Advanced Python Programming Course

Lecture 8.

Python decorators,

OOP in Python: Types of inheritance

Assoc. Prof. Kovalenko S.M.
Department of software engineering and intelligent management technologies,
NTU “KhPI”

Python decorators

Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it.

- `@classmethod`
- `@staticmethod`
- `@property`

First Class Objects

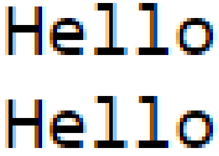
In Python, functions are *first class objects* which means that functions in Python can be used or passed as arguments.

Properties of first class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

Returning functions from another function

```
def outer_function():
    message = "Hello"

    def inner_function():
        print(message)
    return inner_function


my_func = outer_function()
my_func()
my_func()
```

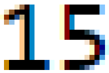


```
Hello
Hello
```

Crosure

- A closure in Python is a function that retains the state of the enclosing lexical environment at the time of its definition. This means that the function can remember and access variables in the enclosing lexical environment, even after it has finished executing.

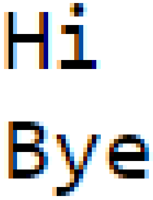
```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

closure = outer_function(10)
result = closure(5)
print(result)
```



```
15
```

Outer function with an argument

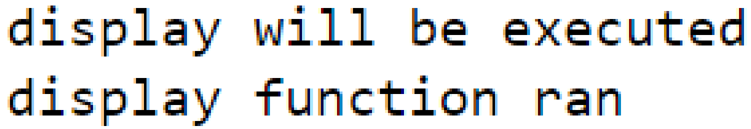
```
def outer_function(msg):
    message = msg

    def inner_function():
        print(message)
    return inner_function

hi_func = outer_function("Hi")
buy_func = outer_function("Bye")
hi_func()
buy_func()
```



```
Hi
Bye
```

Create a decorator

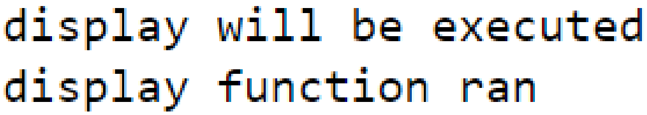
```
def decorator_function(original_function):
    def wrapper_function():
        print(f'{original_function.__name__} will be executed')
        return original_function()
    return wrapper_function


def display():
    print('display function ran')


decorated_display = decorator function(display)
decorated_display()
```



```
display will be executed
display function ran
```

Create a decorator

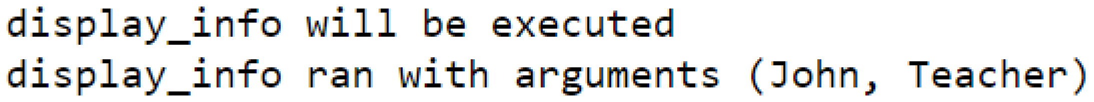
```
def decorator_function(original_function):
    def wrapper_function():
        print(f'{original_function.__name__}
will be executed')
        return original_function()
    return wrapper_function

@decorator_function
def display():
    print('display function ran')

# the same things
# display = decorator_function(display)
display()
```



```
display will be executed
display function ran
```

Using *args, **kwargs

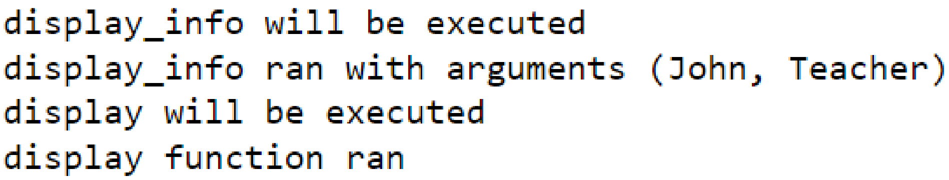
```
def decorator_function(original_function):
    def wrapper_function(*args, **kwargs):
        print(f’’’{original_function.__name__} will be executed’’’)
        return original_function(*args, **kwargs)
    return wrapper_function

@decorator_function
def display_info(name, position):
    print(f’’’display_info ran with arguments ({name}, {position}) ’’’)

display_info(‘John’, ‘Teacher’)
```



```
display_info will be executed
display_info ran with arguments (John, Teacher)
```

Decorators class

```
class decorator_class:

    def __init__(self, original_function):
        self.original_function = original_function

    def __call__(self, *args, **kwargs):
        print(f'{self.original_function.__name__} will be executed')
        return self.original_function(*args, **kwargs)


@decorator_class
def display_info(name, position):
    print(f'display_info ran with arguments ({name}, {position})')

@decorator_function
def display():
    print('display function ran')

display_info("John", "Teacher")
display()
```



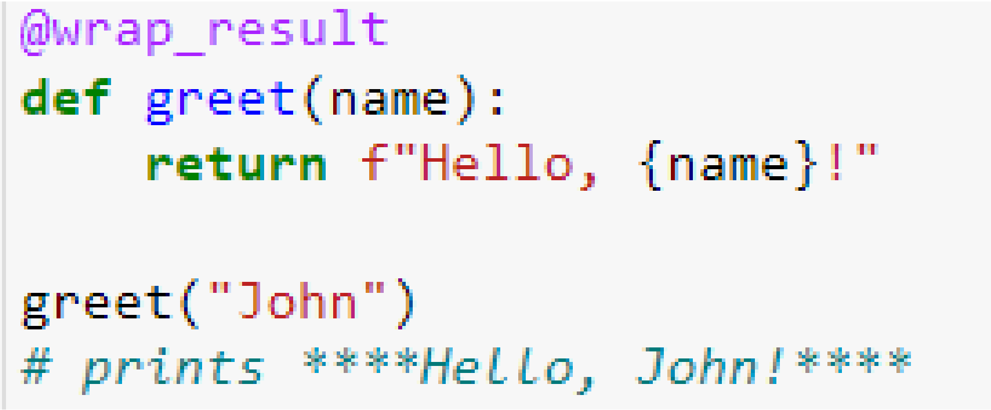
```
display_info will be executed
display_info ran with arguments (John, Teacher)
display will be executed
display function ran
```

Exercises to practice (optional)

- Exercise 1

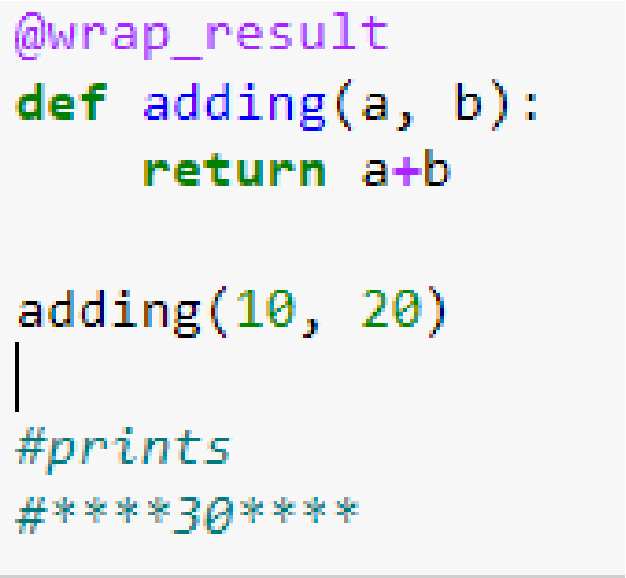
Create a Python decorator to add stars to the left and right sides of the result

Example 1



```
@wrap_result
def greet(name):
    return f"Hello, {name}!"

greet("John")
# prints ****Hello, John!****
```

Example 2



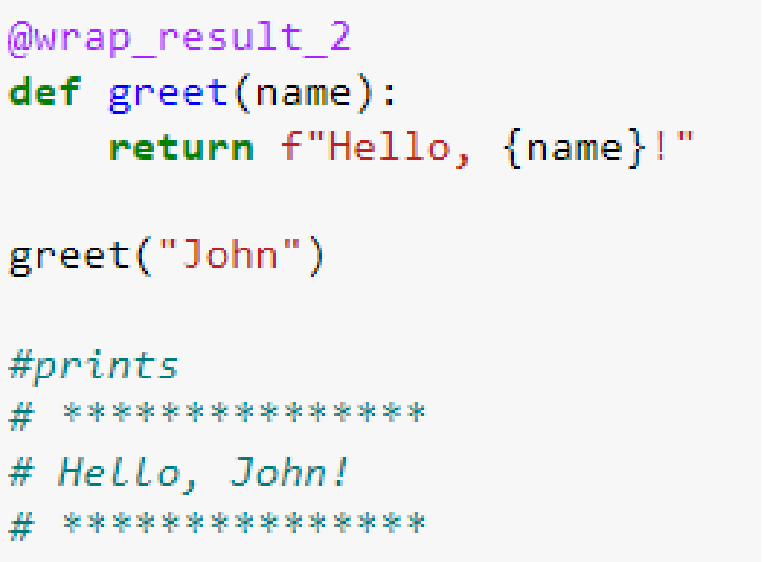
```
@wrap_result
def adding(a, b):
    return a+b

adding(10, 20)

#prints
#****30****
```

Exercises to practice (optional)

- Exercise 2

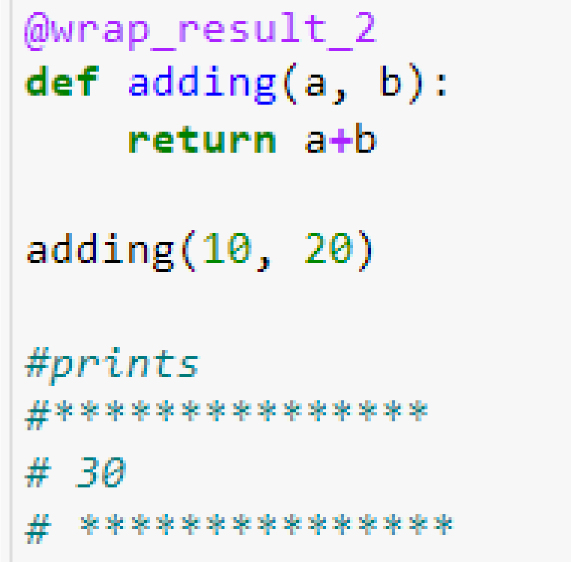
Create a Python decorator to add stars above and below the result

Example 1



```
@wrap_result_2
def greet(name):
    return f"Hello, {name}!"

greet("John")

#prints
# ***************
# Hello, John!
# ***************
```

Example 2



```
@wrap_result_2
def adding(a, b):
    return a+b

adding(10, 20)

#prints
#***************
# 30
# ***************
```

Types Of Inheritance In Python

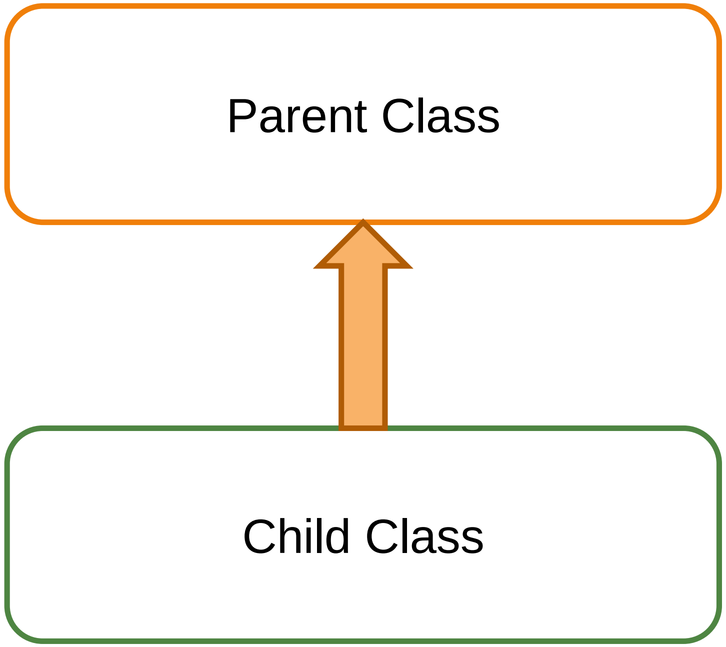
Python supports 5 types of inheritance:

- Single
- Multiple
- Multi-level
- Hierarchical
- Hybrid

https://www.onlinetutorialspoint.com/python/inheritance-in-python.html

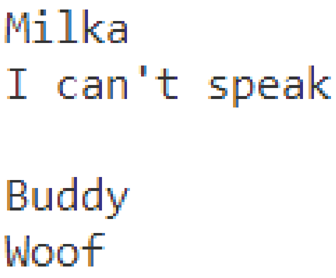
Single Inheritance

- This is the simplest form of inheritance where a derived class inherits from one and only one base class as in a parent-child relationship



Single Inheritance. Example

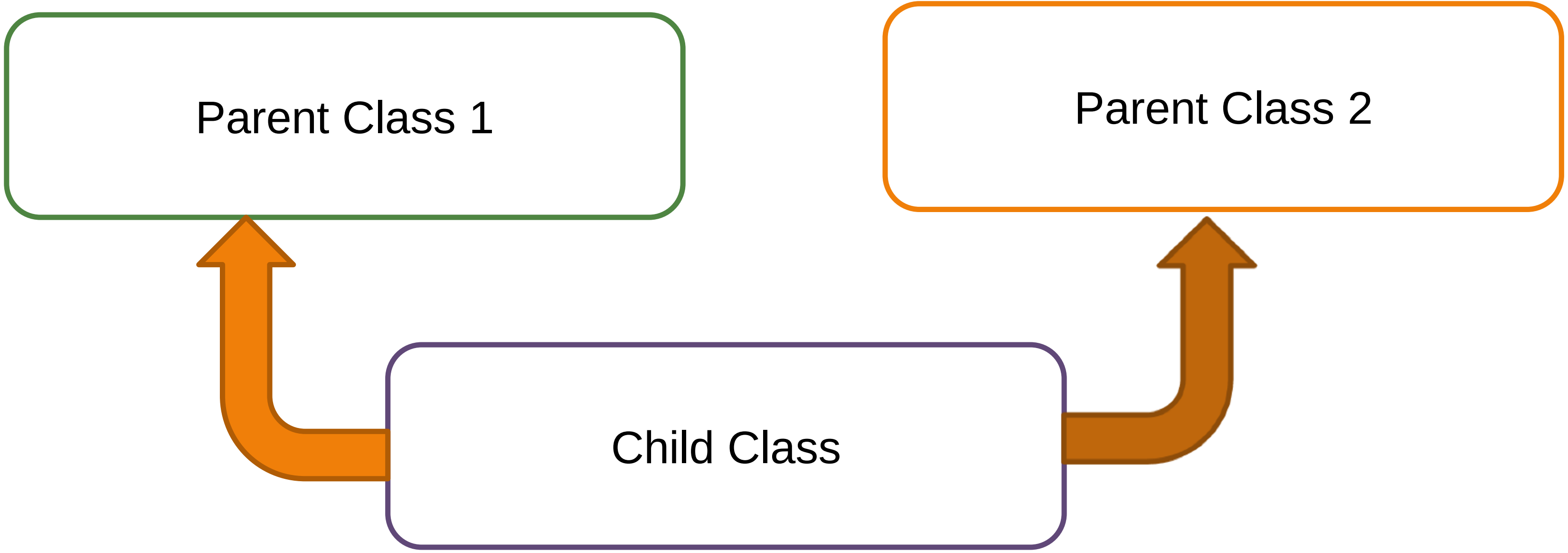
```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("I can't speak")

class Dog(Animal):
    def speak(self):
        print("Woof")

some_animal = Animal("Milka")
print(some_animal.name)
some_animal.speak()
print()

my_dog = Dog("Buddy")
print(my_dog.name)
my_dog.speak()
```



```
Milka
I can't speak

Buddy
Woof
```

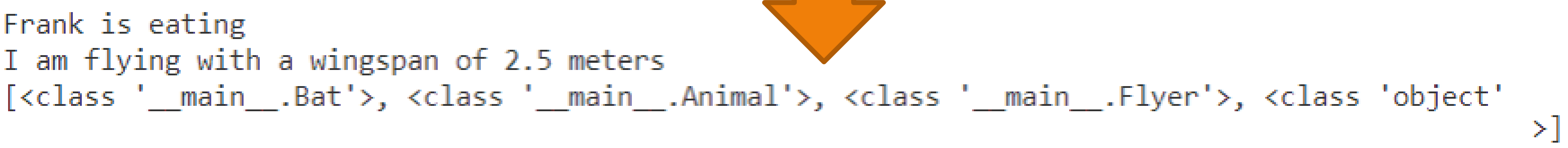
Multiple Inheritance

- This type of inheritance is present where a class directly inherits from two or more (multiple) classes.



Multiple Inheritance. Example

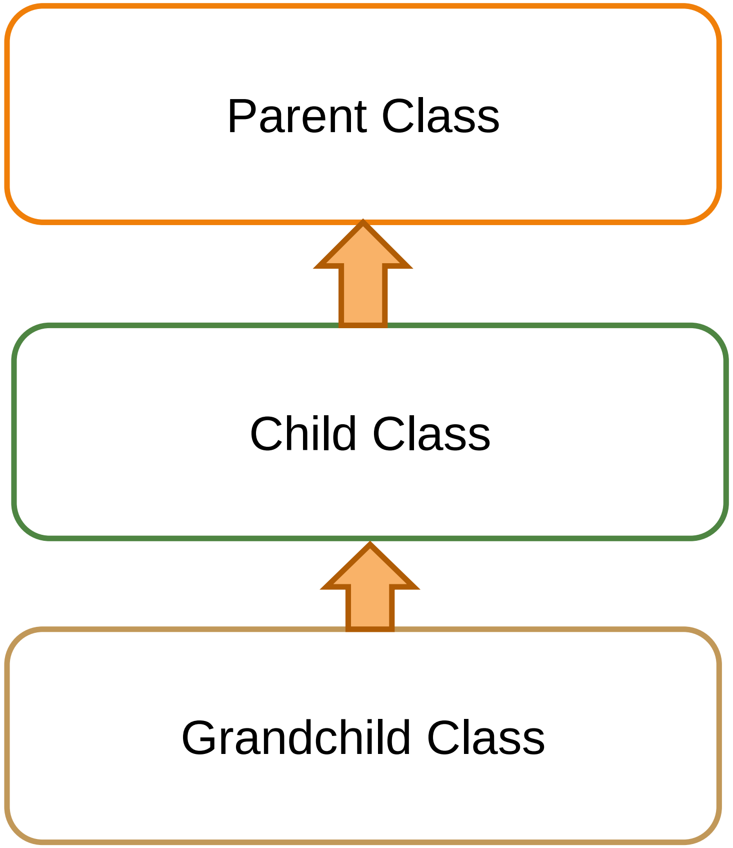
```
class Animal:
    def __init__(self, name):
        self.name = name

    def eat(self):
        print(f"{self.name} is eating")

class Flyer:
    def __init__(self, wingspan):
        self.wingspan = wingspan

    def fly(self):
        print(f"I am flying with a wingspan of {self.wingspan} meters")

class Bat(Animal, Flyer):
    def __init__(self, name, wingspan):
        Animal.__init__(self, name)
        # super().__init__(name)
        Flyer.__init__(self, wingspan)

bat = Bat("Frank", 2.5)
bat.eat()
bat.fly()

print(Bat.mro())
```



```
Frank is eating
I am flying with a wingspan of 2.5 meters
[<class '__main__.Bat'>, <class '__main__.Animal'>, <class '__main__.Flyer'>, <class 'object'
>]
```

Multi-level Inheritance

- This type of inheritance is present where a class **indirectly** inherits from a class as in a grandparent and grandchild relationship. At least 3 classes are involved in this type of inheritance.



Multi-level Inheritance. Example

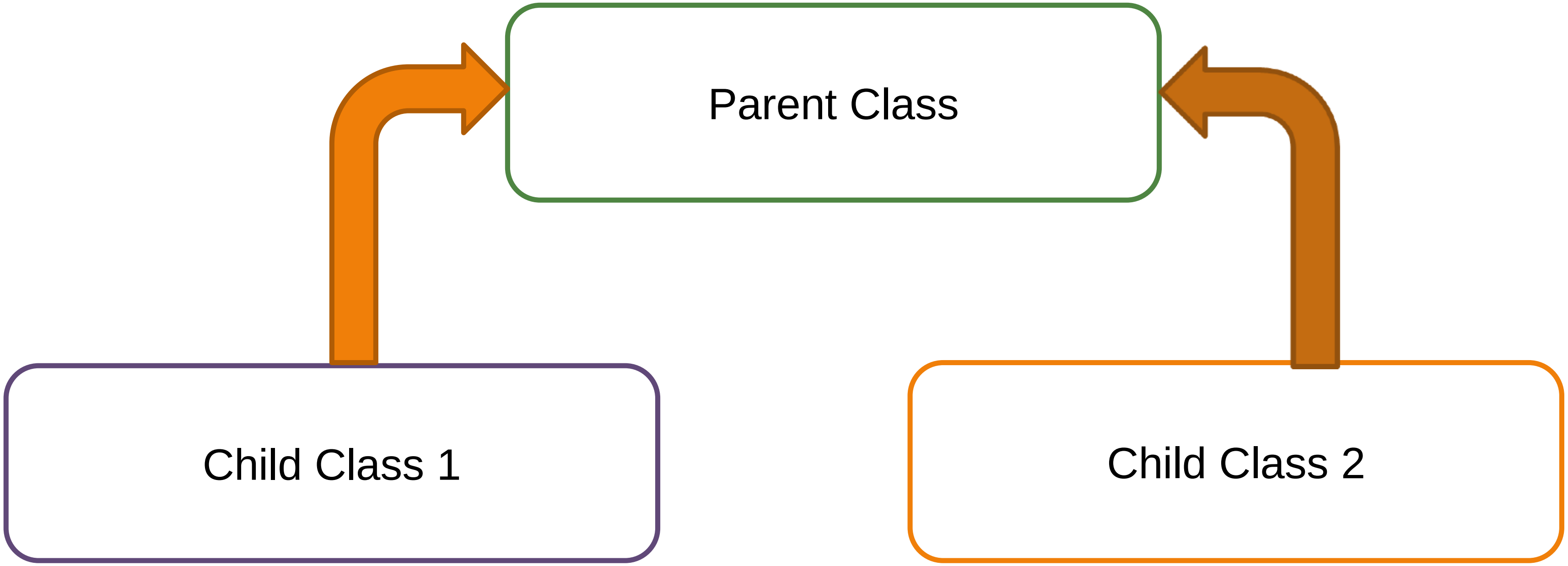
```
class Vehicle:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def get_info(self):
        return f"{self.year} {self.brand} {self.model}"

class Car(Vehicle):
    def __init__(self, brand, model, year, num_doors):
        super().__init__(brand, model, year)
        self.num_doors = num_doors

    def get_info(self):
        return super().get_info() + f", {self.num_doors} doors"
```

Multi-level Inheritance. Example

```
class ElectricCar(Car):
    def __init__(self, brand, model, year,
                 num_doors, battery_capacity):
        super().__init__(brand, model, year, num_doors)
        self.battery_capacity = battery_capacity

    def get_info(self):
        return (super().get_info()
              + f", {self.battery_capacity} kWh battery")


my_car = ElectricCar("Tesla", "Model S", 2022, 4, 100)
print(my_car.get_info())
```

Hierarchical Inheritance

- In Hierarchical inheritance, two or more (multiple) classes inherit from a single Base class. It is similar to a tree-like structure

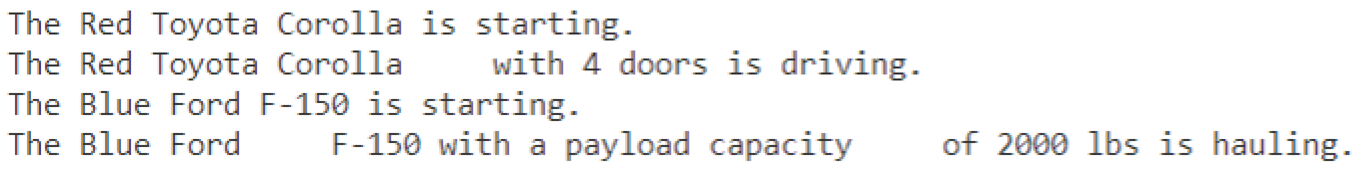


Hierarchical Inheritance. Example

```python
class Vehicle:
    def __init__(self, color, make, model):
        self.color = color
        self.make = make
        self.model = model

    def start(self):
        print(f"The {self.color} {self.make} {self.model} is starting.")

class Car(Vehicle):
    def __init__(self, color, make, model, num_doors):
        super().__init__(color, make, model)
        self.num_doors = num_doors

    def drive(self):
        print(f"The {self.color} {self.make} {self.model} \
    with {self.num_doors} doors is driving.")
```

Hierarchical Inheritance. Example

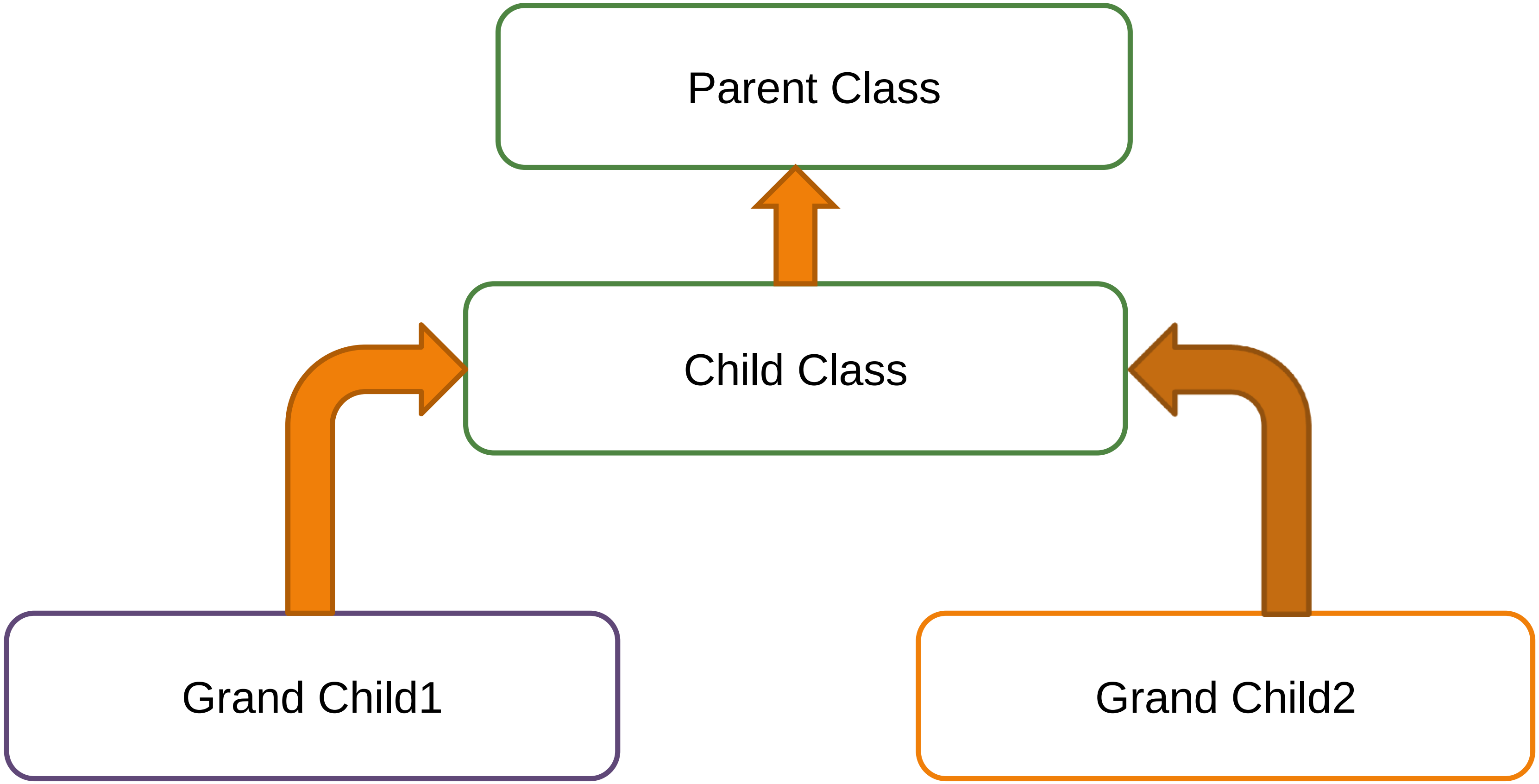
```
class Truck(Vehicle):
    def __init__(self, color, make, model, payload_capacity):
        super().__init__(color, make, model)
        self.payload_capacity = payload_capacity

    def haul(self):
        print(f"The {self.color} {self.make} \
    {self.model} with a payload capacity \
    of {self.payload_capacity} lbs is hauling.")

car = Car("Red", "Toyota", "Corolla", 4)
car.start()
car.drive()

truck = Truck("Blue", "Ford", "F-150", 2000)
truck.start()
truck.haul()
```



```
The Red Toyota Corolla is starting.
The Red Toyota Corolla     with 4 doors is driving.
The Blue Ford F-150 is starting.
The Blue Ford     F-150 with a payload capacity     of 2000 lbs is hauling.
```

Hybrid Inheritance

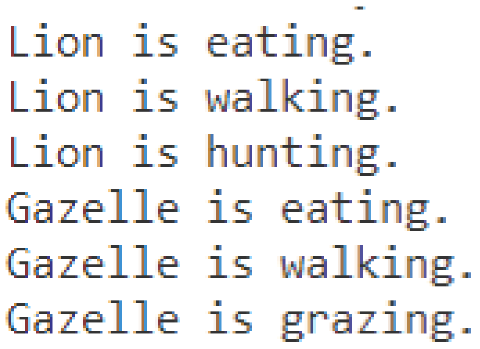
- This type of inheritance is made up of a combination of other types of inheritance.



Hybrid Inheritance. Example

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def eat(self):
        print(f"{self.name} is eating.")

class Mammal(Animal):
    def walk(self):
        print(f"{self.name} is walking.")

class Carnivore(Mammal):
    def hunt(self):
        print(f"{self.name} is hunting.")
```

```python
class Herbivore(Mammal):
    def graze(self):
        print(f"{self.name} \
    is grazing.")

lion = Carnivore("Lion")
lion.eat()
lion.walk()
lion.hunt()

gazelle = Herbivore("Gazelle")
gazelle.eat()
gazelle.walk()
gazelle.graze()
```



```
Lion is eating.
Lion is walking.
Lion is hunting.
Gazelle is eating.
Gazelle is walking.
Gazelle is grazing.
```

SOLID principles

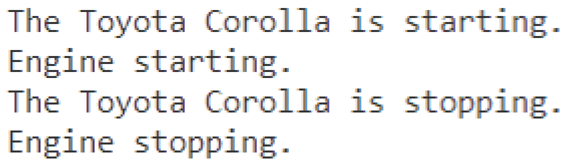
https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898

Class Composition

- In Python**, class composition** is a way to build complex objects by combining simpler objects or components. Instead of inheriting behavior from parent classes, class composition involves creating new objects that contain references to other objects.
- Class composition is often used to implement the **"has-a"** relationship between objects, where one object contains another object as a component or part. For example, a car "has-a" engine, a person "has-a" name, and a book "has-a" title and author.

Class Composition

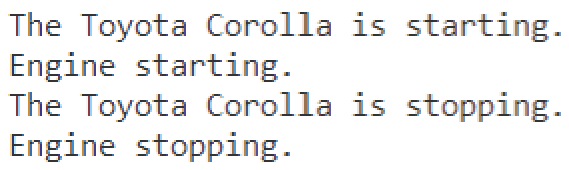
```python
class Engine:
    def start(self):
        print("Engine starting.")

    def stop(self):
        print("Engine stopping.")

class Car:
    def __init__(self, make, model, engine):
        self.make = make
        self.model = model
        self.engine = engine

    def start(self):
        print(f"The {self.make} {self.model} is starting.")
        self.engine.start()

    def stop(self):
        print(f"The {self.make} {self.model} is stopping.")
        self.engine.stop()

engine = Engine()
car = Car("Toyota", "Corolla", engine)
car.start()
car.stop()
```



```
The Toyota Corolla is starting.
Engine starting.
The Toyota Corolla is stopping.
Engine stopping.
```

Nested Classes

```python
class Car:
    class Engine:
        def start(self):
            print("Engine starting.")

        def stop(self):
            print("Engine stopping.")

    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.engine = Car.Engine()

    def start(self):
        print(f"The {self.make} {self.model} is starting.")
        self.engine.start()

    def stop(self):
        print(f"The {self.make} {self.model} is stopping.")
        self.engine.stop()

car = Car("Toyota", "Corolla")
car.start()
car.stop()
# Car.start(car)
# Car.Engine().start()
```



```
The Toyota Corolla is starting.
Engine starting.
The Toyota Corolla is stopping.
Engine stopping.
```