Программирование на Python.

Тема №4.1. Наследование.

Наследование

Все классы, которые мы рассматривали ранее – создавались «с нуля». Такой подход актуален, если сущности, которые мы описываем в программе мало похожи друг на друга.

Как только возникает необходимость, чтобы несколько классов содержали один и тот же метод (именно копию), нам необходимо применить механизм **Наследования**

Наследование

Наследование позволяет использовать уже существующий код для решения новых задач.

Один класс становится наследником другого (**суперкласса,** или **родительского класса**). Все атрибуты и методы суперкласса становятся доступны классу потомку, то есть, **наследуются!**

Пример

Класс **Employee** наследует инициализатор у класса **Person.** Внутри класса Employee описан

```
class Person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
class Employee(Person):
    def work(self):
        print(f'{self.name} работает')

new = Employee( name: 'Работяга',  age: 30)

print(new.name)
new.work()
```

допонлитель-ный метод, которого не было у супер-класса (**work**)

Множественное наследование

```
class Person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
class Employee(Person):
    def work(self):
        print(f'{self.name} работает')
class Student(Person):
    def study(self):
        print(f'{self.name} Учится')
class WorkingStudent(Employee, Student):
    pass
```

Так же – вы можете наследовать атрибуты и свойства от нескольких классов. Например – мы создали класс **Student** с методом **study** и класс **WorkingStudent,** наследующийся от классов **Employee** и **Student**

Множественное наследование

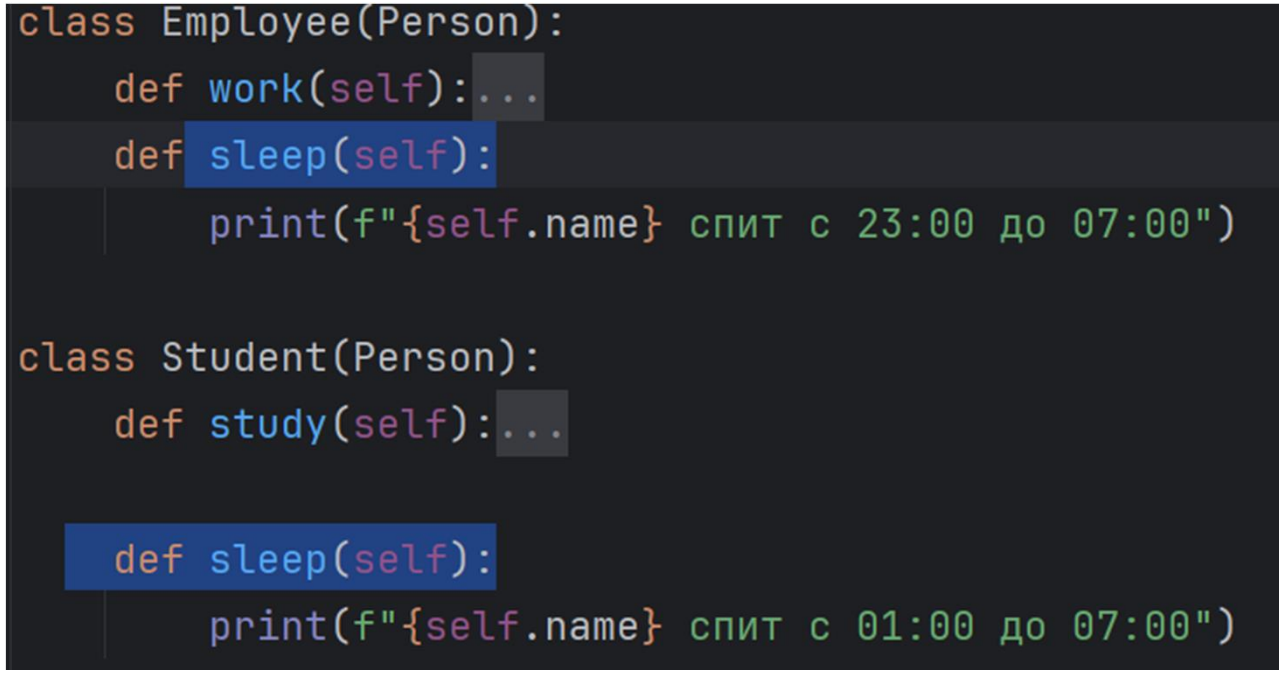
```
class WorkingStudent(Employee, Student):
    pass

ivan = WorkingStudent( name: 'Иван',  age: 18)
ivan.study()
ivan.work()
```

И хотя внутри класса **WorkingStudent** не определены новые методы или атрибуты, он получает доступ ко всем атрибутам и методам родительских классов – **Person** (т.к. Employee и Student являются дочерними для Person), **Employee**, **Student**.

Возникающие проблемы

Представим, что внутри классов **Employee** и **Student** будет метод с одним и тем же именем. Какой из этих методов унаследует дочерний класс?



Возникающие проблемы

```
class WorkingStudent(Employee, Student):
    pass

ivan = WorkingStudent( name: 'Иван',  age: 18)
ivan.sleep() #Иван спит с 23:00 до 07:00
```

При вызове метода **sleep()** мы видим результат что Иван спит с 23 до 7 часов. То есть, наследовался метод из класса **Employee.**

Так получилось, потому что в списке имен наследования при определении класса первым шел именно класс **Employee**

Mro() порядок наследования

При необходимости, мы можем программно посмотреть очередность наследования функционала базовых классов:

```
print(WorkingStudent.__mro__)
print(WorkingStudent.mro()) #или так
```

```
(<class '__main__.WorkingStudent'>, <class '__main__
↘.Employee'>, <class '__main__.Student'>, <class '__main__
↘.Person'>, <class 'object'>)
```

MRO – method Resolution Order

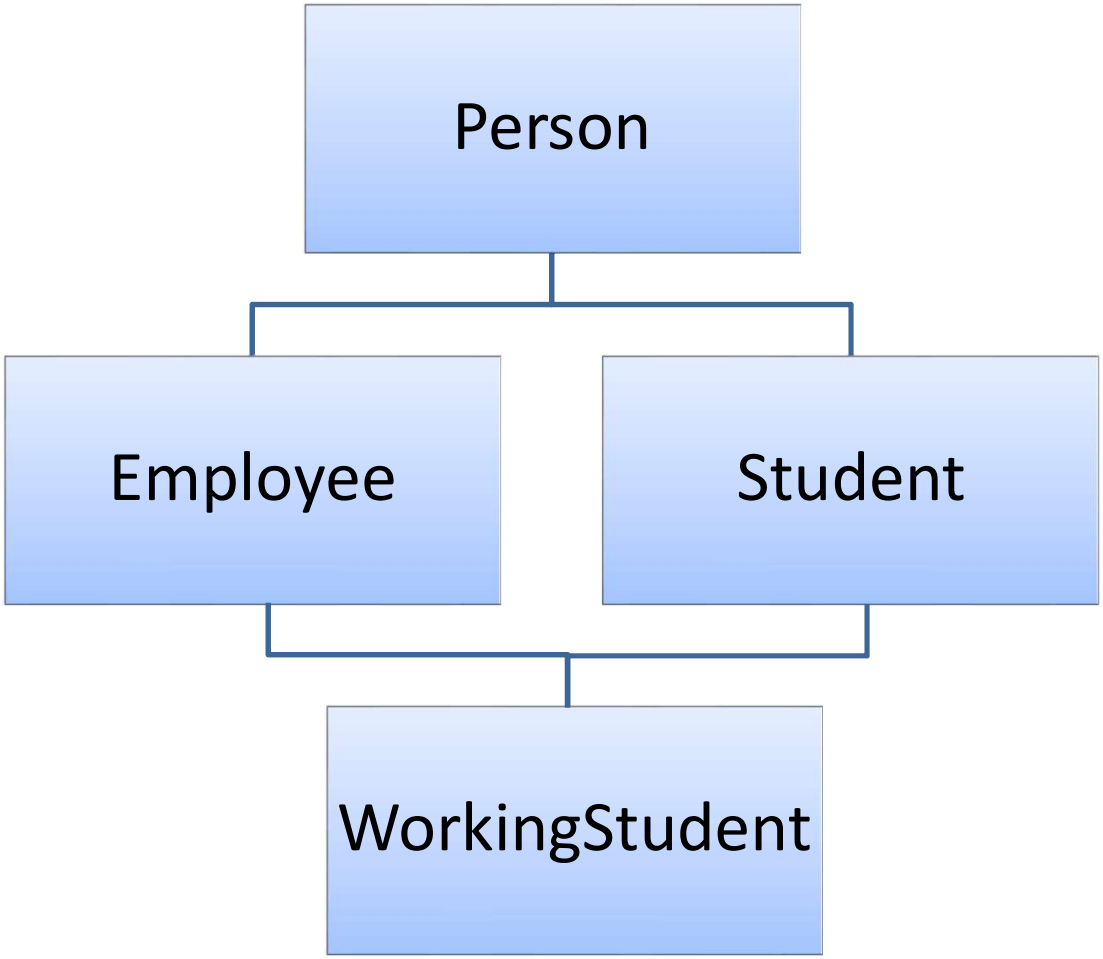
Данный метод показывает, в каком порядке Python будет искать методы и атрибуты внутри классов. То есть, при вызове какого-либо метода, Python сначала будет искать его в классе **WorkingStudent**, когда не найдет – обратится к классу **Employee**, затем в **Student**, затем в **Person**, и затем в суперкласс **Object**

(все остальные классы являются его потомками)

```
(<class '__main__.WorkingStudent'>, <class '__main__
↘.Employee'>, <class '__main__.Student'>, <class '__main__
↘.Person'>, <class 'object'>)
```

Проблема ромбов (Diamond Problem)

Описанная выше проблема всегда возникает при множественном наследовании. Если у класса несколько родителей, а у родителей есть общий предок – получаем ромб в дереве наследования.



C3 - линеаризация

Для поиска методов и атрибутов в дереве родителей Python Использует алгоритм C3- линеаризации. Упрощенно работает так:

- В список добавляются родители объекта.
  - В конец списка добавляются родители родителей и т.д.
- Если какой-то класс оказывается в списке дважды – в списке остается только последнее его вхождение.

В результате – алгортим движется по слоям, не обращаясь к классу-предку, пока не обратимся ко всем его потомкам по цепочке.

Функция Super()

Эта функция обеспечивает «кооперативное» наследование методов – если применить её во всех переопределенных методах, она обепечит вызов методов всех классов родительских классов по алгоритму MRO.

super() – это не класс родитель, а объект, позволяющий вызвать следующий по цепочке MRO класс (именно поэтому – super() не всегда вызывает родительский метод, но может вызвать метод класса-”брата”.

```
(<class '__main__.WorkingStudent'>, <class '__main__
↘.Employee'>, <class '__main__.Student'>, <class '__main__
↘.Person'>, <class 'object'>)
```

Пример super()

Снова видим проблему ромбов. При создании экземпляра **D** в терминале увидим:

```
D init
C init
B init
A init
```

```python
class A:
    def __init__(self):
        print('A init')
class B(A):
    def __init__(self):
        print('B init')
        super().__init__()
class C(A):
    def __init__(self):
        print('C init')
        super().__init__()
class D(C,B):
    def __init__(self):
        print('D init')
        super().__init__()
new = D()
```

Пример super()

Если убрать хотя бы из одного наследника (класс B) метод super(), то метод родителя не будет вызван вовсе:

```
D init
C init
B init
```

То есть, кооперативное наследование нарушается.

```
class A:
    def __init__(self):
        print('A init')
class B(A):
    def __init__(self):
        print('B init')
        #super().__init__()
class C(A):
    def __init__(self):
        print('C init')
        super().__init__()
class D(C,B):
    def __init__(self):
        print('D init')
        super().__init__()
new = D()
```

Пример super()

Причина в том, что из C.__init__ вызывается следующий по MRO класс. Ранее A.__init__ вызывал класс B, но теперь, мы закомментировали этот вызов и цепочка разорвана.

```
print(D.mro())
```

```
[<class '__main__.D'>, <class '__main__.C'>, <class
 '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

```
class A:
    def __init__(self):
        print('A init')
class B(A):
    def __init__(self):
        print('B init')
        #super().__init__()
class C(A):
    def __init__(self):
        print('C init')
        super().__init__()
class D(C,B):
    def __init__(self):
        print('D init')
        super().__init__()
new = D()
```

Применение super() на практике

super() применяется на практике для доступа к атрибутам или методам родительского класса (помним про MRO). Пример:

```
class Shape:
    def __init__(self, name):
        self.name = name
class ColoredShape(Shape):
    def __init__(self, name, color):
        super().__init__(name)
        self.color = color
new = ColoredShape( name: 'квадрат',  color: 'красный')
print(new.name, new.color)
```

Пример

В данном примере мы вызываем родительский метод __init__, чтобы выполнить все необходимые инструкции метода __init__ (сохранить значение атрибута **name**). Затем – мы дополняем метод __init__ дочернего класса новыми инструкциями (сохраняем новый атрибут **color**)

```
class Shape:
    def __init__(self, name):
        self.name = name
class ColoredShape(Shape):
    def __init__(self, name, color):
        super().__init__(name)
        self.color = color
new = ColoredShape( name: 'квадрат',  color: 'красный')
print(new.name, new.color)
```

Упражнение

Создайте класс **Pasport**, принимающий при инициализации номер паспорта в формате **** ****** и проверяющий правильность ввода данных (длина, цифры от 0 до 9) специальным метод **validate_passport().** Если данные введены корректно, номер паспорта сохраняется в атрибуте **passport_number**.

Создайте класс **Citizen**, Принимающий в инициализаторе ФИО человека и номер паспорта, а также вызывающий родительский метод __init__() для проверки правильности ввода данных паспорта и создания атрибута passport_number.

Ограничения super()

Важным **ограничением** является **невозможность выполнения операций над возвращаемым командой super() объектом**, даже если в родительском классе эти операции описаны при помощи «магических методов» – тех самых, которые мы рассматривали при перегрузке операторов.

То есть, мы можем выполнить операцию над экземпляром родительского класса, и Python найдет в родительском классе нужный метод. НО если попытаться сделать то же самое над объектом, возвращаемым методом super(), то получим ошибку.

Пример ограничений

Прямое обращение по индексу – находит нужный метод. При индексации через super – ошибка:

```
class Parent:
    #Функция обращения по индексу [] для примера
    def __getitem__(self, index):
        return index
class Child(Parent):
    def super_index(self,index):
        return super()[index]


child_instance = Child()
print(child_instance[0])                #работает
print(child_instance.super_index(0))#ошибка
```

Переопределение функционала суперкласса

В предыдущем упражнении вы переопределили функционал родительского метода __init__ добавив к его инструкциям дополнительные действия. Таким же образом можно переопределять и другие методы родительского класса.

Добавим еще один метод родительскому классу и рассмотрим пример.

Переопределение функционала суперкласса

Мы добавили еще одну команду **print** к тому, что делал родительский метод **show_info:**

```
class Shape:
    def __init__(self, name):
        self.name = name
    def show_info(self):
        print('Моя форма:', self.name)
class ColoredShape(Shape):
    def __init__(self, name, color):
        super().__init__(name)
        self.color = color
    def show_info(self):
        super().show_info()
        print('Мой цвет:', self.color)

new = ColoredShape( name: 'квадрат',  color: 'красный')
new.show_info()
```

Зачем?

Мы могли реализовать метод **show_info** дочернего класса таким образом:

```
def show_info(self):
    print('Моя форма:', self.name)
    print('Мой цвет:', self.color)
```

Такое решение имеет ряд недостатков:

- Код дочернего метода повторяет код родительского, что противоречит принципу DRY (don’t repeat yourself)
- При изменении метода в базовом классе – придется вручную вносить изменения в дочерний класс, иначе, совместимость этих объектов может нарушиться

Доп. Информация: класс **object**

Все классы в Python имеют один общий суперкласс –**object.** Все классы по-умолчанию наследуют его методы. То есть, класс **object** – является корнем дерева классов. Когда мы вызываем список атрибутов **dir()** для самодельных классов – мы видим перечень атрибутов класса **object**

Подробнее: help(object)

Упражнение

1. Создайте класс **Vehicle** с атрибутами **max_speed** и **mileage**.
2. Создайте класс **Bus**, наследующий атрибуты класса **Vehicle**, и получающий атрибуты **name**, **max_capacity** и **occupied_places, ride_fare** (стоимость проезда)**, cash** (деньги на руках у водителя).

Определите метод **add_passengers**, добавляющий значение к значению **occupied_places**, и проверяющий что количество занятых мест не превышает общей вместимости автобуса.

Определите метод **seating_capacity**() отображающий информацию о количестве свободных мест в формате свободно **occupied_places** из **max_capacity** мест.

Определите метод **collect_fare(),** при вызове которого пассажиры обилечиваются, и сумма денег за проезд отправляется в **cash.**

Упражнение

Создайте класс **SchoolBus** (бесплатный школьный автобус), наследующий атрибуты и методы класса **Bus**, и переопределяющий значение атрибута **ride_fare = 0.** Так же, класс должен переопределять поведение родительского метода **collect_fare.** Вместо сбора денег и отправки в **cash** – данный метод должен выводить сообщение «это бесплатный школьный автобус»

Упражнение

Существует несколько классов объектов: столы, шкафы, тумбочки. Каждый из них имеет свои поля, методы и внутреннюю логику работы. Тумбочки могут выполнять те же задачи, как и столы и шкафы одновременно, а также имеют свои собственные специфические методы и атрибуты.

Напишите ООП-модель, описывающую столы, шкафы и тумбочки.

Пользователь не должен знать ничего о внутренних механизмах работы классов. Он должен использовать методы **put_on_top()** для того, чтобы положить что-то на стол или тумбочку, или метод **put_inside()**, чтобы положить что-то в шкаф или в тумбочку.

Реализуйте классы, создав уникальные и общие атрибуты и интерфейсы.

Миксин-классы

В программировании есть еще один термин, связанный с наследованием: **Миксины (или Mixins – от англ. Mix in - примесь) .** Такие классы представляют собой особые простые классы, которые включают в себя набор методов, предназначенных для добавления другим классам, но не для самостоятельного использования.

Они позволяют расширять функциональность классов без глубокой иерархии наследования (по смыслу – как декораторы для функций)

Для чего нужны миксины

Миксины создаются для того, чтобы предоставлять функции множеству классов. Миксины не предполагают создание объектов и хранение состояния.

Это позволяет создавать гибкие инструменты для улучшения и модификации структуры кода.

Пример Миксина

Допустим, у нас есть несколько классов, представляющих разные типы медиафайлов. Добавим в них возможность воспроизведения:

```
class PlayableMixin:
    def play(self):
        print(f"Воспроизвожу {self.__class__.__name__} в {self.format} формате.")
class VideoFile:
    format = "MP4"
class AudioFile:
    format = "MP3"

class PlayableVideoFile(VideoFile, PlayableMixin):
    pass
class PlayableAudioFile(AudioFile, PlayableMixin):
    pass

video = PlayableVideoFile()
video.play()  #Воспроизвожу PlayableVideoFile в MP4 формате.
```