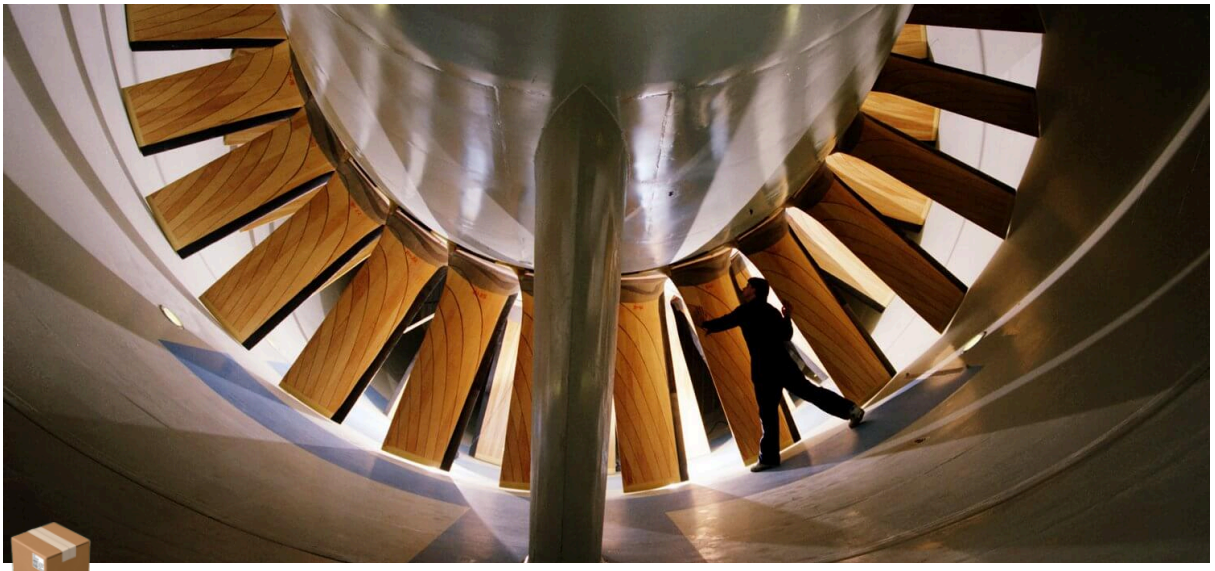

# Notes: OOP basics

Classes allow us to group out data and functions in a way its is easier to reuse. Attributes - Data associated with a certain class. Class methods - methods associated with a certain class.

> More on GitHub: https://github.com/chabrovs/py/tree/main/RoadMap/OOP/PythonOOPNotes

---

## Theoretical Part:
## ▼ Classes and Instances

> This chapter introduces into Classes and Instances outlining the difference between them and usage.

### ▼ A Class

#### Intro:

A **class** is a programming language syntax construct that serves as a blueprint (instruction) for creating *(instantiating)* objects *(instances of a class)*. Based on a single class it's possible to create multiple instances with the same or different data.

In programming, classes allow to model real-world entities, as well as to build relationships between them utilizing techniques such as association, inheritance, and composition.

The conceptual idea behind classes is to encapsulate data *(which is represented as attributes)* and behavior *(which is represented by methods)* logically related to each other into a single object.

#### Methods and Attributes:

**Attributes:**

- `__name__` — The name of the class.
- `__module__` — The name of the module in which class is defined.
- `__class__` — This is the reference to the class's metaclass, which is `type` by default.

- `__doc__` — The class's docstring.
- `__bases__` — A tuple containing the base classes form which the class inherits.
- `__dict__` — A dictionary containing class's namespace, including it's attributes and methods.
- `__mro__` — The Method Resolution Order (MRO), a tuple listing the order in which Python will search for a method.
- `__subclasses__` — A list of classes that directly inherit from the class.

**Methods:**

- `__init_subclass__(cls)` — A hook that is called whenever the class is a subclass of another class.
- `__new__(cls, *args, **kwargs)` — A static method that is called before `__init__` to create and return the new instance. It's often used for metaclass programming.
- `__call__(self, *args, **kwargs)` — When a class is called to create an instance (e.g., `MyClass()`), this method is executed. It, in tern, calls `__new__` and then `__init__`.
- `__dir__(self)` — Returns a list of class's attributes and methods for tab-completion and other introspective tools.

### Example:

```
# main.py

class Person:
    def __init__(self, first_name: str, last_name: str, age: int):
        self.first_name = first_name
        self.last_nage = last_name
        self.age = age
```

### Types Of Classes:

1. **Abstract Base Class:**
   - it's a blueprint that cannot be instantiated on its own.
   - Designed to be a base class for concrete classes, providing a common interface and possibly some shared functionality.
   - Python uses the `abc.ABC` object to define abstract base class.
2. **Concrete Class:**
   - **I**t's a complete and fully implemented class that can be used to instantiate objects.
   - All its attributes and methods are fully defined and it does not require any further implementation.
3. **Data Class:**
   - It's a class designed to hold data rather than functionality.
   - It's defined by Python's decorator `@dataclass`.

## ▼ An Instance

### Intro:

An **object** is in-memory entity that combines data *(state)* and behavior *(methods)*. It's a concrete instance of a class created during program execution or compilation.

### Example:

An object instantiation based on the class:

```
# main.py

class Person:
    def __init__(self, first_name: str, last_name: str, age: int):
        self.first_name = first_name
        self.last_nage = last_name
        self.age = age

if __name__ == "__main__":
    person_1 = Person("Sergei", "Chabrov", 16)
```

## ▼ Data Classes

### Intro:

A **data class** is a special type of a class primary purpose of which is to store data. To declare a data class Python uses the built-in `@dataclass` decorator.

### How it works:

Whenever the `@dataclass` decorator is used, Python automatically generates several standard special methods:

1. `__init__` — A constructor that accepts arguments for each defined field, initializing the instance attributes.
2. `__repr__` — A method that provides a useful, human-readable string representation of the object, showing the class name and values of all its fields.
3. `__eq__` — A method that allows instances to be compared for equality (`==`), checking all it field values are identical.
4. `__hash__` — Automatically generates if the class is designed to be immutable (via `frozen=True` argument), allowing instances to be used as keys in dictionaries or elements in sets.

### Use cases:

- To store data object.

### The `@dataclass` decorator parameters:

- `slots: bool` — tells Python not to create `__dict__` for instances of the class, but to use `__slots__` that stores instance's attributes directly in the object's memory.
    - **Use:** when you do not need to add new attributes to the data class after instantiation. When you want to be more memory efficient. Also, read time a bit faster.
- `init: bool` — generates the `__init__` method.
    - **Use:** set to `False` if you want to define your own custom `__init__` method, for example, to perform complex validation or handle arguments differently.
- `repr: bool` — generates the `__repr__` method.
    - **Use:** set to `False` if you want to provide custom string representation.
- `eq: bool` — generated the `__eq__` method automatically.

    - **Use:** set to `False` if you have a custom logic for comparing instances, or if objects are unique and should never be considered equal based on field values.
- `order: bool` — if set to `True`, it generates comparison methods such as `__lt__`, `__gt__`, `__le__`, `__ge__`.
    - **Use:** when you want to compare instances based on the order of their fields. For example, a `Point` class could be ordered by `x` and then `y`. Comparison is done lexicographically based on the order of fields in the class definition.
- `freeze: bool` — When set to `True` instances of the data class become immutable. This is useful for creating hashable objects (as `__hash__` generated for frozen data classes).
    - **Use:** If you need to use the data class instances as keys in a dictionary or elements of a sets.
- `unsafe_hash: bool` — controls the generation of the `__hash__` method. If `eq=True` and `frozen=False`, the `__hash__` method is not generated, making the data class unhashable. Setting `unsafe_hash=True` forces a `__hash__` method to be generated.
    - **Use:** You would use `unsafe_hash=True` to make a mutable data class hashable. This is considered unsafe because the hash value of a mutable object can change, which can lead to bugs, as dictionaries and sets rely on a consistent hash value.

### Examples:

```
# main.py

from dataclasses import dataclass

@dataclass
class Point3D:
    x: int
    y: int
    z: int


if __name__ == "__main__":
    point1 = Point3D(1, 2, 3)
    print(point1) # STDOUT: Point3D(x=1, y=2, z=3)
```

## ▼ Abstract Base Classes

### Intro:

An **Abstract Base Class (ABC)** is a class that cannot be instantiated on its own but serves as a blueprint for other classes. ABCs can include both fully implemented methods and abstract methods (methods that have no implementation and must be defined in a subclass). In Python, ABCs are part of the `abc` module.

**Key Idea**: ABCs are used when you want to enforce certain methods or properties in subclasses but also provide some shared functionality.

### Example:

```
# main.py

from abc import ABC, abstractmethod
```

```
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

    def sleep(self):
        print("Sleeping...")


class Dog(Animal):
    def make_sound(self):
        print("Bark!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")


if __name__ == "__main__":
    # Animal cannot be instantiated
    animal = Animal()  # STDERR: TypeError

    dog = Dog()
    dog.make_sound()    # STDOUT: Bark!
    dog.sleep()         # STDOUT: Sleeping...

    cat = Cat()
    cat.make_sound()    # STDOUT: Meow!
```

**Explanation:**

In the example above, the `Animal` is an abstract class, it defines the abstract method `make_sound`, which must be implemented by every subclass. It also provides a concrete method `sleep`, which is available to all subclasses. This approach allows to define a common structure while enforcing a contract for the subclasses to follow.

# ▼ Attributes: Instance, Class, Static

> This chapter outlines clear distinguishment between Instance, Class, and Static attributes within a class.

## ▼ Instance attributes:

### Intro:

**Instance attributes** are unique to each object *(instance)* created from a class. They are defined inside a class's methods, most commonly in the `__init__` method, using the `self` keyword which refers to the particular instance of a class.

Each time a new object is instantiated, it gets its own copy of the instance attributes, and changing the value of an attribute on one instance does not affect the others.

Instance attributes are stored in the instance's `__dict__`.

- **Declaration:** Typically in the constructor (`__init__` method) using `self.attribute_name = value`.
- **Scope:** Local to the specific object instance.
- **Access:** Accessed via an instance of the class (e.g., `my_object.attribute_name`).

### Use case:

- Store data (state) that is unique to each object.

### Example:

```
# main.py

class Employee:
    def __init__(self, name: str):
        self.name = name


if __name__ == "__main__":
    employee1 = "Sergei"
    employee2 = "Alex"

    print(employee1.name) # STDOUT: Sergei
    print(employee2.name) # STDOUT: Alex
```

## ▼ Class attributes:

### Intro:

**Class attributes** are shared by all instances of a class. They are defined directly in the class body namespace, outside of any methods. They are a part of the class itself and not tied to any specific instance. Changes to a class attribute will affect all instances that do not have their own instance attribute with the same name.

They are stored in the class's `__dict__`.

- **Declaration:** Defined directly within the class body.
- **Scope:** Shared across all instances of the class.
- **Access:** Accessed via the class itself (e.g., `Dog.species`) or an instance (`my_object.species`).

### Use case:

- Data shared among all instances of the same class.
- Constants.
- Shared configurations.
- Tracking state: Managing count reference to all objects created from a class. For example, `_total_connections` attribute in a `Database` class, or `_total_users` in a `User` class to track how many instances were created.

### Example:

```
# main.py
```

```
class Dog:
    species = "Canis lupus familiaris"

    def __init__(self, name: str):
        self.name = name

if __name__ == "__main__":
    dog1 = Dog("Bubby")
    dog2 = Dog("Lucky")

    print(dog1.species) # STDOUT: Canis lupus familiaris
    print(dog2.species) # STDOUT: Canis lupus familiaris
```

## ▼ Static Attributes:

### Intro:

A **Static attribute** is essentially the same as a **class attribute** in Python. The term “static” comes from other programming languages like Java, C++, where static members are tied to the class rather than an instance. In Python, the concept of a “static attribute” is synonymous with a **class attribute.**

⚠️ There is no special keyword like `static` to declare them.

### How to reinforce the idea of a static attribute in Python:

To reinforce the idea of a static attribute in Python, it’s possible to declare a class attribute and declare a static method using the `@staticmethod` decorator to access the classes attribute.

### Use cases:

- Same as for class attributes in Python.

**Example:**

```
# main.py

class Dog:
    species = "Canis lupus familiaris"

    @staticmethod
    def get_species():
        return Dog.species
```

# ▼ Methods: Instance, Class, Static

> This chapter outlines clear distinguishment between Instance, Class, and Static methods within a class.

## ▼ Instance methods:

### Intro:

An **instance method** is a method that operates on an instance of the class and can access and modify both instance and class attributes. When the instance method is called, the like to the instance itself is automatically passes as the first argument, conventionally named `self`.

- **How it works:** When `my_object.method()` is called, Python automatically passes `my_object` as the first `self` argument.
- **Purpose:** To define the behavior of individual objects.
- **Access:** Can access `self` (the instance) and `self.__class__` (the class).

### Use cases:

- Modify object's state.
- Access object's data.
- Performing actions on a specific object.

## ▼ Class methods:

### Intro:

A **class method** operates on the class itself, not on specific instance. It's defined using the `@classmethod` decorator. When a class method is called, Python automatically passes a link to the class itself as the first argument, conventionally named `cls`.

- **How it works:** When `Class.method()` is called, Python automatically passes `Class` as the first `cls` argument.
- **Purpose:** To define alternative constructors for the class or to work with class-level attributes.
- **Access:** Can access `cls` (the class) but not `self` (the instance).

### Use cases:

- Logic should apply to a class itself, not an individual instance.
- Alternative constructors (Factory methods): a class method can be used to create an instance in a different way rather than the standard `__init__` constructor.
- Tracking class-wide state.
- Accessing class attributes.

### Example:

```
# main.py

class Car:
    _total_cars = 0

    def __init__(self, brand):
        self.brand = brand
        Car._tatal_cars += 1

    @classmethod
    def get_total_cars(cls):
        return cls._total_cars
```

## ▼ Static methods:

### Intro:

A **static method** operates only with function's named arguments, and key-word argument. The `@staticmethod` decorator changes Python's default behavior of passing the instance as the first parameter to the function.

### Use cases:

Utility function that are logically grouped with a class but do not need to access to the class or any instance.

**For example:**

- Helper functions.
- Namespace organization: grouping related utility functionality together.
- Static validation: a static method can be used to validate input before an object is even created.

### Example:

```
# main.py

class User:
    def __init__(self, username: str):
        self.validate_username(username)
        self.username = username

    @staticmethod
    def validate_username(username: str):
        if len(username) < 3:
            raise ValueError(
                f"username.__length__ must be > 3, got {len(username)}"
            )


if __name__ == "__main__":
    user = User("ab")
    # STDERR: ValueError: username.__length__ must be > 3, got 2
```

## ▼ How it works under the hood:

The `@classmethod` and `@staticmethod` decorators are implemented using the *descriptor protocol*. They are both non-data descriptors that implement the `__get__` method.

When a method decorated with `@classmethod` or `@staticmethod` is accessed from a class or an instance, Python's descriptor protocol is triggered. The descriptor's method `__get__` called to transform the function.

1. `@classmethod` — Its `__get__` method takes the class object (`cls`) ans the function and returns a new function (a bound method) that automatically passes `cls` as the first argument.

```
# main.py

class ClassMethod(object):
        def __init__(self, f):
            self.f = f

    def __get__(self, instance, owner):
```

```
        """
        :Params:
        :``self``: Non-Data Descriptor instance.
        :``instance``: Thefucntion itself.
        :``owner``: The class the fucntion declared in.
        """

          def new_func(*args, **kwargs):
             return self.f(owner, *args, **kwargs)
          return new_func

     def __call__(self, *args, **kwargs):
        return self.f(self, instance, *args, **kwargs)


if __name__ == "__main__":
    class Test:
        def my_class_method(cls):
            print(cls)

        class_method = ClassMethod(my_class_method)

    test = Test()
    test.my_class_method() # STDOUT: <class '__main__.Test'>
```

2. `@staticmethod` — Its `__get__` method simply returns that original function without any modifications. It does not bind the function to an instance or a class, which is why it does not receive `self` or `cls`.

```
# main.py

"""
By default, Python passed a link to the object instance.
"""

from typing import Callable


class StaticMethod:
    def __init__(self, f: Callable):
        self.f = f

    def __get__(self, instance, owner):
        return self.f


class Test:
        # @staticmethod
    def my_static_method(arg1, arg2):
        print(f"arg1={arg1}; arg2={arg2}")
```

```
    my_static_method = StaticMethod(my_static_method)


if __name__ == "__main__":
    test = Test()
    test.my_static_method(
        "Hello", "Static"
    )  # STDOUT: arg1=Hello; arg2=Static
```

### Quick Recall:

- **instance method** — enforces the interpreter to pass the class object instance as the first parameter to the function. Conventionally, named `self` . Named, and key-word argument are also passed after the link to the object instance.
- **Class method** — the `@classmethod` decorator enforces to pass the class itself as the first parameter to the function. Conventionally, named `cls`. Named, and key-word argument are also passed after the link to the class.
- **Static method** — the `@staticmethod` decorator enforces to pass only named arguments, and key-word arguments to the function. Does not use any keywords.

# ▼ Inheritance

> This chapter describes inheritance mechanism in Python.

## ▼ Introduction:

### Definition:

Inheritance is a way to establish an inter-class relationships between multiple classes widely used in Object-Oriented Programming.

### Relationship type — `Is-a-type-of` :

Inheritance enhances the **is-a-type-of** relationship between classes. Where a

subclass **Is-a-type-of** the superclass relationship.

**Examples:**

- An *engineer* **is a type of** *an employee*.
- A *square* **is a type of** *a rectangle*.
- *A cube* **is a type of** a square.
- And so on...

## ▼ Method Resolution Order (MRO):

### Into:

**Method Resolution Order (MRO)** is the order Python looks for methods in a hierarchy of classes. To determine which method of all method with same name within a class hierarchy to invoke.

Every class has the `.__mro__` attribute that allows to inspect the order.

### How to Define MRO:

- **Method Resolution Order (MRO)** can be defined by the order in which a subclass inherits from superclasses.
- Also, **Method Resolution Order (MRO)** can be altered by the `super()` function parametrization where the first parameter sets the lower bound of the search and the second parameter tights the lower bound object and its hierarchy to the subclass the `super()` function called from.

### Example:

```
# main.py

class Rectangle:
    def __init__(self, width: float, height: float):
        self.width = width
        self.height = height

    def area(self) -> float:
        return float(self.width * self.height)

    def perimeter(self) -> float:
        return float(2 * self.width + 2 * self.height)


class Square(Rectangle):
    def __init__(self, length):
        """
        The parametrized `super(Subclass, self)` function == \
            `super()`, is an equivalent of the non-parametrized \
            function.
        """

        super(Square, self).__init__(length, length)


class Triangle:
    def __init__(self, base: float, height: float):
        self.base = base
        self.height = height

    def area(self) -> float:
        return float(self.base * self.height * 0.5)


class RightPyramid(Triangle, Square):
    def __init__(self, base, slant_height):
        self.base = base
        self.slant_height = slant_height
        super().__init__(self.base) # TypeError
        s = super()
        print(type(s))

    def area(self) -> float:
```

```
        base_area = super().area()
        perimeter = super().perimeter()

        return float(0.5 * perimeter * self.slant_height + base_area)

if __name__ == "__main__":
    pyramid = RightPyramid(2, 4)
    print(RightPyramid.__mro__)
    print(pyramid.area())
    # Raises a type error as Tragne missing 1 attribute heigth

    class RightPyramid(Square, Triangle):
    def __init__(self, base, slant_height):
        self.base = base
        self.slant_height = slant_height
        super().__init__(self.base) # TypeError
        s = super()
        print(type(s))

    def area(self) -> float:
        base_area = super().area()
        perimeter = super().perimeter()

        return float(0.5 * perimeter * self.slant_height + base_area)

     pyramid = RightPyramid(2, 4)
     print(RightPyramid.__mro__)
     print(pyramid.area())

     # Now MRO point to the Square class first.
```

### C3 Linearization Algorithm:

The C3 Linearization Algorithms is used by Python in MRO. It respects two rules:

1. Children precede their parents.
2. If a class inherits from multiple classes, they are kept in the order specified in the tuple of the base class.

The algorithm follows these rules:

- Inheritance graph determined the structure of the Method Resolution Order.
- User have to visit the super class only after the method of the local classes are visited.
- Monotonicity.

## ▼ The `super()` function:

### Intro:

The `super()` method is a Python built-in function that is used inside a subclass's constructor (`__init__()`) to access the superclass's attributes and methods.

The `super()` function returns a **temporary instance** of the superclass, which allows to access its attributes and method within a subclass.

## The `super()` function in Single Inheritance:

**Example:**

```
# main.py

class Rectangle:
    def __init__(self, height: float, width: float):
        self.width = width
        self.height = height

    def calculate_area(self) -> float:
        return float(self.width * self.height)

    def calculate_perimeter(self) -> float:
        return float(2 * self.width + 2 * self.height)

class Square(Rectangle):
    """
    In this subclass we do not have to write \
    calculate_area and calculate_perimeter methods \
    as we can use Rectengle's methods can satisfy our goals.
    """

    def __init__(self, length: float):
        super().__init__(length, length)

class Cube(Square):
    def __init__(self, length: float):
        super().__init__(length)

    def calculate_volume(self) -> float:
        return float(self.calculate_area() * self.length)

    def calculate_surface_area(self) -> float:
        return float(self.calculate_area() * 6)
```

## Parameters for `super(SuperClass, self)`:

The `super()` method essentially takes two parameters:

1. Is the superclass.
2. An instance of the first argument (`self`).

**Why? and How it Works?:**

This can be used to alter Method Resolution Order (MRO) for a subclass's instance. In default behavior (when parameters to the `super()` method are not provided), the MRO for a subclass instance behaves following way — ...

**Note:** If the first parameter of the `super()` function matches the superclass, and the second parameter is `self` — it's an equivalent of a parameterless call of the `super()` function.

### Example:

```
class A:
    def method(self):
        ...

class B(A):
    # here we override the method form class A
    def method(self):
        ...

class C(B):
    def method(self):
            # But still call method from class A
        super(B, self).method()
```

### The `super()` function in Multiple inheritance:

**Note:**

- Designing class that use multiple inheritance it's important to exclude same signatures in separate classes. As it can confuse the MRO mechanism so far as the first signature matched to be invoked.

**Example of Cooperative Inheritance:**

```
"""
This module shows how to use the `super()` function \
    in multiple inheritance.
An example of Cooperative Inheritance.
"""


class Shape:
    def __init__(self, **kwargs):
        # Call 'object's' init.
        super().__init__(**kwargs)


class Rectangle:
    def __init__(self, width: float, length: float, **kwargs):
        self.width = width
        self.length = length
        super().__init__(**kwargs)

    def area(self) -> float:
        return float(self.width * self.length)

    def perimeter(self) -> float:
        return float(2 * self.length + 2 * self.width)
```

```
class Square(Rectangle):
    def __init__(self, length: float, **kwargs):
        super(Square, self).__init__(
            width=length, length=length, **kwargs
        )


class Triangle(Shape):
    def __init__(self, base: float, height: float, **kwargs):
        self.base = base
        self.height = height
        super().__init__(**kwargs)

    def tri_area(self) -> float:
        return float(0.5 * self.base * self.height)


class RightPyramid(Square, Triangle):
    def __init__(self, base: float, slant_length: float, **kwargs):
        self.base = base
        self.slant_length = slant_length
        kwargs["height"] = slant_length
        kwargs["length"] = base
        super(RightPyramid, self).__init__(base=base, **kwargs)

    def area(self) -> float:
        base_area = super(RightPyramid, self).area()
        perimeter = super(RightPyramid, self).perimeter()

        return float(0.5 * perimeter * self.slant_length + base_area)

    def area_2(self) -> float:
        base_area = super(RightPyramid, self).area()
        triangle_area = super(RightPyramid, self).tri_area()

        return float(triangle_area * 4 + base_area)


if __name__ == "__main__":
    right_pyramid = RightPyramid(2, 4)
    print(right_pyramid.area())
    print(right_pyramid.area_2())
```

In this Example:

| Class | Named Argument | kwags |
|---|---|---|
| `Shape` | None | {} |
| `Rectangle` | width, length | {”base”: 2, “height: 4”} |
| `Square` | length | {”base”: 2, “height: 4”} |

| | | |
|---|---|---|
| `Triangle` | base, height | {} |
| `RightPyramid` | base, slant_height | {"length": 2, "height: 4"} |

## ▼ Design Patterns Using Inheritance

### Intro:

Python allows to to make up different design patterns base on rather *single inheritance* and *multiple inheritance(MI)*. There are three common design patterns using inheritance.

### 1. Classical / Regular Inheritance (Single)

- Inherits only from one parent.

### 2. Cooperative Inheritance

This design pattern used to manege the complexity of *multiple inheritance(MI)*.

- It requires all classes to explicitly call the `super()` function.
- It ensures that all parent classes in the Method Resolution Order (MRO) are properly initialized and that shared methods ere executed collaboratively without conflicts and redundancy.
- It allows a single call to `super().__init__()` in a child class to correctly initialize all its parents, even those parents are siblings in the hierarchy.
- It's often achieved by:
  - Introducing an abstract class (like Python's `object`) that has an empty `__init__()` method.
  - Ensuring each class constructor in the class hierarchy calls the `super().__init__(**kwargs)` function.
  - Accepting key-word arguments `**kwargs` in every constructor within the class hierarchy.
  - Obeying naming conventions where each argument name (including named argument and key-word argument) should be unique to avoid argument misapplication and `AttributeError` and `TypeError`.
  - In the child class additional key-word argument can be set before calling the `super().__init__(**kwargs)` function.

### 3. Mix-In Inheritance

This design pattern leverages *multiple inheritance(MI)* to share functional behavior.

- A Mix-In class is not instantiated alone; it provides a set of reusable methods (like loading JSON serialization capabilities).
- Mix-In inheritance follows the **has-a** pattern. Where a subclass has some functionality from the superclass (Mix-In in our case).
- This pattern helps to achieve **composition over inheritance** by separating core identity from added functionality.

# ▼ Composition & Aggregation

> This chapter describes alternative ways to build assassination between Python classes.

## ▼ Composition

### Intro:

**Composition** is a concept that is used to establish relationships between object.

### Relationship type — `has-a` :

*Composite* **has-a** *Component.*

*Often referred as* ***strong*** *has-a relationship.*

### Example:

A `Car` has an `Engine`, and the Car cannot function and exist without an engine.

```
# main.py


class Engine:
    def __init__(self):
        self.type = "V9"

class Car:
    def __init__(self):
        self.engine = Engine()

if __name__ == "__main__":
    car = Car()
```

## ▼ Aggregation

### Intro:

Aggregation is a form of composition where dependent objects *(components)* are instantiated outside of the composite class and then ready-to-use are injected.

Leverage Dependency Injection techniques to achieve aggregation.

### Relationship type — `has-a` :

*Often referred as* ***weak*** *has-a relationship.*

### Examples:

**Example #1:**

A `Department` has `Professor`s, but Professors can exist independently of a department. This example leverages dependency injection(DI) through the setter method, which allows to inject dependent objects in the runtime.

```
# main.py

class Professor:
    def __init__(self, name: str):
        self.name = name


class Department:
    def __init__(self, name: str):
        self.name = name
        self.professor_list = []
```

```
    def assign_professor(self, professor: Professor) -> None:
        self.professor_list.append(professor)

if __name__ == "__main__":
    professor1 = Professor("Sergei")
    professor2 = Prodessor("Chabrov")

    department = Department("Computer Science")

    department.assign_professor(professor1)
    department.assing_prodessor(professor2)
```

**Example #2:**

```
# main.py

class DatabaseConneciton:
    def __init__(self):
        ...
    def execute_sql(self, statemnt: str):
        ...

class Logger:
    def __init__(self, db: DatabaseConnection):
        self.db = db

    def log_to_db(self, log_message: str):
        self.db.execute_sql(
            f"INSERT INTO log_table (messages) VALUES log_message"
        )

if __name__ == "__main__":
    db_conn = DatabaesConnection()
    logger = Logger(db=db_conn)
    logger.log_to_db("Hello DI")
```

## ▼ Composition vs. Aggregation

### Intro:

The key different between *composition* and *aggregation* lies in dependent objects *(components)* lifespans.

- **Composition** — dependent objects *(components)* are destroyed after the composite object is freed from the memory.
- **Aggregation** — dependent object *(components)* are still alive in the memory even after the composite object is freed.

### Difference in Implementation:

- **Composition** — to achieve composition dependent objects *(components)* are instantiated within the composite's class constructor.

- **Aggregation** — to achieve aggregation dependent objects *(components)* must be injected into composite's constructor using dependency injection (DI) techniques. This allows dependent object to remain in the memory even after the composite object is freed.

# ▼ Special (magic/dunder) Methods

> This chapter describes what methods stand behind Python regular syntax, including keywords.

## ▼ Description

### Intro:

**Special/Magic/Dunder Methods** are Python's built-in method every Python object has.

- They are not meant to be called manually.
- They are invoked automatically by Python interpreter in response to certain actions operations and syntax.
- They serve as underlying mechanism for operation overloading, attribute access, object representation, and many more.

## ▼ Standard Attributes of an Instance

### Standard Attributes:

- `__class__`: A reference to the class from which the object was instantiated.
- `__dict__`: A dictionary that stores the instance's attributes. This is where all attributes assigned to the object (e.g., `a.x = 10`) are stored.
- `__doc__`: The docstring of the class.
- `__module__`: The name of the module in which the class is defined.
- `__weakref__`: A special attribute that allows weak references to be made to the object.

## ▼ Standard Methods of an Instance

### Standard Methods:

- `__init__(self, *args, **kwargs)`: The constructor method. It's automatically called when a new instance is created. It initializes the instance's attributes.
- `__new__(cls, *args, **kwargs)`: This method is called before `__init__` and is responsible for creating and returning the new object instance.
- `__del__(self)`: The destructor method. It's called when an object's reference count drops to zero, and it is about to be garbage collected.
- `__str__(self)`: Returns a human-readable string representation of the object.
- `__repr__(self)`: Returns an "official" string representation of the object. It should be a valid Python expression that could be used to recreate the object.
- `__eq__(self, other)`: Compares two objects for equality (`==`).
- `__hash__(self)`: Returns a hash value for the object, allowing it to be used as a key in a dictionary or an element in a set.
- `__dir__(self)`: Returns a list of the object's attributes and methods.
- `__getattribute__(self, name)`: Called for every attribute access, allowing you to intercept and customize how attributes are retrieved.

- `__setattr__(self, name, value)`: Called when an attribute is assigned a value, allowing you to intercept and customize how attributes are set.
- `__delattr__(self, name)`: Called when an attribute is deleted, allowing you to intercept and customize how attributes are deleted.

# ▼ Descriptor Protocol (setter, getter, deleter)

> This chapter explains how to convert a method into an attribute and customize its reading and modification behavior.

## ▼ Descriptor Protocol

### Intro:

A **descriptor** is an object that implements one of these three dunder methods `__get__`, `__set__`, `__delete__`. When a descriptor is assigned as a class-level attribute, Python automatically invokes the appropriate dunder method of the descriptor object to handle operations on attributes.

⚠️ *The **Descriptor Protocol** works only when a descriptor object is an attribute of another class. If used in the `__init__` it'll behave as a regular object.*

### Think of Descriptors:

Instead of a simple variable, a descriptor acts as a smart variable of a proxy. When descriptor object is assigned as a class attribute, any operation on that attribute (like `instance.access` or `instance.attribute = value`) is not a direct interaction with a variable. Instead, Python intercepts the operation and delegates it to the descriptor's specific methods, which contain the logic for managing the attribute.

- **Without Descriptor:** `my_car.color` directly accesses the value "red" stored in memory.
- **With Descriptors:** `my_car.color` triggers a function call to descriptor's `__get__` method, which might check a database, validate a value, or perform a calculation before returning the color.

### Use Cases:

Descriptors are powerful tool for managing attribute access.

- **Data Validation and Type Checking:** A descriptor can enforce rules on the type and value of an attribute when it's set. For example, ensure that "price" attribute is a positive number, or "email" attribute is a valid email address. This presents invalid data assigned to an object.
- **Lazy Loading:** The first time attribute is accessed, the descriptor's `__get__` method performs the expensive operation, and cached the result to be returned for subsequent get calls.
- **Creating Managed Attributes:**
- **Object-Relational Mappers (ORMs):** Descriptors are the foundation of many ORMs like SQLAlchemy. They are used to map a Python class attributes to a column in a database table. For example, when `user.username` is accessed descriptor handles the logic of querying the database to retrieve the corresponding value.
- **The `@property` Decorator:** The most common used of descriptors. It is a Python built-in decorators that facilitates creation of getters, setters, and deleters for an attribute.
- `@classmethod` and `@staticmethod`: Built-in decorators are also implemented using the descriptor protocol. They modify how a functions is called, allowing to receive the class itself (`classmethod`) or nothing but its arguments (`staticmethod`).

### Descriptor Methods:

Or simple descriptors:

1. `__get__(self, instance, owner)` — Called when an attribute is accessed (e.g., `obj.attr`).
   - `self` — The descriptor instance itself.
   - `instance` — The object on which the attribute was accessed (e.g., `obj`). It's `None` when accessed via the class.
   - `owner` — The class to which the descriptor is attached (e.g., `OwnerClass`).
2. `__set__(self, instance, value)` — Called when an attributes is assigned a value (e.g., `obj.attr = value`).
   - `self` — The descriptor instance.
   - `instance` — The object on which the assignment was made.
   - `value` — The value being assigned.
3. `__delete__(self, instance)` — Called when an attribute is deleted (e.g., `del obj.attr`).
   - `self` — The descriptor instance.
   - `instance` — The object on which the deletion was made.

### Types of Descriptors:

1. **Data Descriptors:** A descriptor that implements either the `__set__` or `__delete__` method (or both). Because they have logic for writing or deleting an attribute, they are considered "data" descriptors.
2. **Non-Data Descriptors:** A descriptor that only implements the `__get__` method. It does not have logic for setting or deleting the attribute.

## ▼ Non-Data Descriptors

### Intro:

A **Non-Data Descriptor** is a descriptor that implements only `__get__` method. Note that descriptors van only be used as a class-attributes.

- Non-Data Descriptors have the same priority as class attributes.
- Can be used only to read data from some attribute.

### Syntax:

**Example #1:**

If a class does not have `__init__` implemented it cannot be instantiated, thus `__get__`'s `instance` is `None`.

```
# main.py

#========================
# Descriptior declaration
#========================
class MyDescriptor:
    def __get__(self, instance, owner) -> str:
        return instance.__dict__["some_attr"]

#=================
# Usage
#=================
```

```
class MyClass:
    class_attr = MyDescriptior()


if __name__ == "__main__":
    my_class = MyClass()
    my_class.class_attr # STDERR: `KeyError`
```

```
# main.py

#========================
# Descriptior declaration
#========================
class MyDescriptor:
    def __get__(self, instance, owner) -> str:
        return instance.__dict__["some_attr"]

#=================
# Usage
#=================

class MyClass:
    class_attr = MyDescriptior()

    def __init__(self):
        self.some_var = "World"


if __name__ == "__main__":
    my_class = MyClass()
    my_class.class_attr # STDOUT: "World"
```

```
# main.py

#========================
# Descriptior declaration
#========================
class MyDescriptor:
    def __get__(self, instance, owner) -> str:
        return owner.__dict__["some_attr"]

#=================
# Usage
#=================

class MyClass:
    class_attr = MyDescriptior()
      some_var = "Hello"
```

```
if __name__ == "__main__":
    my_class = MyClass()
    my_class.class_attr # STDOUT: "Hello"
```

**Example #2:**

```
# main.py

class ReadIntX:
    """
    Non-Data Descriptor.
    """

    def __set_name__(self, instance, name):
        self.name = "_x"

    def __get__(self, instance, owner):
        return getattr(instance, self.name)


class Integer:
    def __set_name__(self, owner, name: str):
        self.name = "_" + name

    def __get__(self, instance, owner):
        """
        This is the getter method which defines get behavior for the \
            Descriptor object.

        :Params:
        :``self``: Instance of the Descriptor class itself. \
            (Integer in our case).
        :``instance``: A link to the instance of a class the \
            Descriptor is instantiated in.
            (Point3D() or `point` in our case).
        :``owner``: A link to the class that the Descriptor is \
            instantiated in. (Point3D in our case).
        """

            print(f"__get__:{self.name}")

            return getattr(instance, self.name)

        def __set__(self, instance, value: int):
            """
        This is the setter method which defines the set attribute \
        behavior for the Descriptor object.

        :Params:
        :``self``: Instance of the Descriptor class itself.\
            (Integer in our case).
```

```
        :``instance``: A link to the instance of a class the Descriptor \
        is stantiated in. (Point3D() or `point` in our case).
        :``value``: The value to be set for the Descriptor object.
        """

            print(f"__set__:{self.name}={value}"
            self.validate_coord(value)
            setatter(instance, self.name, value)

        def __delete__(self, instance):
        """
        This is the deleter method which define the behavior for \
        the 'del' operation.

        :Params:
        :``self``: Instance of the Descriptor class itself.\
            (Integer in our case).
        :``instance``: A link to the instance of a class the Descriptor \
        is instantiated in. (Point3D() or `point` in our case).
        """

        print(f"__delete__:{self.name}")
        delattr(instance, self.name

        @classmethod
        def validate_coord(cls, coord: int) -> None:
                if type(coord) != int:
                    raise ValueError(
                        f"Attr must be <class: int>, got {type(coord)}"
                    )


class Point3D:
    x = Integer() # Data Descriptor
    y = Integer() # Data Descriptor
    z = integer() # Data Descriptor
    rx = ReadIntX() # Non-Data Descriptor

    def __init__(self, x: int, y: int, z: int):
        self.x = x
        self.y = y
        self.z = z


if __name__ == "__main__"
    point = Point3D(1, 2, 3)
    print(point.rx) # STDOUT: 1
    del point.x
    point.__dict__() # STDOUT: {'_y': 2, '_z': 3}
```

## ▼ Data Descriptors

## Into:

A **Data Descriptor** is a descriptor that implement one of these methods `__set__`, `__delete__` in addition to the `__get__` method. Thus, allowing to set and delete data.

## Syntax:

**Example #1:**

In this example, the setter and getter functionality relies directly on the `__dict__` attribute which provides an access to object's namespace, accordingly, it's possible mange an object's namespace using this attribute. And, the `__set_name__()` method that is invoked automatically every time an object is being instantiated, used to set a named attribute.

**Task:**

- Write a class that models a point in 3-D space.
- Each point has three coordinates: x, y, z.
- Every coordinated must be an integer, if not raise the `ValueError`.

```
# main.py

class Integer:
    def __set_name__(self, owner, name: str):
        self.name = "_" + name

    def __get__(self, instance, owner):
        """
        This is the getter method which defines get behavior for the \
            Descriptor object.

        :Params:
        :``self``: Instance of the Descriptor class itself. \
            (Integer in our case).
        :``instance``: A link to the instance of a class the \
            Descriptor is instantiated in.
            (Point3D() or `point` in our case).
        :``owner``: A link to the class that the Descriptor is \
            instantiated in. (Point3D in our case).
        """

            print(f"__get__:{self.name}")

            return instance.__dict__[self.name]

        def __set__(self, instance, value: int):
            """
        This is the setter method which defines the set attribute \
        behavior for the Descriptor object.

        :Params:
        :``self``: Instance of the Descriptor class itself.\
            (Integer in our case).
        :``instance``: A link to the instance of a class the Descriptor \
```

```
        is stantiated in. (Point3D() or `point` in our case).
        :``value``: The value to be set for the Descriptor object.
        """

            print(f"__set__:{self.name}={value}"
            self.validate_coord(value)
            instance.__dict__[self.name] = value

        @classmethod
        def validate_coord(cls, coord: int) -> None:
                if type(coord) != int:
                    raise ValueError(
                        f"Attr must be <class: int>, got {type(coord)}"
                    )


class Point3D:
    x = Integer()
    y = Integer()
    z = integer()

    def __init__(self, x: int, y: int, z: int):
        self.x = x
        self.y = y
        self.z = z

if __name__ == "__main__"
    point = Point3D(1, 2, 3)
    point.__dict__() # STDOUT: {'_x': 1, '_y': 2, '_z': 3}

    # ============================
    point = Point3D(1.1, 2.2, 3.3)
    # The `ValueError` due to the validation failure.
    point.x = 12.
    # Also, `ValueError` due to the validation failure.
    # ============================
```

**Example #2:**

In this example, the setter and getter methods rely on the Python's built-in `setattr()`, `getattr()`, and `delattr()` method. These methods provide a robust interface to manage an object's namespace (i.e. `obj.__dict__`).

```
# main.py

class Integer:
    def __set_name__(self, owner, name: str):
        self.name = "_" + name

    def __get__(self, instance, owner):
        """
```

```
        This is the getter method which defines get behavior for the \
           Descriptor object.

        :Params:
        :``self``: Instance of the Descriptor class itself. \
           (Integer in our case).
        :``instance``: A link to the instance of a class the \
           Descriptor is instantiated in.
           (Point3D() or `point` in our case).
        :``owner``: A link to the class that the Descriptor is \
           instantiated in. (Point3D in our case).
        """

           print(f"__get__:{self.name}")

           return getattr(instance, self.name)

        def __set__(self, instance, value: int):
            """
        This is the setter method which defines the set attribute \
        behavior for the Descriptor object.

        :Params:
        :``self``: Instance of the Descriptor class itself.\
           (Integer in our case).
        :``instance``: A link to the instance of a class the Descriptor \
        is stantiated in. (Point3D() or `point` in our case).
        :``value``: The value to be set for the Descriptor object.
        """

           print(f"__set__:{self.name}={value}"
           self.validate_coord(value)
           setatter(instance, self.name, value)

        def __delete__(self, instance):
        """
        This is the deleter method which define the behavior for \
        the 'del' operation.

        :Params:
        :``self``: Instance of the Descriptor class itself.\
           (Integer in our case).
        :``instance``: A link to the instance of a class the Descriptor \
        is instantiated in. (Point3D() or `point` in our case).
        """

        print(f"__delete__:{self.name}")
        delattr(instance, self.name

        @classmethod
        def validate_coord(cls, coord: int) -> None:
```

```
            if type(coord) != int:
                raise ValueError(
                    f"Attr must be <class: int>, got {type(coord)}"
                )


class Point3D:
    x = Integer()
    y = Integer()
    z = integer()

    def __init__(self, x: int, y: int, z: int):
        self.x = x
        self.y = y
        self.z = z

if __name__ == "__main__"
    point = Point3D(1, 2, 3)
    del point.x
    point.__dict__() # STDOUT: {'_y': 2, '_z': 3}
```

## ▼ The Precedence Rule and MRO

### Intro:

**The Precedence Rule** is fundamental to how Python works. When you access an attribute (`obj.attr1`), Python follows this order.

1. Lookup for `attr1` in the class's `__dict__` and see if it's a data descriptor. If so, use it's `__get__` method.
2. Lookup for `attr1` in object's *(instance's)* `__dict__`. If exits, use that value.
3. Look for `attr1` in the class's `__dict__` and see if it's non-data descriptor. If so, use its `__get__` method.
4. If none of these above are found, for for `attr` in the class's base classes (following the MRO).

### The Precedence Rule.

```
(High-priority)
      ^
   1  | |-- Class.__dict__ (data descriptor)
      |  |
   2  |  |-- Instance.__dict__ (instance attributes)
      |     |
   3  |     |-- Class.__dict__ (non-data descriptor)
      |        |
   4  |        |-- Class.__mro__ (standart class attrs)
      |
(Low-Priority)
```

`Class.__dict__` (data-descriptor) → `instance.__dict__` → `Class.__dict__` (non-data descriptor) → `Class.__mro__` (standard MRO).

## ▼ Property Decorator ( `@property` ):

## Intro:

The `@property` decorator is Python's built-in decorator that servers to facilitate usage of the descriptor protocol. This decorator becomes in handy when a class does not have much attributes.

## Semantics:

The `@property` decorator takes a method and turns it into a special kind of attribute. Under the hood, the decorator creates a data descriptor instance. This data descriptor has automatically implemented `__get__` , `__set__`, and `__delete__` methods.

## Examples:

```
# main.py

"""
This module show how to use the `@property` decorator.

The `@property` decorator is a high-level API Python provides for the \
   Descriptor Protocol.

It allows to:
   - Turn methods into a special kind of an attribute.
   - Create Setter and Getter without utilizing special methods.
"""


from decimal import Decimal


class Employee:
   def __init__(self, first_name: str, last_name: str, salary: Decimal):
      self.first_name = first_name
      self.last_name = last_name
      self.salary = salary

   @property
   def full_name(self) -> str:
      return "{} {}".format(
         self.first_name.capitalize(),
         self.last_name.capitalize()
      )

   @full_name.setter
   def full_name(self, full_name: str):
      first_name, last_name = full_name.split()

      self.first_name = first_name.capitalize()
      self.last_name = last_name.capitalize()

   @full_name.deleter
   def full_name(self):
      """
```

```
        This deleter forbids to delete Employee's name.
        """

        raise ValueError("Required Fields Cannot be Deleted")

    @property
    def email(self):
        if not "_email" in self.__dict__:
            self._email = "{}-{}@mail.com".format(
                self.first_name,
                self.last_name
            )
            return self._email

        return self._email

    @email.setter
    def email(self, *args, **kwargs):
        if args or kwargs:
            self.email = None
            return

        self.email = "{}-{}@mail.com".format(
            self.first_name,
            self.last_name
        )

    @email.deleter
    def email(self):
        self._email = None


if __name__ == "__main__":
    employee = Employee("sergei", "chabrov", 1_000_000)
    employee.full_name = "sergei2 chabrov"
    print(employee.full_name)       # STDOUT: Sergei2 Chabrov
    print(employee.email)           # STDOUT: Sergei2-Chabrov@mail.com
    del employee.email
    print(employee.email)           # STDOUT: None
    print(employee.__dict__)        # STDOUT: {
                                    # 'first_name': 'Sergei2',
                                    # 'last_name': 'Chabrov',
                                    # 'salary': 1000000,
                                    # '_email': None
                                    # }
```

**The same but using the Descriptor Protocol:**

```
# main.py

from decimal import Decimal
```

```
class FullName:
    def __set_name__(self, instance, name):
        self.name = "_" + name

    def __get__(self, instance, owner):
        if instance:
            return getattr(instance, self.name)

        return getattr(owner, self.name)

    def __set__(self, instance, value: str):
        first_name, last_name = value.split()
        first_name = first_name.capitalize()
        last_name = last_name.capitalize()

        setattr(instance, "_first_name", first_name)
        setattr(instance, "_last_name", last_name)
        setattr(instance, "_full_name", "{} {}".format(
            first_name, last_name
        ))

    def __delete__(self, instance):
        raise ValueError("Required Fields Cannot be Deleted")


class Email:
    def __set_name__(self, instance, name):
        self.name = "_" + name

    def __get__(self, instance, owner):
        if instance:
            return instance.__dict__[self.name]

        return owner.__dict__[self.name]

    def __set__(self, instance, value):
        email = "{}-{}@mail.com".format(
            instance.__dict__["_first_name"],
            instance.__dict__["_last_name"]
        )
        instance.__dict__[self.name] = email

    def __delete__(self, instance):
        if instance.__dict__[self.name]:
            instance.__dict__[self.name] = None


class Employee:
    full_name = FullName()
```

```
    email = Email()

    def __init__(self, first_name: str, last_name: str, salary: Decimal):
        self.first_name = first_name
        self.last_name = last_name
        self.salary = salary
        self.full_name = "{} {}".format(first_name, last_name)
        self.email = "123"  # The value won't be used.


if __name__ == "__main__":
    employee = Employee("sergei", "Chabrov", 1_000_000)
    print(employee.first_name)  # STDOUT: sergei
    print(employee.full_name)   # STDOUT: Sergei Chabrov
    print(employee.email)       # STDOUT: Sergei-Chabrov@mail.com
    del employee.email
    print(employee.email)       # STDOUT: None
    print(employee.__dict__)    # STDOUT: {
                                # 'first_name': 'sergei',
                                # 'last_name': 'Chabrov',
                                # 'salary': 1000000,
                                # '_first_name': 'Sergei',
                                # '_last_name': 'Chabrov',
                                # '_full_name': 'Sergei Chabrov',
                                # '_email': None
                                # }
```

# More Examples:

## ▼ Inheritance

### Definition and examples

Inheritance allows a class to inherit attributes and methods form another class.

It is useful because we can:

1. Make subclasses and get all the functionality of the parent class.
2. Overwrite and add completely functionality without effecting the parent class and its instances.

Example:

Let’s say that we want to create a new specific class of Employee. E.g.: Developers and Managers.

```
class Employee:
    pay_raise = 1.04

    def __init__(self, first, last, pay) -> None:
        self.first = first
        self.last = last
        self.pay = pay
        self.email = "{}.{}@mai.com".format(self.first, self.last)
```

```
    def raise_pay(self) -> None:
        self.pay = int(self.pay * self.pay_raise)


class Developer(Employee):
    pass
```

### Method Resolution Order (MRO)

Method resolution order is the order in which the interpreter looks for attributes and methods before accessing them. Methods and attributes with the same names can be prioritized by the resolution order.

```
print(help(Developer))
```

Using methods resolution order, we can overwrite class attribute and methods for each subclass. And they will be valid only for this subclass.

Example:

Let’s say we want to change the pay_raise attribute only for the Developers up to 100 %.

```
class Developer(Employee):
    pay_raise = 2 # Attribute overwriting for the subclass.
    pass

dev_1 = Developer('Corn', 'Boob', 30000)
print(dev_1.pay) # Stdout: 30000
dev_1.raise_pay()
print(dev_1.pay) # Stdout: 60000

dev_1 = Employee('Corn', 'Boob', 30000)
print(dev_1.pay) # Stdout: 30000
dev_1.raise_pay()
print(dev_1.pay) # Stdout: 31200
```

### Extending child instance’s attributes

We can enrich the child class instances functionality by adding new instance attribute to the child’s __init__ method. We can access the parent instances attributes and methods without copying the code form the parent class.

Example:

```
class Developer(Employee):
    pay_raise = 2

    def __init__(self, first, last, pay, prog_lang) -> None:
        super().__init__(first, last, pay) # Same as: Employee.__init__(self, first, last, pay)
        # Line above passes arguments into the parent class's __init__ method,
        # meanwhile, the child's __init__ just extend Developer's instance attributes.
        self.prog_lang = prog_lang
```

```
dev_1 = Developer('Corn', 'Boob', 30000, 'Python')
print(dev_1.first) # Stdout: 'Corn' | the 'first' attribute is handled by the Employee class.
print(dev_1.prog_lang) # Stdout: 'Python' | the 'prog_lang' attribute is handled by the Developer cla
ss.


class Manager(Employee):
    pay_raise = 1.12

    def __init__(self, first, last, pay, employees=None) -> None:
        super().__init__(first, last, pay)
        if not employees:
            self.employees = []
        else:
            self.employees = employees

    def add_emp(self, emp):
        if emp not in self.employees:
            self.employees.append(emp)

    def rem_emp(self, emp):
        if emp not in self.employees:
            print("No such employee")
        else:
            self.employees.remove(emp)

    def print_employees_names(self):
        for employee in self.employees:
            print(f"Employee: {employee.first} {employee.last} {employee.__class__}")

man_1 = Manager('Raya', 'Abbe', 32000, [dev_1])
print(man_1.print_employees_names())
man_1.rem_emp(dev_1)
print(man_1.employees)
```



To get the class name fro the class instance. class_instance.__class__.__name__

### Isinstance() and issubclass() methods

Instance method will tell us if an object is an instance of the class

Example:

```
isinstance(dev_1, Employee) #Stdout: True
isinstance(dev_1, Developer) #Stdout: True
isinstance(man_1, Developer) #Stdout: False
```

Is subclass method will tell us if a subclass of another subclass

Example:

```
issubclass(Developer, Employee) # Stdout: True
issubclass(Manager, Developer) # Stdout: False
```

## ▼ Special (magic/dunder) methods

### Definition

A Special (Magic/Dunder) methods are meant to set or change the default behavior of a class.

For example: when we use the ‘+’ symbol on integers, they will be added arithmetically, but if we use the ‘+’ symbol on strings they will be concatenated by default. As follows, the default behavior is tighten to an object special methods.

```
print(str.__add__('a', 'b')) # Stdout: 'ab'
print(int.__add__(1, 2)) # Stdout: 3
```

### Changing standard behavior

```
class Employee:
    def __init__(self, first, last, pay) -> None:
        self.first = first
        self.last = last
        self.pay = pay
        self.email = "{}.{}@mail.com".format(self.first, self.last)


emp_1 = Employee('John', 'Harries', 20000)
print(emp_1)  # Stdout: <__main__.Employee object at 0x7a1bc8657f70>
print(repr(emp_1))  # Stdout: <__main__.Employee object at 0x7258d855bf70>

# Change the behavior of the standard print method
class Employee:
    def __init__(self, first, last, pay) -> None:
        self.first = first
        self.last = last
        self.pay = pay
        self.email = "{}.{}@mail.com".format(self.first, self.last)

    def __str__(self) -> str:
        return "{} {}, {}".format(self.first, self.last, self.__class__.__name__)

    def __repr__(self) -> str:
        return "{}('{}', '{}', '{}')".format(self.__class__.__name__, self.first, self.last, self.pay)

    def __add__(self, other: Employee) -> int:
        return int(self.pay + other.pay)


emp_1 = Employee('John', 'Harries', 20000)
emp_2 = Employee('David', 'Baron', 40000)
```

```
print(emp_1)  # Stdout: "John, Harries, Employee"
print(repr(emp_1))  # Stdout: Employee(John, Harries, 20000)
print(emp_1 + emp_2) # 60000
```

> __repr__ is used to represent tech information. __str__ is used to represent user friendly information. It is a good practice to return the command the instance was created with for the __repr__ method.

One more example:

```
class A:
    def __init__(self, a) -> None:
        self.a = a

    # def __eq__(self, value: object) -> bool:
    #     Standard behavior is to compare objects bit by bit.

# Not operator == will compare length of the instance with the length of a value.
# Instead comparing bit by bit values

a = A('hello')
print(a == 'olleh') # False

class A:
    def __init__(self, a) -> None:
        self.a = a

    def __eq__(self, value: object) -> bool:
        # New behavior is to compare lengths of two objects
        return len(self.a) == len(value)

a = A('hello')
print(a == 'olleh') # True
```

## ▼ Property decorators, setters, getters, deleters

Property decorators allow to access: setter, getter and deleter functionality. Property decorator allows to define methods that can be accesses as attributes.

The problem it solves:

```
class Employee:
    def __init__(self, first, last, pay) -> None:
        self.first = first
        self.last = last
        self.pay = pay
        self.email = "{}.{}@.mail.com".format(first, last)

    def full_name(self) -> str:
        return "{} {}".format(self.first, self.last)

emp_1 = Employee('John', 'Clark', 40000)
print(emp_1.email) # Stdout: John.Clark@.mail.com
```

```
emp_1.last = 'Kirov'
print(emp_1.email) # Stdout: John.Clark@.mail.com
```

As you can see, changing the last name did not lead to the email address change. But let's suppose, that we need to change the email address every time the first or last names are changed. We can add a new instance method that allows us to receive the mail, but it will lead us to the problem where everyone who uses the Employee class must start using our new method instead if revering referring to the instance's attribute 'email'.

```
class Employee:
    def __init__(self, first, last, pay) -> None:
        self.first = first
        self.last = last
        self.pay = pay

    def full_name(self) -> str:
        return "{} {}".format(self.first, self.last)

    def email(self) -> str:
        return "{}.{}@.mail.com".format(self.first, self.last)

emp_1 = Employee('John', 'Clark', 40000)
print(emp_1.email()) # Stdout: John.Clark@.mail.com
emp_1.last = 'Kirov'
print(emp_1.email()) # Stdout: John.Kirov@.mail.com
```

To tackle this problem the property decorated comes in handy.

### Property decorator (as getter)

Note: we co not want to refactor every piece of code that refers to the 'email' attribute. To achieve the desired behavior, we can use the @property decorator, *which will allow us to refer to the method as we refer to the attribute.*

```
class Employee:
    def __init__(self, first, last, pay) -> None:
        self.first = first
        self.last = last
        self.pay = pay

    @property
    def full_name(self) -> str:
        return "{} {}".format(self.first, self.last)

    @property
    def email(self) -> str:
        return "{}.{}@.mail.com".format(self.first, self.last)

emp_1 = Employee('John', 'Clark', 40000)
print(emp_1.email) # Stdout: John.Clark@.mail.com
```

```
emp_1.last = 'Kirov'
print(emp_1.email) # Stdout: John.Kirov@.mail.com
```

Note: We do not have to use parentheses when calling the 'email' method.

### Setter

Let's say that, we want to set a new fist and last attributes for our Employee 1 instance. We can do so manually, or use the setter mechanism.

```
print(emp_1.full_name) #  John Kirov
try:
    emp_1.full_name = 'Alex Ross' # Will raise the AttributeError: can't set attribute 'full_name'
except Exception as e:
    print(e)
```

Using setter mechanism. In order to make the setter mechanism operational we need to use the property decorator.

```
class Employee:
    def __init__(self, first, last, pay) -> None:
        self.first = first
        self.last = last
        self.pay = pay

    @property
    def full_name(self) -> str:
        return "{} {}".format(self.first, self.last)

    # The setter mechanism
    @full_name.setter
    def full_name(self, name: str) -> None:
        first, last = name.split()
        self.first = first
        self.last = last

emp_1 = Employee('John', 'Clark', 40000)
print(emp_1.full_name) #  John Kirov
emp_1.full_name = 'Alex Ross' # Will NOT raise any error now.
print(emp_1.full_name)
```

### Deleter

We also can implement the deleter the same way as we did with the setter.

```
class Employee:
    def __init__(self, first, last, pay) -> None:
        self.first = first
        self.last = last
        self.pay = pay

    @property
```

```
    def full_name(self) -> str:
        return "{} {}".format(self.first, self.last)

    # The setter mechanism
    @full_name.deleter
    def full_name(self) -> None:
        self.first = None
        self.last = None

emp_1 = Employee('John', 'Clark', 40000)
print(emp_1.full_name) # Stdout: John Clark
del emp_1.full_name # Deletes the instance of the class
print(emp_1.full_name) # Stdout: None None
```

---

### Credentials:


1. https://realpython.com/python-super/#:~:text=An Overview of Python's super() Function,-If you have&text=While the official documentation is,to call that superclass's methods.
2. https://docs.python.org/3/howto/descriptor.html
3. https://www.youtube.com/watch?v=ACqsYPbgePk&ab_channel=selfedu
4. https://github.com/chabrovs/py/tree/main/RoadMap/OOP/descriptors
5. https://github.com/chabrovs/py/tree/main/RoadMap/OOP/descriptor_protocol