Chapter 7: Python OOP Inheritance and Composition

Contents

Red light: No AI

Time required: 90 minutes

DRY

Don't **R**epeat **Y**ourself

Tutorials

Please go through the following tutorials.

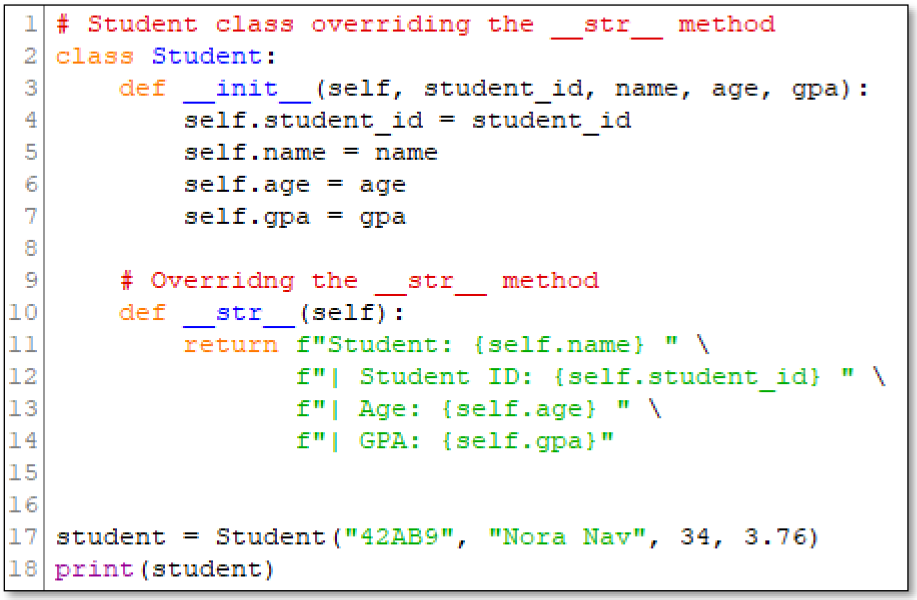
- https://www.w3schools.com/python/python_inheritance.asp

Special Methods

The __str__ method is called a dunder (double underscore) method. __str__ is the string representation of an object. By default, it returns the memory address. It would be better to return a representation of the object. Let's override the __str__ method to display relevant information about the object.

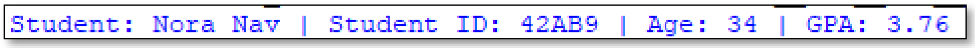
Tutorial 7.1: Student Override __str__

Student example:



```python
# Student class overriding the __str__ method
class Student:
    def __init__(self, student_id, name, age, gpa):
        self.student_id = student_id
        self.name = name
        self.age = age
        self.gpa = gpa

    # Overridng the __str__ method
    def __str__(self):
        return f"Student: {self.name} " \
               f"| Student ID: {self.student_id} " \
               f"| Age: {self.age} " \
               f"| GPA: {self.gpa}"


student = Student("42AB9", "Nora Nav", 34, 3.76)
print(student)
```

Example run:



```
Student: Nora Nav | Student ID: 42AB9 | Age: 34 | GPA: 3.76
```

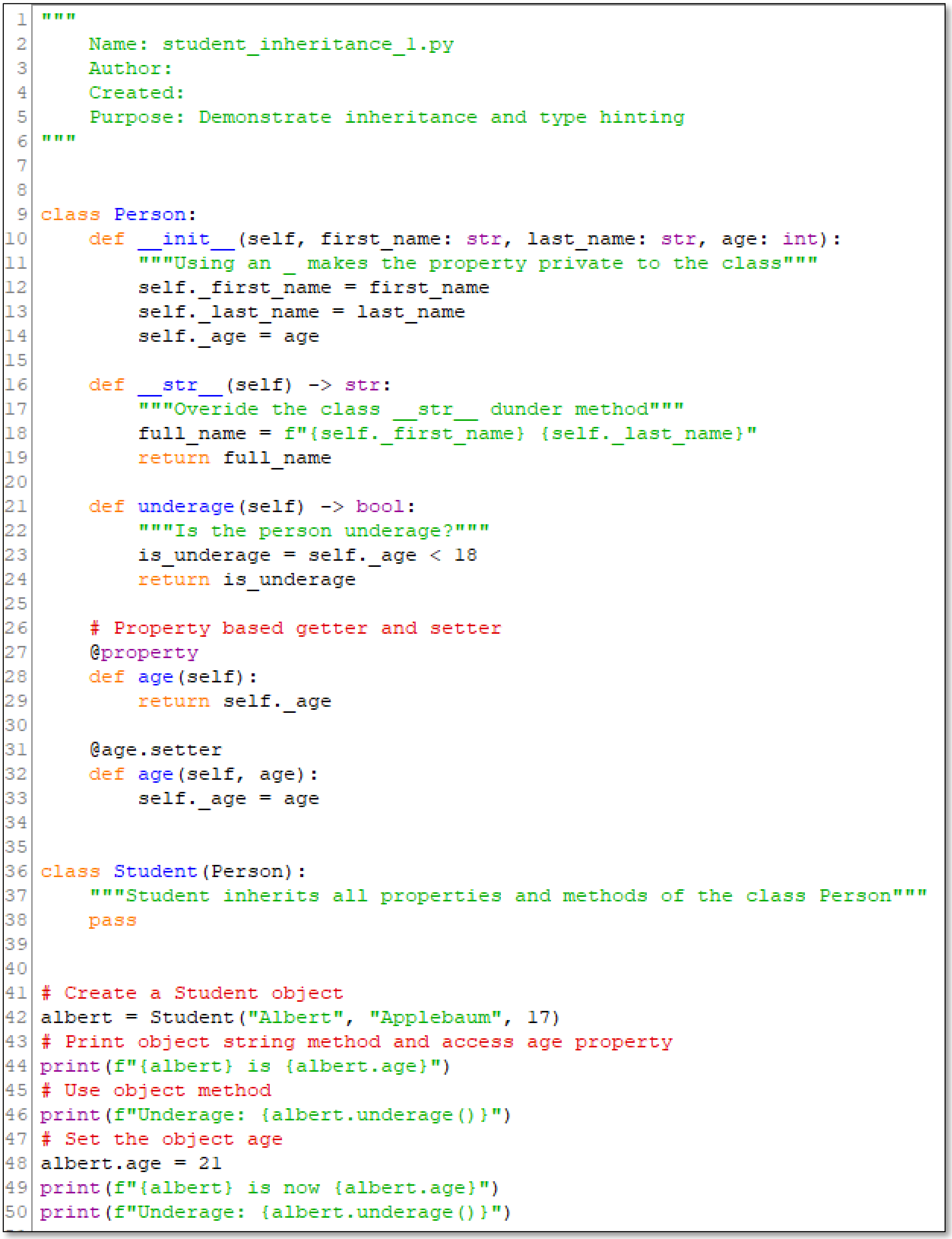
Tutorial 7.2: Inheritance

A powerful feature of object-oriented programming is the ability to create a new class by extending an existing class. When extending a class, we call the original class the parent class and the new class the child class.

An inherited class builds from another class. When you do this, the new class gets all the variables and methods of the class it is inheriting from (called the base class). It can then define additional variables and methods that are not present in the base class, and it can also override some of the methods of the base class. It can rewrite them to suit its own purposes.

When you define a new class, between parentheses you can specify another class. The new class inherits all the attributes and methods of the other class, i.e, they are automatically part of the new class.

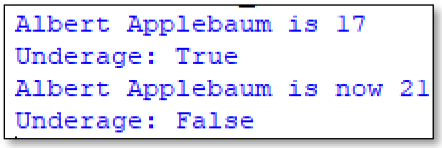
The following is an example of a parent Person class with a child student class.



```python
"""
    Name: student_inheritance_1.py
    Author:
    Created:
    Purpose: Demonstrate inheritance and type hinting
"""


class Person:
    def __init__(self, first_name: str, last_name: str, age: int):
        """Using an _ makes the property private to the class"""
        self._first_name = first_name
        self._last_name = last_name
        self._age = age

    def __str__(self) -> str:
        """Overide the class __str__ dunder method"""
        full_name = f"{self._first_name} {self._last_name}"
        return full_name

    def underage(self) -> bool:
        """Is the person underage?"""
        is_underage = self._age < 18
        return is_underage

    # Property based getter and setter
    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, age):
        self._age = age


class Student(Person):
    """Student inherits all properties and methods of the class Person"""
    pass


# Create a Student object
albert = Student("Albert", "Applebaum", 17)
# Print object string method and access age property
print(f"{albert} is {albert.age}")
# Use object method
print(f"Underage: {albert.underage()}")
# Set the object age
albert.age = 21
print(f"{albert} is now {albert.age}")
print(f"Underage: {albert.underage()}")
```

The Student class inherits all properties and methods of the class Person.

Example run:

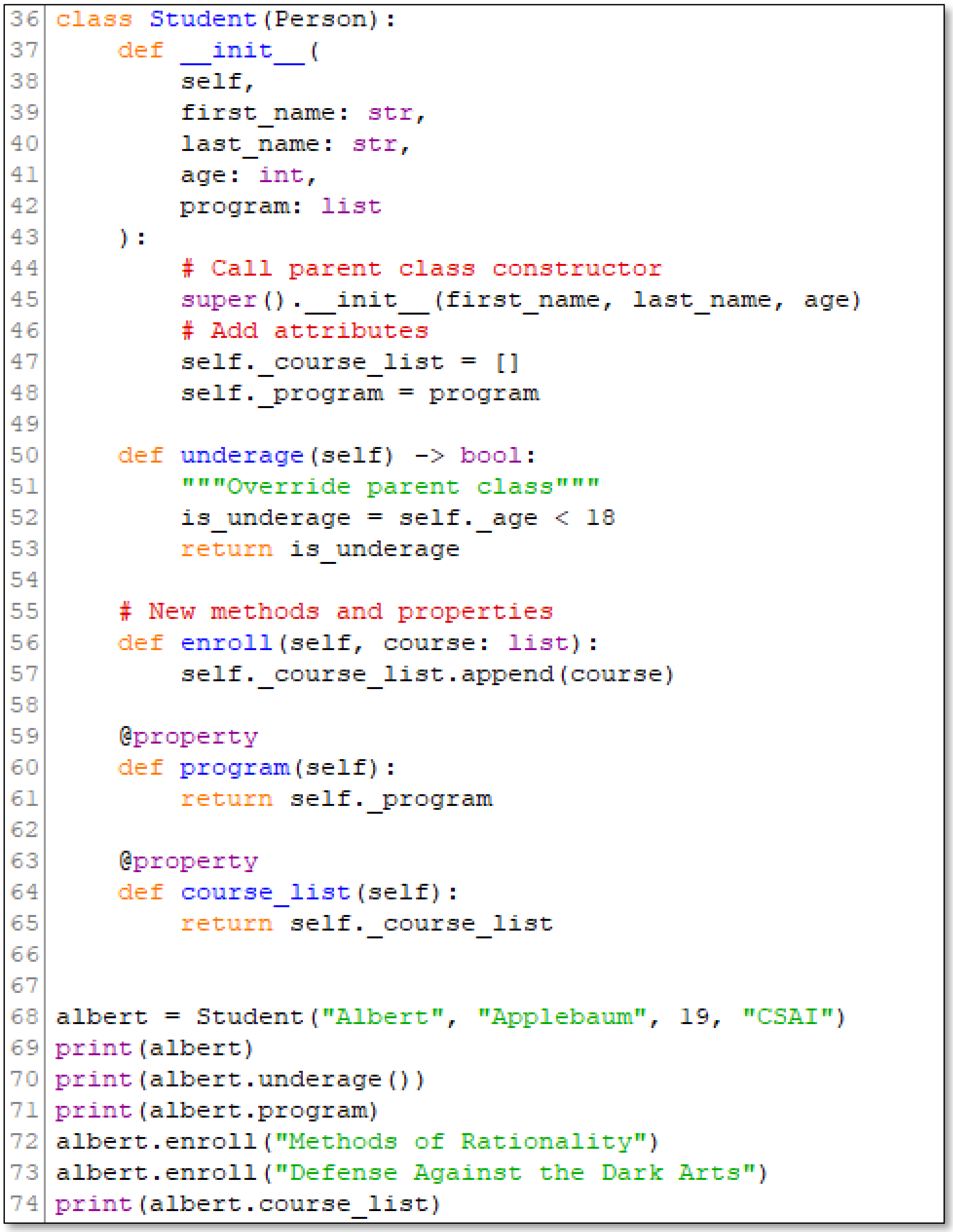


```
Albert Applebaum is 17
Underage: True
Albert Applebaum is now 21
Underage: False
```

Tutorial 7.3: Inheritance Implementation

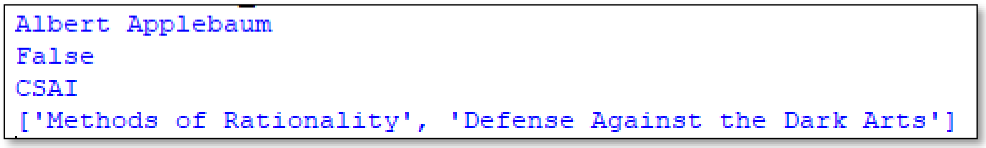
In the code below, the class Student gets two new attributes: a program and a course list. The method __init__() gets overridden to create these new attributes, but also calls the __init__() method of Person. Student gets a new method, enroll(), to add courses to the course list. We overrode the method underage() to make students underage when they are not 21 yet (sorry about that).

The people class remains the same as the previous tutorial.



```python
class Student(Person):
    def __init__(
        self,
        first_name: str,
        last_name: str,
        age: int,
        program: list
    ):
        # Call parent class constructor
        super().__init__(first_name, last_name, age)
        # Add attributes
        self._course_list = []
        self._program = program

    def underage(self) -> bool:
        """Override parent class"""
        is_underage = self._age < 18
        return is_underage

    # New methods and properties
    def enroll(self, course: list):
        self._course_list.append(course)

    @property
    def program(self):
        return self._program

    @property
    def course_list(self):
        return self._course_list


albert = Student("Albert", "Applebaum", 19, "CSAI")
print(albert)
print(albert.underage())
print(albert.program)
albert.enroll("Methods of Rationality")
albert.enroll("Defense Against the Dark Arts")
print(albert.course_list)
```

Example run:

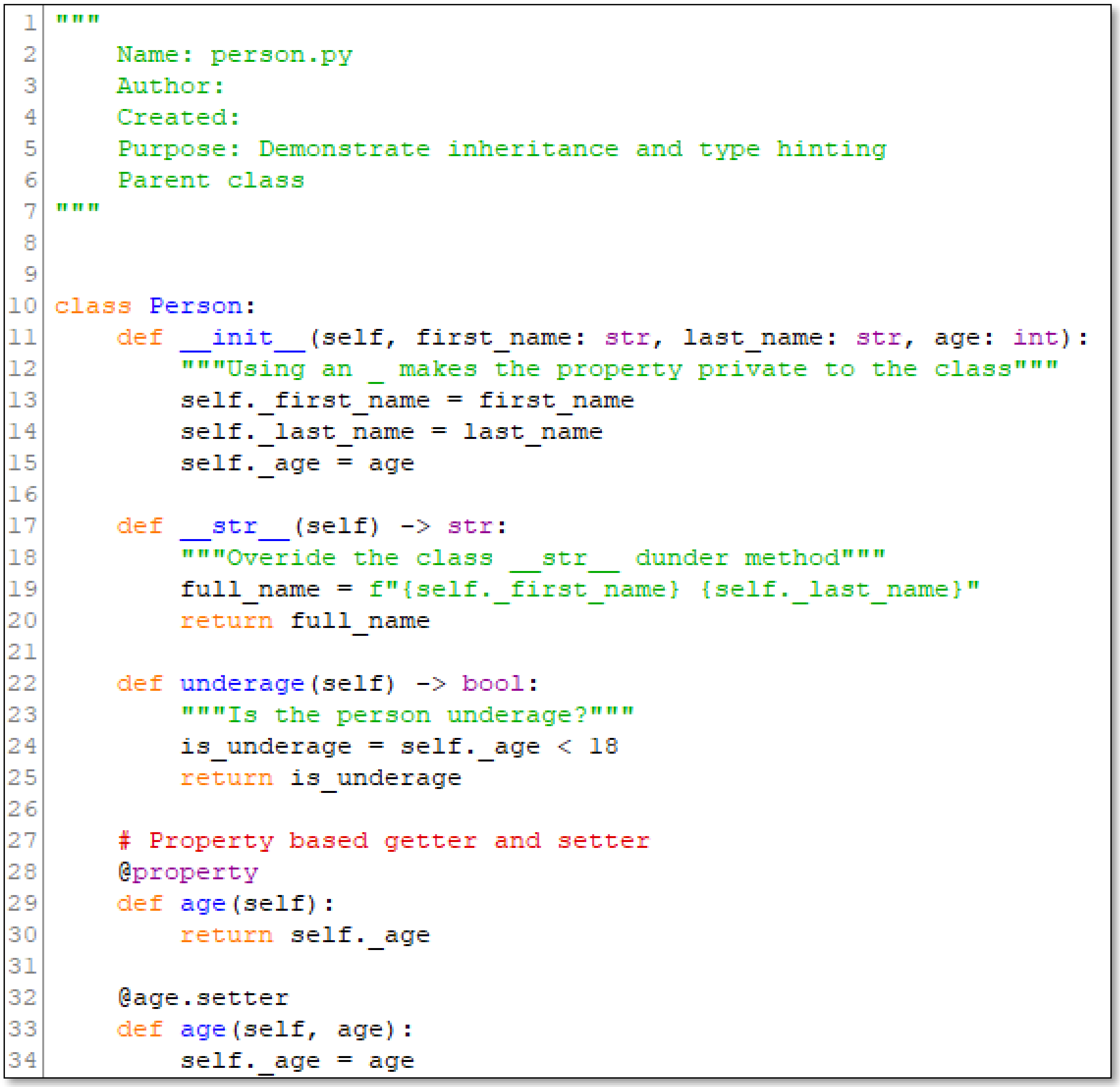


```
Albert Applebaum
False
CSAI
['Methods of Rationality', 'Defense Against the Dark Arts']
```

Tutorial 7.4: Inheritance with Multiple Files

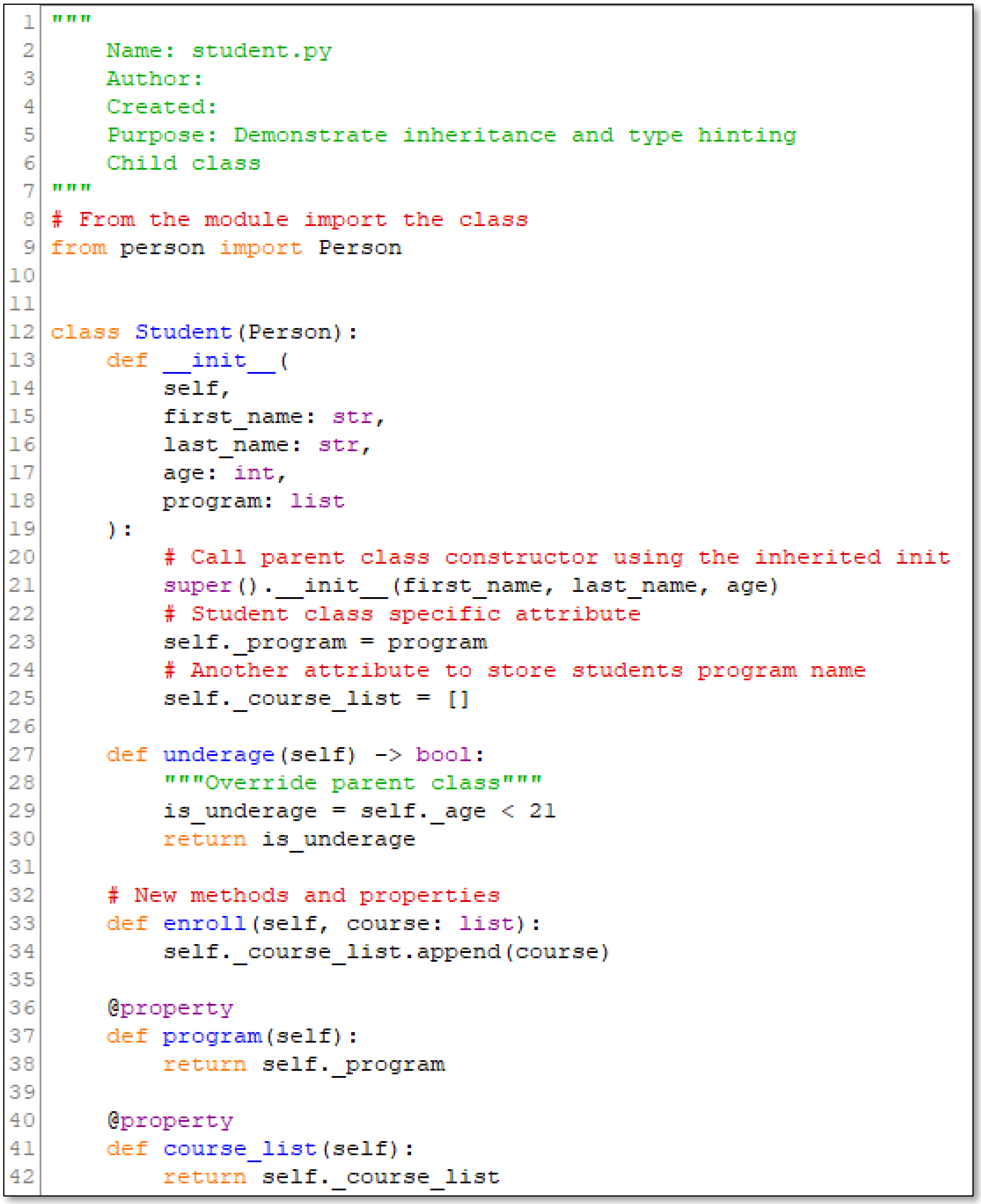
Let’s take the same program code and split it into two class files and an application file. This is a better implementation, especially with larger programs.

person.py



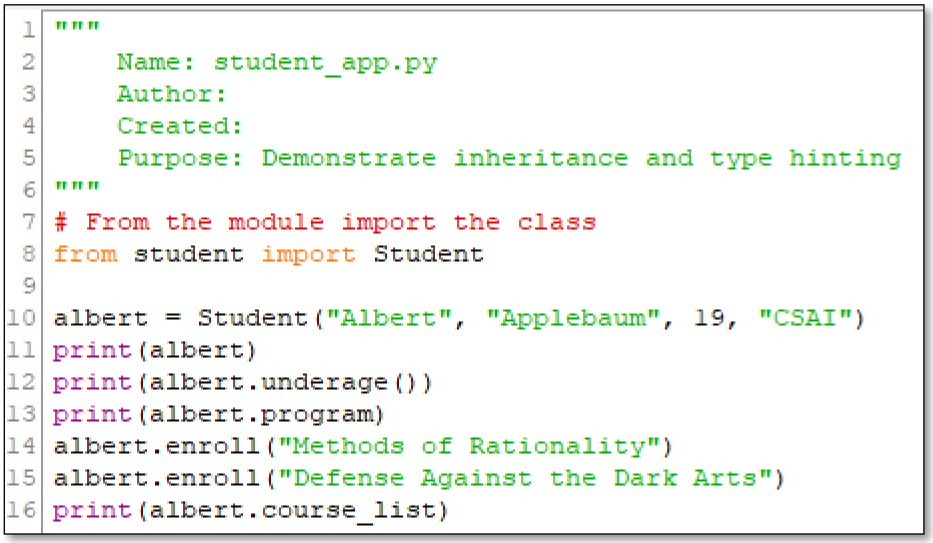
```python
"""
    Name: person.py
    Author:
    Created:
    Purpose: Demonstrate inheritance and type hinting
    Parent class
"""


class Person:
    def __init__(self, first_name: str, last_name: str, age: int):
        """Using an _ makes the property private to the class"""
        self._first_name = first_name
        self._last_name = last_name
        self._age = age

    def __str__(self) -> str:
        """Overide the class __str__ dunder method"""
        full_name = f"{self._first_name} {self._last_name}"
        return full_name

    def underage(self) -> bool:
        """Is the person underage?"""
        is_underage = self._age < 18
        return is_underage

    # Property based getter and setter
    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, age):
        self._age = age
```

student.py



```python
"""
    Name: student.py
    Author:
    Created:
    Purpose: Demonstrate inheritance and type hinting
    Child class
"""
# From the module import the class
from person import Person


class Student(Person):
    def __init__(
        self,
        first_name: str,
        last_name: str,
        age: int,
        program: list
    ):
        # Call parent class constructor using the inherited init
        super().__init__(first_name, last_name, age)
        # Student class specific attribute
        self._program = program
        # Another attribute to store students program name
        self._course_list = []

    def underage(self) -> bool:
        """Override parent class"""
        is_underage = self._age < 21
        return is_underage

    # New methods and properties
    def enroll(self, course: list):
        self._course_list.append(course)

    @property
    def program(self):
        return self._program

    @property
    def course_list(self):
        return self._course_list
```

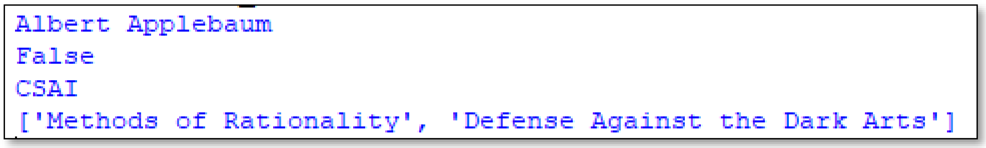
student_app.py



```python
"""
    Name: student_app.py
    Author:
    Created:
    Purpose: Demonstrate inheritance and type hinting
"""
# From the module import the class
from student import Student

albert = Student("Albert", "Applebaum", 19, "CSAI")
print(albert)
print(albert.underage())
print(albert.program)
albert.enroll("Methods of Rationality")
albert.enroll("Defense Against the Dark Arts")
print(albert.course_list)
```

The output is the same. The code base is much easier to maintain and expand.

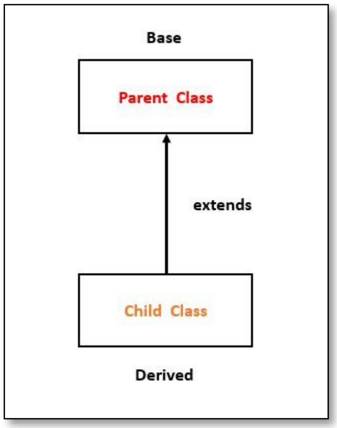
Example run:



```
Albert Applebaum
False
CSAI
['Methods of Rationality', 'Defense Against the Dark Arts']
```

Composition

"is-a" Relationship



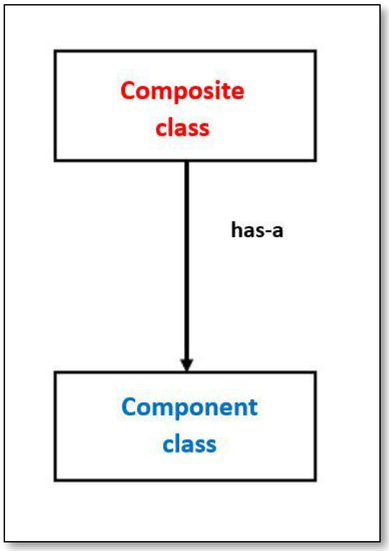
In an "is-a" relationship, one class is a subclass of another. This implies that the child class inherits the properties and behaviors of the parent class.

```
# Define a base class
class Animal:
    def __init__(self, species):
        self.species = species

    def speak(self):
        pass

# Create a subclass
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Create an instance of Dog
my_dog = Dog("Canine")
print(f"My {my_dog.species} says: {my_dog.speak()}")
```

In this example, `Dog` is a subclass of `Animal`, and it inherits the `species` attribute and the `speak()` method.

"has-a" Relationship

Watch the following video: [Python OOP Composition Clearly Explained]()



In a "has-a" relationship, one class contains an instance of another class as one of its attributes. This allows for greater flexibility and modularity in your code.

Composition allows you to create classes that are composed of other classes. These classes work together to achieve a more complex behavior.

For instance, in a gaming system, you might have a "Player" class that contains a "Weapon" class as one of its attributes. The "Player" has a "Weapon," and this relationship is an example of composition.

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

class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
        # Create a class as an attribute
        self.engine = Engine()

    def start_engine(self):
        # Use .start() from the Engine class
        self.engine.start()

# Creating a car
my_car = Car("Toyota", "Camry")
my_car.start_engine()
```

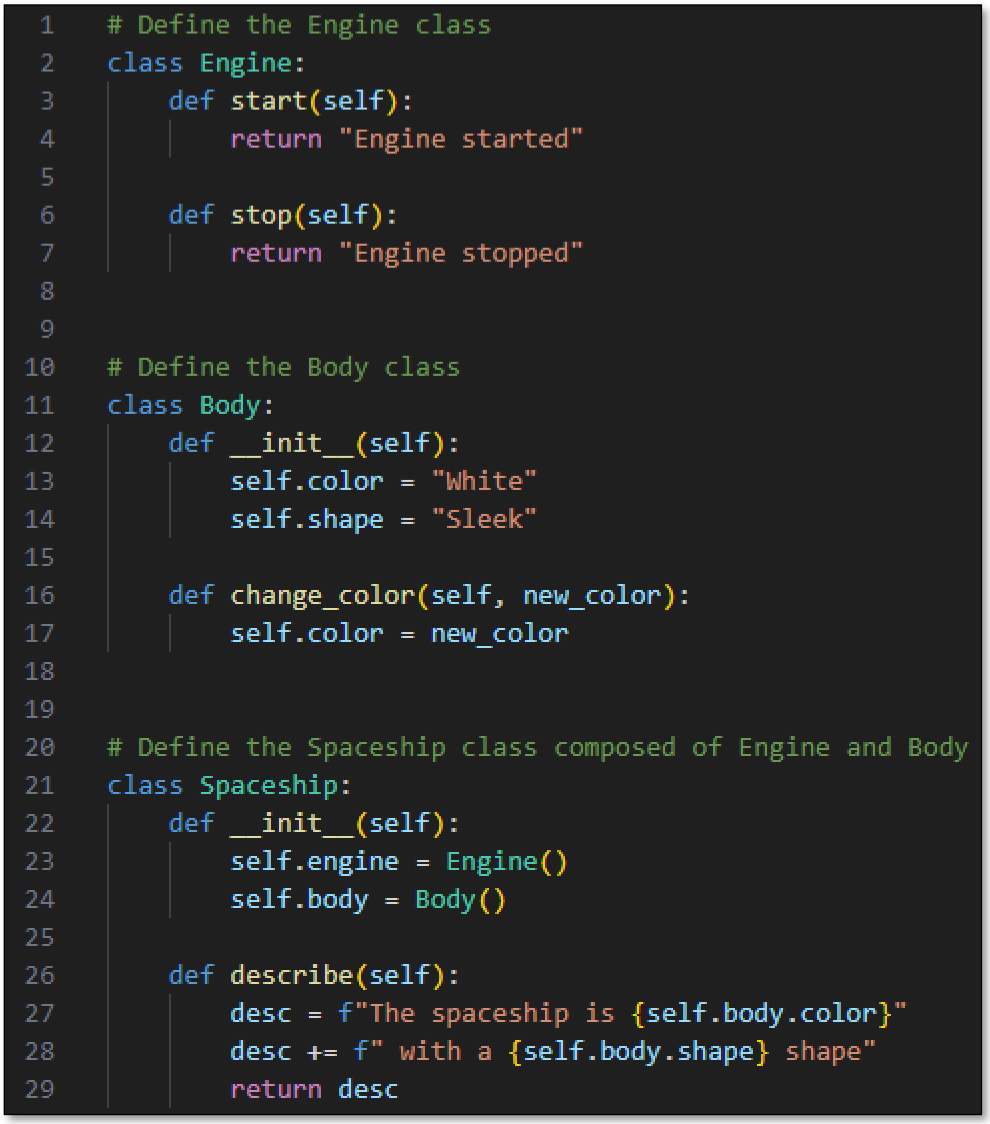
Another example.

```
# Create a class representing a Car
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

# Create a class representing a Person
class Person:
    def __init__(self, name, car):
        self.name = name
        self.car = car

    def drive(self):
        return f"{self.name} is driving a {self.car.make} {self.car.model}"

# Create instances of Car and Person
my_car = Car("Toyota", "Camry")
me = Person("John", my_car)
```

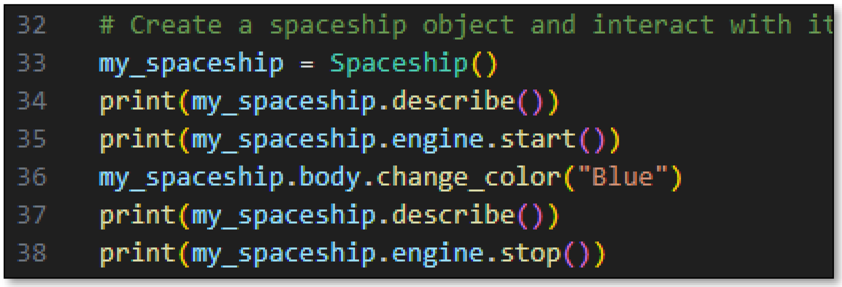
```
# Accessing the "has-a" relationship
print(me.drive())
```

In this case, the `Person` class "has-a" relationship with the `Car` class as a person can own a car. The `car` attribute of the `Person` class is an instance of the `Car` class.

Tutorial 7.5: A Composed Spaceship

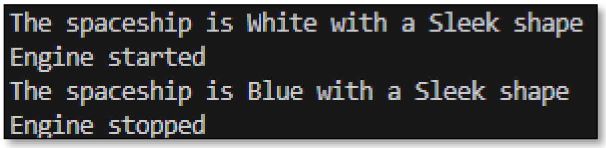


```python
# Define the Engine class
class Engine:
    def start(self):
        return "Engine started"

    def stop(self):
        return "Engine stopped"


# Define the Body class
class Body:
    def __init__(self):
        self.color = "White"
        self.shape = "Sleek"

    def change_color(self, new_color):
        self.color = new_color


# Define the Spaceship class composed of Engine and Body
class Spaceship:
    def __init__(self):
        self.engine = Engine()
        self.body = Body()

    def describe(self):
        desc = f"The spaceship is {self.body.color}"
        desc += f" with a {self.body.shape} shape"
        return desc
```



```python
# Create a spaceship object and interact with it
my_spaceship = Spaceship()
print(my_spaceship.describe())
print(my_spaceship.engine.start())
my_spaceship.body.change_color("Blue")
print(my_spaceship.describe())
print(my_spaceship.engine.stop())
```

Example run:

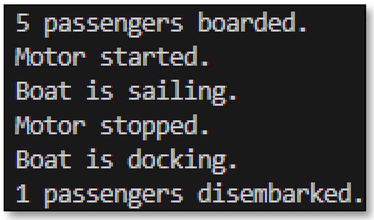


```
The spaceship is White with a Sleek shape
Engine started
The spaceship is Blue with a Sleek shape
Engine stopped
```

Assignment 1: Boat Class with Composition

Create a Boat class using Composition. The Boat class is composed of a Motor class and a Passenger class.

Example run:



```
5 passengers boarded.
Motor started.
Boat is sailing.
Motor stopped.
Boat is docking.
1 passengers disembarked.
```

Glossary

attribute A variable that is part of a class.

class A template that can be used to construct an object. Defines the attributes and methods that will make up the object.

child class A new class created when a parent class is extended. The child class inherits all of the attributes and methods of the parent class.

constructor An optional specially named method (__init__) that is called at the moment when a class is being used to construct an object. Usually this is used to set up initial values for the object.

destructor An optional specially named method (__del__) that is called at the moment just before an object is destroyed. Destructors are rarely used.

inheritance When we create a new class (child) by extending an existing class (parent). The child class has all the attributes and methods of the parent class plus additional attributes and methods defined by the child class.

method A function that is contained within a class and the objects that are constructed from the class. Some object-oriented patterns use 'message' instead of 'method' to describe this concept.

object A constructed instance of a class. An object contains all the attributes and methods that were defined by the class. Some object-oriented documentation uses the term 'instance' interchangeably with 'object'.

parent class The class which is being extended to create a new child class. The parent class contributes all its methods and attributes to the new child class.

Assignment Submission

1. Attach the pseudocode or create a TODO.
2. Attach all tutorials and assignments.
3. Attach screenshots showing the successful operation of each tutorial program.
4. Submit in Blackboard.