName(self[, parameterList])

Example of Defining A Subclass

```
class Product:
    def __init__(self, product_name = "", price = 0.0, rebate = 0.0, warranty = 1):
        self.__product_name = product_name
        self.__price = price
        self.__rebate = rebate
        self.__warranty = warranty

       def display(self):
               print("Product name:", self.__product_name)
               print("Price:", self.__price)
               print("Rebate:", self.__rebate)
               print("Warranty:", self.__warranty)

class Computer(Product):
    def __init__(self, computer_type, product_name = "", price = 0.0, rebate = 0.0, warranty = 1):
        super().__init__(product_name, price, rebate, warranty)
        self.__computer_type = computer_type

    def display(self):
        super().display()
        print("Computer type:", self.__computer_type)

class Car(Product):
    def __init__(self, horsepower, product_name = "", price = 0.0, rebate = 0.0, warranty = 1):
        Product.__init__(self, product_name, price, rebate, warranty)
        self.__horsepower = horsepower

    def display(self):
        super().display()
        print("Horsepower:", self.__horsepower)
```

Polymorphism

- Polymorphism is a feature of inheritance that lets you treat objects of subclasses as if they were objects of the superclass.
- If you access a method of a superclass object and the method is overridden in the subclasses of that class, polymorphism determines which method is executed based on the object’s type.

```
def main():
    products = []
    products.append(Computer("Desktop", "Computer", 1000.00, 12.5, 2))
    products.append(Car(300.0, "Car", 50000.00, 0.5, 5))

    for product in products:
        product.display()
        print()
```

Checking An Object’s Type

- The isinstance() function that returns True if the object is an instance of the specified class. Otherwise, returns False.
- You can use it to perform different processing for different types of objects.

  isinstance(object, [moduleName.]ClasssName)

```
def display_product(product):
    print("PRODUCT DATA")
    print("Product Name:", product.product_name)
    if isinstance(product, Computer):
        print("Computer Type:", product.computer_type)
    if isinstance(product, Car):
        print("Horsepower:", product.housepower)
    print("Best price:   {:.2f}".format(product.calculate_price()))
    print()
```

The object Class

- The object class is the superclass for all classes. Therefore, every class inherits the object class. As a result, the methods defined by the object class are available to all classes.
- The __str__() method is a special method that's automatically called whenever an object needs to be converted to a string such as when the print statement prints an object to the console or when the str() function attempts to convert an object to a string.
- The __str__() method in the object class returns a message that includes the name of the class for the object as well as its identifier. If that's not what you want, you can override this behavior by defining your own __str__() method.
- When coding classes, you often want to override the __str__() method so it returns a string that's concise, informative, and easy to read.

You can use __dir__ method to list all attributes and methods in object class

```
>>> object.__dir__(object)
['__repr__', '__call__', '__getattribute__', '__setattr__', '__delattr__', '__init__', '__new__', 'mro', '__subclasses__', '__prepare__', '__instancec
heck__', '__subclasscheck__', '__dir__', '__sizeof__', '__basicsize__', '__itemsize__', '__flags__', '__weakrefoffset__', '__base__',
'__dictoffset__'
, '__mro__', '__name__', '__qualname__', '__bases__', '__module__', '__abstractmethods__', '__dict__', '__doc__', '__text_signature__',
'__hash__', '_
_str__', '__lt__', '__le__', '__eq__', '__ne__', '__gt__', '__ge__', '__reduce_ex__', '__reduce__', '__subclasshook__', '__init_subclass__',
'__format
__', '__class__']
```

Defining Iterators for Your Classes

- When having a container in your classes, we usually want to overrode the __iter__() and __next__() methods so other classes can iterate through the objects within the container. The iterator can help to improve the encapsulation of the objects and makes it easier for other programmers to use the object.
- It's common to use an iterator to provide a public way to access the objects that are stored in a private attribute of an object.
- The __iter__(self) returns the iterator for the object and initializes the index for the iterator.
- The __next__(self) returns the next object in the sequence of objects. If there are no more objects, this method should raise the StopIteration exception.

Example of defining Iterators for Your Classes

```
class Bookstore:
    def __init__(self, store_name):
        self.__list = []
        self.__store_name = store_name

    def add_book(self, book):
        self.__list.append(book)

    def displayall(self):
        for book in self.__list:
            book.display()

    # define the Book object as the iterator
    def __iter__(self):
        self.__index = -1   # initialize index for each iteration
        return self

    # define the method that gets the next object
    def __next__(self):
        if self.__index >= len(self.__list)-1:
            raise StopIteration()
        self.__index += 1
        book = self.__list[self.__index]
        return book
```

Inheritance vs Aggregation

- If an object is a type of another object, it typically makes senses to use inheritance to create the relationship between the two classes.
    - The subclass primary adds features to the superclass.
- If an object has a type of another object, it typically makes sense to use aggregation to create the relationship between the two classes.

Python - Public, Protected, Private Member

- Public attributes and methods of a class are accessible from outside the class.
- Protected attributes and methods of a class are accessible from within the class and are also available to its sub-classes. Python's convention to make an instance variable protected is to add a prefix _ (single underscore) to it.
    - However, this doesn't prevent instance variables from accessing or modifying the instance. Therefore, it becomes programmer’s responsibility for not accessing and modifying instance variables prefixed with _ from outside its class
- The private members of a class are only accessible within the class. In Python, a private member can be defined by using a prefix __ (double underscore).

Exercise - This program contains errors. Comment out the errors so it can be compiled and executed to produce the expected output.

```
class ClassA:
    def __init__(self, a, b, c):
        self.a = a          # public attribute
        self._b = b         # protected attribute
        self.__c = c        # private attribute

    def method_a(self):      # public method
        print("method_a =", self.a);

    def _method_b(self):     # protected method
        print("method_b =", self._b);

    def __method_c(self):    # private method
        print("method_c =", self.__c);

class ClassB(ClassA):
    def __init__(self, a, b, c, d):
        ClassA.__init__(self, a, b, c)
        self._d = d         # protected attribute

    def method_d(self):      # protected method
        print("method_d =", self._d);

    def method_test(self):   # protected method
        # These two attrobute are allowed
        print("a =", self.a)
        print("b =", self._b)
        print("c =", self.__c)

        # these two method calls are allowed
        ClassA.method_a(self)
        ClassA._method_b(self)
        ClassA.__method_c(self)

def main():
    bobj = ClassB(2, 4, 6, 8)
    bobj.method_test()

    print("bobj.a =", bobj.a)
    print("bobj._b =", bobj._b)
    print("bobj.__c =", bobj.__c)

    bobj.method_a()
    bobj._method_b()
    bobj.__method_c()
    bobj.method_d()
if __name__ == "__main__":
    main()
```

Python super()

- The super() function is used to give access to methods and properties of a parent or sibling class.

Without using super()

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    @property
    def name(self):
        return self.__name

    @property
    def age(self):
        return self.__age

    def display(self):
        print("name =", self.__name)
        print("age =", self.__age)

class Student(Person):
    def __init__(self, name, age, gpa):
        Person.__init__(self, name, age)
        self.__gpa = gpa

    @property
    def gpa(self):
        return self.__gpa

    def display(self):
        Person.display(self)
        print("gpa =", self.__gpa)

    def study(self):
        print(self.name, "has been studying so hard")
```

Using super()

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    @property
    def name(self):
        return self.__name

    @property
    def age(self):
        return self.__age

    def display(self):
        print("name =", self.__name)
        print("age =", self.__age)

class Student(Person):
    def __init__(self, name, age, gpa):
        super().__init__( name, age)
        self.__gpa = gpa

    @property
    def gpa(self):
        return self.__gpa

    def display(self):
        super().display()
        print("gpa =", self.__gpa)

    def study(self):
        print(self.name, "has been studying so hard")
```

vars() and dir()

- The vars() function returns the __dic__ attribute of an object.
  - The __dict__ attribute is a dictionary containing the object's changeable attributes.
- The dir() function returns all properties and methods of the specified object, without the values.
  - This function will return all the properties and methods, even built-in properties which are default for all object.

```
>> from person import Student
>>> s = Student('Peter', 20, 3.5)
>>> vars(s)
{'_Person__name': 'Peter', '_Person__age': 20, '_Student__gpa': 3.5}
>>> dir(s)
['_Person__age', '_Person__name', '_Student__gpa', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'age', 'display', 'gpa', 'name', 'study']
>>> dir(Student)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'age', 'display',
'gpa', 'name', 'study']
>>> vars(Student)
mappingproxy({'__module__': 'person', '__init__': <function Student.__init__ at 0xf7bf0c88>, 'gpa': <property object at
0xf7bf3f50>, 'display': <function Student.display at 0xf7bf0d18>, 'study': <function Student.study at 0xf7bf0d60>,
'__doc__': None})
```