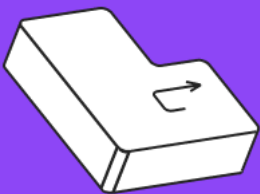




Лекция 10.

ООП. Начало

Погружение в Python



Оглавление

На этой лекции мы	2
Дополнительные материалы к лекции	2
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
Подробный текст лекции	
1. Основы ООП в Python	4
Атрибуты класса и экземпляров	6
Параметр self	9
Передача аргументов в экземпляр	9
Методы класса	10
2. Инкапсуляция	12
Одно подчёркивание в начале	13
3. Наследование	17
4. Полиморфизм	25

На этой лекции мы

1. Разберёмся с объектно-ориентированным программированием в Python.
2. Изучим особенности инкапсуляции в языке
3. Узнаем о наследовании и механизме разрешения множественного наследования.
4. Разберёмся с полиморфизмом объектов.

Дополнительные материалы к лекции

1. C3 линейаризация

<https://ru.wikipedia.org/wiki/C3-%D0%BB%D0%B8%D0%BD%D0%B5%D0%B0%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F>

2. MRO Python <https://www.python.org/download/releases/2.3/mro/>

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрали замыкания в программировании
2. Изучили возможности Python по созданию декораторов
3. Узнали как создавать декораторы с параметрами
4. Разобрали работу некоторых декораторов из модуля `functools`

Термины лекции

- **Объектно-ориентированное программирование (сокр. ООП)** — методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.
- **Инкапсуляция** — свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе.
- **Наследование** — свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствованной функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом.
- **Полиморфизм** — свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.
- **Класс** — универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является

моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей).

Подробный текст лекции

1. Основы ООП в Python

Как вы помните, что в Python всё объект. Объекты строятся на основе классов. Объекты также называют экземплярами класса. Например выполняя следующий код вы создаёте экземпляр класса list обратившись к классу list

```
data = list((1, 2, 3))
print(f'{data = }, {type(data) = }, {type(list) = }')
```

Тип функции list — type. Перед нами класс языка Python. Привычные функции для создания базовых объектов Python на самом деле являются классами: list, tuple, int, str и т.д.

При разработке языка было введено минимум новых команд и действия для написания кода в ООП стиле похожи на написание кода в функциональном стиле. Отличия и особенности разберём далее.

Классы

Вы уже знаете, что в Python есть несколько областей видимости. Например переменные объявленные внутри функции, являются локальными переменными функции. Так же вы помните о модулях, которые могут хранить в себе переменные и функции. Для обращение к ним используется импорт и точечная нотация:

```
import random
import supermodule

result1 = random.randint(1, 10)
result2 = supermodule.randint(42)
```

При этом разные модули могут содержать одноимённые функции и переменные. Это не будет вызывать ошибок, т.к. пространство имён разграничивает их по областям видимости.

Класс Python это ещё один способ создать локальную область видимости, поместив в неё переменные и функции класса. При этом для обращения к ним так же используется точечная нотации. Зачем же вводить классы, если они ведут себя как модули? Классы позволяют создавать свои экземпляры. Количество экземпляров зависит от решаемой задачи. И у каждого экземпляра будут свои переменные и функции со своими значениями. Каждый экземпляр класса создаёт свою локальную область видимости.

Представим, что мы пишем компьютерную игру. Создадим класс `Person`. Будем рассматривать особенности ООП на его примере в рамках занятия.

```
class Person:
    pass
```

Зарезервированное слово `class` указывает на создание нового класса. Далее указываем имя класса и ставим двоеточие. Тело класса записывается с отступами относительно его определения.



PEP-8! Имя класса принято записывать в стиле TitleCase, т.е. первая буква заглавная, а остальные строчные. Если название класса состоит из нескольких слов, они записываются слитно без использования символа подчеркивания. Каждое слово с большой буквы: `class MyBestSuperData`



PEP-8! Определение класса записывается в начале файла, после импортов и констант уровня модуля. До и после класса оставляют по две пустых строки.

Экземпляры класса

Для создания экземпляра класса необходимо выполнить операцию присваивания вызвав класс. Точно так же как в примере со списком `list`.

```
class Person:
```

```
max_up = 3
```

```
p1 = Person()  
print(p1.max_up)
```

Теперь переменная `p1` является экземпляром класса `Person`. Мы можем обращаться в переменным класса из экземпляра.

Обычно в литературе приводят сравнение класса с чертежом, а его экземпляра с объектом, созданным на основе чертежа. Для Python так же будет верно сравнение класса с прототипом, а экземпляра с серийным объектом, созданным на основе прототипа. Класс — такой же объект, как и экземпляр. С ним так же можно взаимодействовать. Это не просто чертёж на бумаге.

```
print(Person.max_up)
```

Атрибуты класса и экземпляров

Переменная `max_up` в классе считается атрибутом класса. В некоторой литературе такие переменные называют свойствами класса. Также иногда используют термин поля класса. Свойства позволяют хранить значения и переходят ко всем экземплярам класса.

Рассмотрим несколько особенностей работы с атрибутами.

```
class Person:  
    max_up = 3  
  
p1 = Person()  
p2 = Person()  
print(f'{Person.max_up = }, {p1.max_up = }, {p2.max_up = }')  
p1.max_up = 12  
print(f'{Person.max_up = }, {p1.max_up = }, {p2.max_up = }')  
Person.max_up = 42  
print(f'{Person.max_up = }, {p1.max_up = }, {p2.max_up = }')
```

Изначально у класса и двух его экземпляров значения `max_up` совпадают. Экземпляры “не видят” `max_up` у себя и берут значение у класса.

После изменения свойства у экземпляра p1 мы видим новое значение у него, но старые у класса и экземпляра p2. Изменения экземпляра всегда затрагивают этот экземпляр.

Далее мы меняем значение свойства у класса. Экземпляр p2 “не видит” max_up у себя, поэтому снова обращается к классу и возвращает новое значение. Экземпляр p1 имеет собственную локальную переменную max_up, поэтому его свойство не изменилось. Изменения свойств класса затрагивают те экземпляры, которые не имеют собственных свойств.

Динамическая структура класса

Класс и экземпляр являются динамическими объектами. Мы можем добавлять атрибуты в процессе работы, а не только в момент создания класса.

```
class Person:
    max_up = 3

p1 = Person()
p2 = Person()
Person.level = 1
print(f'{Person.level = }, {p1.level = }, {p2.level = }')
p1.health = 100
print(f'{Person.health = }, {p1.health = }, {p2.health = }') #
AttributeError: type object 'Person' has no attribute 'health'
print(f'{p1.health = }, {p2.health = }') # AttributeError:
'Person' object has no attribute 'health'
print(f'{p1.health = }')
```

Добавление свойства level для класса позволяет обращаться к нему и из экземпляров.

Когда же мы добавляем свойство health для экземпляра p1, получаем ошибку AttributeError. Ни класс, ни экземпляр p2 не могут получить доступ к данному атрибуту.

Возможность динамически изменять класс может быть использована как аналог работы со словарями dict.

```
class Person:
    pass
```

```

p1 = Person()
p1.level = 1
p1.health = 100

dict_p1 = {}
dict_p1['level'] = 1
dict_p1['health'] = 100

print(f'{p1.health = }')
print(f'{dict_p1["health"] = }')

```

Если в словаре мы указываем строковой ключ в квадратных скобках, в экземпляре достаточно точечной нотации без лишних скобок и кавычек.

Конструктор экземпляра

При создании класса обычно используют функцию конструктор `__init__`.

```

class Person:
    max_up = 3

    def __init__(self):
        self.level = 1
        self.health = 100

p1 = Person()
p2 = Person()
print(f'{p1.max_up = }, {p1.level = }, {p1.health = }')
print(f'{p2.max_up = }, {p2.level = }, {p2.health = }')
print(f'{Person.max_up = }, {Person.level = }, {Person.health = }')
# AttributeError: type object 'Person' has no attribute 'level'
Person.level = 100
print(f'{Person.level = }, {p1.level = }, {p2.level = }')

```

Внутри класса создаём функцию `__init__`. Два символа подчёркивания до и после имени говорят о том, что это “магическое имя”. Подобные имена нужны для добавления новых возможностей в работе класса и его экземпляров.

Внутри функции заданы две переменные `level` и `health`. Это атрибуты экземпляров. Любой экземпляр получает заранее присвоенные значения. При этом сам класс не имеет доступа к заданным атрибутам.


Также при попытке определить свойства `level` у класса мы не меняем значения экземпляров. Это разные объекты, находящиеся в разных локальных областях видимости.

- **Параметр `self`**

Ещё раз посмотрим на код конструктора:

```
def __init__(self):  
    self.level = 1  
    self.health = 100
```

В качестве параметра указана переменная `self`. Далее мы не просто присваиваем значения переменным, а указываем `self` с точечной нотацией. В работе с классами `self` является указателем на тот экземпляр класса, к которому происходит обращение. Например для `p1` это `p1.level = 1`. Какое бы имя вы не дали экземпляру, `self` подставляет его на своё место.

 **PEP-8!** Имя `self` не является зарезервированным. Вместо него можно использовать любое. Но соглашение о написании кода требует писать `self`. Так ваш код поймут другие разработчики, а IDE верно его проанализируют.

В некоторых языках при написании кода используется запись `this.name`. При некотором допущении можно считать, что Python использует вместо `this` слово `self` с той же логикой.

- **Передача аргументов в экземпляр**

При создании экземпляра можно передать значения в конструктор и тем самым добавить свойства, характерные для конкретного экземпляра.

```
class Person:  
    max_up = 3  
  
    def __init__(self, name, race='unknown'):  
        self.name = name  
        self.race = race  
        self.level = 1  
        self.health = 100  
  
p1 = Person('Сильвана', 'Эльф')  
p2 = Person('Иван', 'Человек')  
p3 = Person('Грогу')  
print(f'{p1.name} = {p1.race}')  
print(f'{p2.name} = {p2.race}')
```

```
print(f'{p3.name = }, {p3.race = }')
```

У `__init__` определено три параметра. При этом первый параметр всегда `self` и он не учитывается при передаче аргументов. Вызывая класс ожидаются два параметра, при этом второй имеет значение по умолчанию. За исключением `self` логика такая же как и при создании обычной функции. Все изученные в теме функции знания применимы и к функциям класса.

Методы класса

Функция внутри класса называется методом. Мы уже рассмотрели магический метод `__init__`, который вызывается при создании экземпляра класса. Помимо этого можно создавать любые методы внутри класса и обращаться к ним из экземпляра через точечную нотацию. Различие между обращением к свойству и к методу - круглые скобки после имени.

```
class Person:
    max_up = 3

    def __init__(self, name, race='unknown'):
        self.name = name
        self.race = race
        self.level = 1
        self.health = 100

    def level_up(self):
        self.level += 1

p1 = Person('Сильвана', 'Эльф')
p2 = Person('Иван', 'Человек')
p3 = Person('Грогу')
print(f'{p1.level = }, {p2.level = }, {p3.level = }')
p3.level_up()
p1.level_up()
p3.level_up()
print(f'{p1.level = }, {p2.level = }, {p3.level = }')
```

Метод `level_up` берёт значение `level` у экземпляра, который вызвал этот метод и увеличивает на единицу. Работа метода не затрагивает другие экземпляры.



PEP-8! Между методами класса оставляется по одной пустой строке до и после. Как вы помните в модуле до и после функции оставляют по две пустые строки.

При желании можно передавать в метод аргументы. И так как в Python всё объект, можно передать даже экземпляр класса.

```
class Person:
    max_up = 3

    def __init__(self, name, race='unknown'):
        self.name = name
        self.race = race
        self.level = 1
        self.health = 100

    def level_up(self):
        self.level += 1

    def change_health(self, other, quantity):
        self.health += quantity
        other.health -= quantity

p1 = Person('Сильвана', 'Эльф')
p2 = Person('Иван', 'Человек')
p3 = Person('Грогу')
print(f'{p1.health = }, {p2.health = }, {p3.health = }')
p1.change_health(p2, 10)
print(f'{p1.health = }, {p2.health = }, {p3.health = }')
```

Метод `change_health` принимает ещё один экземпляр и количество здоровья. Атрибут надо изменить и у экземпляра, вызвавшего метод и у второго, переданного экземпляра.



Внимание! Чаще всего для указания на другой экземпляр того же класса используют параметр `other` в имени метода. Соответственно записи `other.name` аналогичны `self.name`, но изменяют другой, переданный экземпляр класса.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
class User:
    count = []

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

u1 = User('One', '123-45-67')
u2 = User('NoOne', '76-54-321')

u1.count.append(42)
u1.count.append(73)

u2.counter = 256
u2.count.append(u2.counter)
u2.count.append(u1.count[-1])

print(f'{u1.name = }, {u1.phone = }, {u1.count = }')
print(f'{u2.name = }, {u2.phone = }, {u2.count = }')
```

2. Инкапсуляция

В ряде языков программирования под инкапсуляцией понимается сокрытие части свойств и методов класса от доступа извне класса. Как вы понимаете пользователи программы не пишут код. Они используют графический или консольный интерфейс. Или даже пользуются программой даже не догадываясь об этом, к примеру получая данные от программы-сервера во время сёрфинга в интернете. Следовательно инкапсуляция нужна одним разработчикам для сокрытия от других разработчиков и самих себя.

Python исповедует принципы разумного программирования. Если один разработчик решает внести изменения в код другого, он понимает что делает. В результате в Python нет строгой инкапсуляции как в ряде других языков. Часть инкапсуляции прописана как соглашения. И мы сталкивались с ними, когда разбирали модули и импорт переменных и функций из модулей.

Модификаторы доступа

В классическом ООП выделяют следующие модификаторы доступа:

- `public` — публичный доступ, т.е. возможность обратиться к свойству или методу из любого другого класса и экземпляра
- `protected` — защищённый доступ, позволяющий обращаться к свойствам и методам из класса и из классов наследников.
- `private` — приватный доступ, т.е. отсутствие возможности обратиться к свойству или методу из другого класса или экземпляра.

В Python по умолчанию все свойства и методы публичные. Однако существуют соглашения по стилю имён, которые делают атрибуты защищёнными/приватным. Речь в очередной раз пойдёт о символе подчёркивания.

Одно подчёркивание в начале

Одиночный символ подчёркивания в начале имени свойства или метода говорит о том, что он защищён. Мы просим других разработчиков не менять значения защищённых свойств и не вызывать защищённые методы.

```
class Person:
    max_up = 3
    _max_level = 80

    def __init__(self, name, race='unknown', speed=100):
        self.name = name
        self.race = race
        self.level = 1
        self.health = 100
        self._speed = speed

    def _check_level(self):
        return self.level < self._max_level

    def level_up(self):
        if self._check_level():
            self.level += 1
```

```

def change_health(self, other, quantity):
    self.health += quantity
    other.health -= quantity

p1 = Person('Сильвана', 'Эльф', 120)
p2 = Person('Иван', 'Человек')
p3 = Person('Грогу', speed=60)
print(f'{p1._max_level = }')
print(f'{p2._speed = }')
p2._speed = 150
print(f'{p2._speed = }')

p3.level_up()
print(f'{p3.level = }')
p3.level = 80
p3.level_up()
print(f'{p3.level = }')

```

Переменная уровня класса `_max_level` и переменная уровня экземпляра `_speed` говорят другим разработчикам, что они защищены. Так мы просим их не использовать. Однако мы сможем обратиться к ним через точечную нотацию. И даже изменить, если очень захотим. Скорее всего IDE будет указывать на неверные действия с подобными переменными.

Аналогично метод `_check_level` говорит о том, что он защищён. Метод нужен классу для проверки достижения максимального уровня персонажа и не должен использоваться напрямую вне класса. А вот его вызов из метода `level_up` считается нормальным поведением.

Два подчёркивания в начале

Двойное подчёркивание делает свойство или метод приватным. При этом срабатывает механизм защиты имени за пределами класса.



Важно! Переменная с двумя подчёркиваниями в начале не может иметь более одного подчёркивания в конце имени. Двойное подчёркивание до и после имени — магическая переменная Python. Подобно `__init__` такие имена зарезервированы для особых действий.

```
class Person:
```

```

__max_up = 3
__max_level = 80

def __init__(self, name, race='unknown', speed=100):
    self.name = name
    self.race = race
    self.level = 1
    self.health = 100
    self._speed = speed
    self.up = 3

def _check_level(self):
    return self.level < self._max_level

def level_up(self):
    if self._check_level():
        self.level += 1

def change_health(self, other, quantity):
    self.health += quantity
    other.health -= quantity

def add_up(self):
    self.up += 1
    self.up = min(self.up, self.__max_up)

p1 = Person('Сильвана', 'Эльф', 120)
print(f'{p1.up = }')
p1.up = 1
print(f'{p1.up = }')
for _ in range(5):
    p1.add_up()
    print(f'{p1.up = }')

print(p1.__max_up)      # AttributeError: 'Person' object has no
                        # attribute '__max_up'
print(Person.__max_up)  # AttributeError: type object 'Person'
                        # has no attribute '__max_up'

```

Переменная `__max_up` доступна внутри класса и его экземпляров. Мы используем её для увеличения количества жизней персонажа в методе `add_up`. Никаких проблем с доступом нет.

Когда же пытаемся обратиться к свойству напрямую, получаем ошибку доступа к атрибуту. Аналогичные ошибки будут и при обращении к методу, начинающемуся с двух подчёркиваний.

Доступ к приватным переменным

Приватная переменная `__max_up` не исчезает за пределами класса. Срабатывает механизм модификации имени. В общем случае он превращает переменную `__name` в переменную `_classname__name`.

```
class Person:
    __max_up = 3
    ...

print(f'{p1._Person__max_up = }')
```

В нашем примере переменная `__max_up` превратилась в `_Person__max_up`. Обратите внимание на заглавную `P` в имени, ведь Python является регистрозависимым языком.

Важно! У вас должны быть особые обстоятельства чтобы обращаться и тем более изменять значения переменных, которые начинаются с двух подчёркиваний за пределами их класса. Скорее всего есть другой способ решить вашу задачу.

Задание

Перед вами несколько строк кода. Какие ошибки и недочёты есть в коде. У вас 3 минуты.

```
class User:

    def __init__(self, name, phone, password):
        self.__name__ = name
        self._phone = phone
        self.__password = password

u1 = User('One', '123-45-67', 'qwerty')

print(f'{u1.__name__ = }, {u1._phone = }, {u1._User__password = }')
```


3. Наследование

В Python все объекты являются наследниками класса object. В начале лекции мы рассмотрели пример создания класса:

```
class Person:
    pass
```

Представленная запись создания класса является упрощённой. На самом деле класс Person наследуется от класса object. Т.е. object — родительский класс для Person, а Person — дочерний класс для object.

```
class Person(object):
    pass
```



Важно! Наследование от object принято опускать при создании класса.

Создадим класс героя на основе класса персонажа.

```
class Person:
    __max_up = 3
    _max_level = 80

    def __init__(self, name, race='unknown', speed=100):
        self.name = name
        self.race = race
        self.level = 1
        self.health = 100
        self._speed = speed
        self.up = 3

    def _check_level(self):
        return self.level < self._max_level

    def level_up(self):
        if self._check_level():
            self.level += 1

    def change_health(self, other, quantity):
        self.health += quantity
        other.health -= quantity
```

```

def add_up(self):
    self.up += 1
    self.up = min(self.up, self.__max_up)

class Hero(Person):
    def __init__(self, power, *args, **kwargs):
        self.power = power
        super().__init__(*args, **kwargs)

p1 = Hero('archery', 'Сильвана', 'Эльф', 120)
print(f'{p1.name = }, {p1.up = }, {p1.power = }')

```

Класс Person мы не изменяли. Он перенесён из главы про инкапсуляцию. Далее мы создаём класс Hero и указываем в скобках класс Person. Герой - дочерний класс для персонажа. Мы хотим добавить герою свойство power и прописываем его в методе инициализации. Далее вызываем метод super().__init__, т.е. метод инициализации родительского класса. Без такого вызова не будут созданы атрибуты родительского класса.

Теперь при создании экземпляра класса Hero мы вначале передаём его аргументы, а далее аргументы родительского класса Person.

Переопределение методов

При наследовании мы можем использовать в дочернем классе все общедоступные свойства и методы родительского класса. Кроме того можно создать свои. И если имена будут совпадать, произойдёт переопределение. Будут браться значения дочернего класса.

```

class Person:
    ...

class Hero(Person):
    def __init__(self, power, *args, **kwargs):
        self.power = power
        super().__init__(*args, **kwargs)

    def change_health(self, other, quantity):

```

```

        self.health += quantity * 2
        other.health -= quantity * 2

    def add_many_up(self):
        self.up += 1
        self.up = min(self.up, self._Person__max_up * 2)

p1 = Hero('archery', 'Сильвана', 'Эльф', 120)
p2 = Person('Маг', 'Троль')

print(f'{p1.health = }, {p2.health = }')
p1.change_health(p2, 10)
print(f'{p1.health = }, {p2.health = }')
p2.change_health(p1, 10)
print(f'{p1.health = }, {p2.health = }')

p1.add_many_up()
print(f'{p1.up = }')

```

В примере создан метод `change_health` с дополнительным множителем. Он срабатывает у героя. Но при вызове метода у экземпляра класса `Person` срабатывает старый метод.

В методе `add_many_ups` для обхода инкапсуляции используем запись `self._Person__max_up`. Экземпляр обращается к приватному атрибуту родительского класса, напрямую указав его.

Множественное наследование

Python поддерживает множественное наследование. Класс может быть наследником сразу двух и более классов. В некоторых языках множественное наследование недоступно по причине усложнения кода. Например наследуя класс существо от классов птицы и рыбы позволит создать летающую рыбу или плавающую птицу?

Рассмотрим вариант множественного наследования.

```

class Person:
    __max_up = 3
    __max_level = 80

    def __init__(self, name, race='unknown', speed=100):

```

```

        self.name = name
        self.race = race
        self.level = 1
        self.health = 100
        self._speed = speed
        self.up = 3

    def _check_level(self):
        return self.level < self._max_level

    def level_up(self):
        if self._check_level():
            self.level += 1

    def change_health(self, other, quantity):
        self.health += quantity
        other.health -= quantity

    def add_up(self):
        self.up += 1
        self.up = min(self.up, self.__max_up)

class Address:
    def __init__(self, country, city, street):
        self.country = country or ''
        self.city = city or ''
        self.street = street or ''

    def say_address(self):
        return f'Адрес героя: {self.country}, {self.city}, {self.street}'

class Weapon:
    def __init__(self, left_hand, right_hand):
        self.left_hand = left_hand or 'Клинок'
        self.right_hand = right_hand or 'Лук'

class Hero(Person, Address, Weapon):
    def __init__(self, power, name=None, race=None, speed=None,
                  country=None, city=None, street=None,
                  left_hand=None, right_hand=None):
        self.power = power
        Person.__init__(self, name, race, speed)
        Address.__init__(self, country, city, street)

```

```

        Weapon.__init__(self, left_hand, right_hand)

    def change_health(self, other, quantity):
        self.health += quantity * 2
        other.health -= quantity * 2

    def add_many_ups(self):
        self.up += 1
        self.up = min(self.up, self._Person__max_up * 2)

p1 = Hero('archery', 'Сильвана', 'Эльф', 120,
          country='Эльфляндия', street='Ночного эльфа',
          left_hand='Стрела')

print(f'{p1.say_address()}')
print(f'{p1.right_hand = }, {p1.left_hand = }')

```

Мы создали классы Address и Weapon. Добавив их к нашему герою, получаем сочетание атрибутов и методов всех перечисленных классов. Обратите внимание на то как происходит инициализация родительских классов внутри Hero __init__. Прописали все параметры из родительских классов в инициализации класса Hero. Далее вручную распределяем аргументы между методами __init__ каждого из родительских классов. Подобный приём не лучшая практика. При изменении параметров у родительских классов, дочерние могут перестать работать. При простых реализациях наследования достаточно функции super(). Обычно не стоит усложнять код до того состояния, когда внутренние механизмы не справляются с наследованием.

MRO

Аббревиатура MRO — method resolution order переводится как “порядок разрешения методов”. Относится этот порядок не только к методам, но и ко всем атрибутам класса. Это внутренний механизм, определяющий порядок наследования.

Забегая вперёд, иногда механизм не справляется с задачей. И чаще всего это говорит о сложности кода и неверной логики построения наследования. Т.е. нерабочий механизм наследования намекает разработчику на проблемы в его коде. Рассмотрим работу MRO на нескольких простых классах.

```

class A:
    def __init__(self):

```

```

        print('Init class A')
        self.data_a = 'A'

class B:
    def __init__(self):
        print('Init class B')
        self.data_b = 'B'

class C:
    def __init__(self):
        print('Init class C')
        self.data_c = 'C'

class D:
    def __init__(self):
        print('Init class D')
        self.data_d = 'D'

class X1(A, C):
    def __init__(self):
        print('Init class X1')
        super().__init__()

class X2(B, D):
    def __init__(self):
        print('Init class X2')
        super().__init__()

class X3(A, D):
    def __init__(self):
        print('Init class X3')
        super().__init__()

class Z(X1, X2, X3):
    def __init__(self):
        print('Init class Z')
        super().__init__()

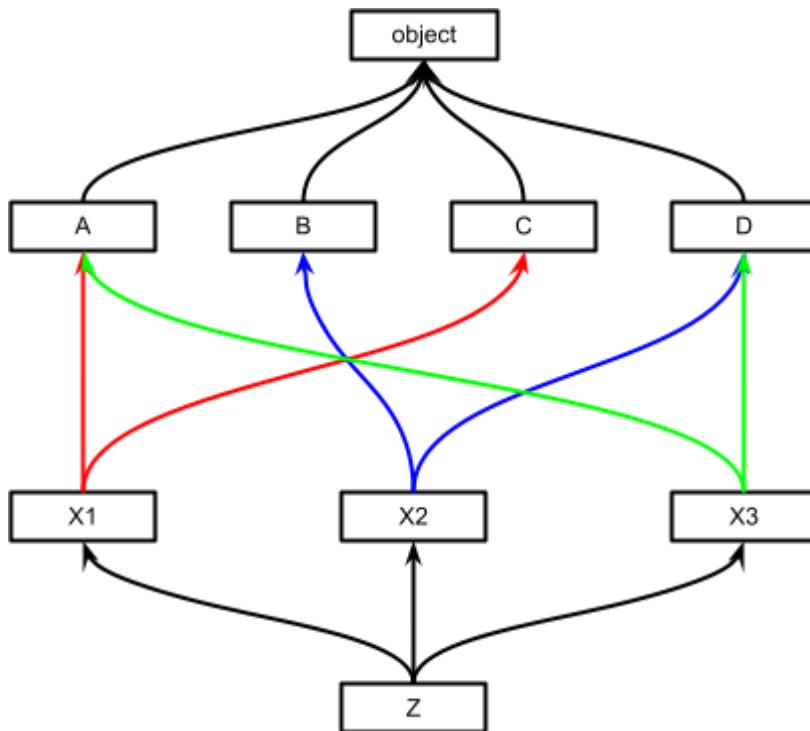
print(*Z.mro(), sep='\n')

```

1. Четыре класса A, B, C, D не имеют родительского класса. Точнее они наследуются от прародителя object. У каждого из классов есть по параметру.
2. Далее три класса X имеют по два родительских класса.
3. В финале класс Z наследуется от трёх классов X.

У каждого класса добавили текстовый вывод при вызове методу `__init__`. Также обратите внимание на наличие функции `super`, которая вызывает инициализацию родительского класса.

На схеме наследование будет выглядеть так:



У каждого класса есть метод `mro`, который вычисляет порядок наследования. Он отвечает за инициализацию каждого класса один раз в порядке слева направо и по старшинству, т.е. родитель не может быть инициализирован раньше дочернего класса.. Подробнее про то как работает “монотонная линеаризация суперкласса” можно прочитать по [ссылке](#).

Разберём результат работы `mro` с нашим классом Z.

- В первую очередь отрабатывает инициализация самого класса.
- Далее начинаем двигаться слева направо по списку родительских классов: X1, X2
- Следующим будет класс B. Почему он, а не X3? Класс B является родительским только для класса X2. Так мы не нарушаем порядок слева направо и старшинство.
- Следующим инициализируется X3, последний из родительских классов у Z.

- Далее идёт инициализация класса А. Он родитель для Х1 и Х3. Следовательно его инициализация была невозможна раньше дочерних классов.
- Классы С и D инициализируются последними, они правее А, В и С в списке родительских классов у “иксов”.
- Класс object всегда инициализируется в последнюю очередь.

Поиск аргументов и методов в экземпляре класса Z будет происходить в порядке, представленном методом mro.

Добавим несколько строк кода и посмотрим на результат:

```
z = Z()
print(f'{z.data_b = }')
print(f'{z.data_a = }')    # AttributeError: 'Z' object has no
attribute 'data_a'
```

Вызов метода `__init__` остановился на классе В. Мы не дописали ему вызов `super`, считая что он и так не имеет наследников. В результате аргумент `data_a` не был создан в экземпляре класса z. Попробуем описать классу А дополнительный метод.

```
class A:
    def __init__(self):
        print('Init class A')
        self.data_a = 'A'

    def say(self):
        print('Текст')
        print(self.data_b)
    ...

z = Z()
z.say()
```

Вызов метода `say` из класса А отработал без ошибок. Мы нашли его двигаясь по цепочке линейаризации. При этом метод даже смог обратиться к свойству другого класса. Связано это с тем, что мы работаем из экземпляра класса Z и он собрал в себя аргументы и методы наследуемых классов.



Важно! Не стоит из родительских классов обращаться к аргументам и методам дочерних классов или классов того же уровня наследования.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
class A:
    name = 'A'

    def call(self):
        print(f'I am {self.name}')

class B:
    name = 'B'

    def call(self):
        print(f'I am {self.name}')

class C:
    name = 'C'

    def call(self):
        print(f'I am {self.name}')

class D(C, A):
    pass

class E(D, B):
    pass

e = E()
e.call()
```

4. Полиморфизм

С полиморфизмом мы уже сталкивались раньше. Например сложение чисел возвращает их сумму, а сложение строк возвращает новую строку состоящую из двух исходных. Одинаковые действия приводят к разному, но ожидаемому результату.

Полиморфизм был ранее в этой лекции:

```
class Person:
    ...

    def change_health(self, other, quantity):
        self.health += quantity
        other.health -= quantity
    ...

class Hero(Person):
    def __init__(self, power, *args, **kwargs):
        self.power = power
        super().__init__(*args, **kwargs)

    def change_health(self, other, quantity):
        self.health += quantity * 2
        other.health -= quantity * 2
    ...

p1 = Hero('archery', 'Сильвана', 'Эльф', 120)
p2 = Person('Маг', 'Тролль')
print(f'{p1.health = }, {p2.health = }')
p1.change_health(p2, 10)
print(f'{p1.health = }, {p2.health = }')
p2.change_health(p1, 10)
print(f'{p1.health = }, {p2.health = }')
```

Один и тот же метод `change_health` по разному меняет параметр `health` у обычного персонажа и у героя.

Рассмотрим ещё один вариант полиморфизма.

```
path_1 = '/home/user'
path_2 = '/my_project/workdir'
result = path_1 / path_2    # TypeError: unsupported operand
                             type(s) for /: 'str' and 'str'
```

Python не поддерживает деление строк. Но мы уже сталкивались с тем как класс `Path` из модуля `pathlib` создавал новый путь используя символ деления. Реализовать подобный полиморфизм можно например так.

```
class DivStr(str):
    def __init__(self, obj):
        self.obj = str(obj)
```

```

def __truediv__(self, other):
    first = self.obj.endswith('/')
    start = self.obj

    if isinstance(other, str):
        second = other.startswith('/')
        finish = other
    elif isinstance(other, DivStr):
        second = other.obj.startswith('/')
        finish = other.obj
    else:
        second = str(other).startswith('/')
        finish = str(other)

    if first and second:
        return DivStr(start[:-1] + finish)
    if (first and not second) or (not first and second):
        return DivStr(start + finish)
    if not first and not second:
        return DivStr(start + '/' + finish)

path_1 = DivStr('/home/user/')
path_2 = DivStr('/my_project/workdir')
result = path_1 / path_2
print(f'{result} = ', {type(result)})
print(f'{result} / "text" = ')
print(f'{result} / 42 = ')
print(f'{result} * 3 = ')

```

Создаём класс DivStr как наследник класса str. При инициализации определяем аргумент obj, который является обычной строкой. Вся магия деления строк будет спрятана в магическом методе __truediv__ который срабатывает при делении экземпляра класса DivStr на другой такой же экземпляр или на обычную строку str:

1. Первым делом определяем заканчивается ли первая часть на символ /, а саму строку сохраняем в переменной start.
2. Далее проверяем начинается ли на символ / и сохраняем вторую половинку в finish
 - a. Если вторая половинка строка, работаем с объектом other как со строкой.
 - b. Если вторая половинка экземпляр класса DivStr, работаем со свойством obj.
 - c. Если оба варианта неверны, пробуем привести объект к строковому виду.

3. В зависимости от того заканчивается или нет первая часть на символ / и начинается или нет вторая часть на символ / соединяем объекты и возвращаем новый экземпляр DivStr.

Используя полиморфизм и переопределение метода мы смогли “разделить” два экземпляра DivStr. Также мы можем “делить” на строки и даже на числа, ведь числа имеют строковое представление. А так как мы наследовались от str, объект поддерживает и привычные операции со строками.

Подробнее про переопределение магических методов поговорим на следующем занятии. Эта тема заслуживает отдельного внимания.

Вывод

На этой лекции мы:

1. Разобрались с объектно-ориентированным программированием в Python.
2. Изучили особенности инкапсуляции в языке
3. Узнали о наследовании и механизме разрешения множественного наследования.
4. Разобрались с полиморфизмом объектов.

Краткий анонс следующей лекции

1. Разберёмся с созданием и удалением классов
2. Узнаем о документировании классов
3. Изучим способы представления экземпляров
4. Узнаем о возможностях переопределения математических операций
5. Разберёмся со сравнением экземпляров
6. Узнаем об обработке атрибутов

Домашнее задание

Возьмите 1-3 задачи из прошлых занятий и попробуйте перенести переменные и функции в класс. Убедитесь, что созданный вами класс позволяет верно решать поставленные задачи.