Программирование на Python

Базовый курс. Часть 3. Объектно-ориентированное программирование

Андрей Скороходов

udv|group 2023

Python. Объектно-ориентированное программирование

ООП – методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования (Wikipedia).

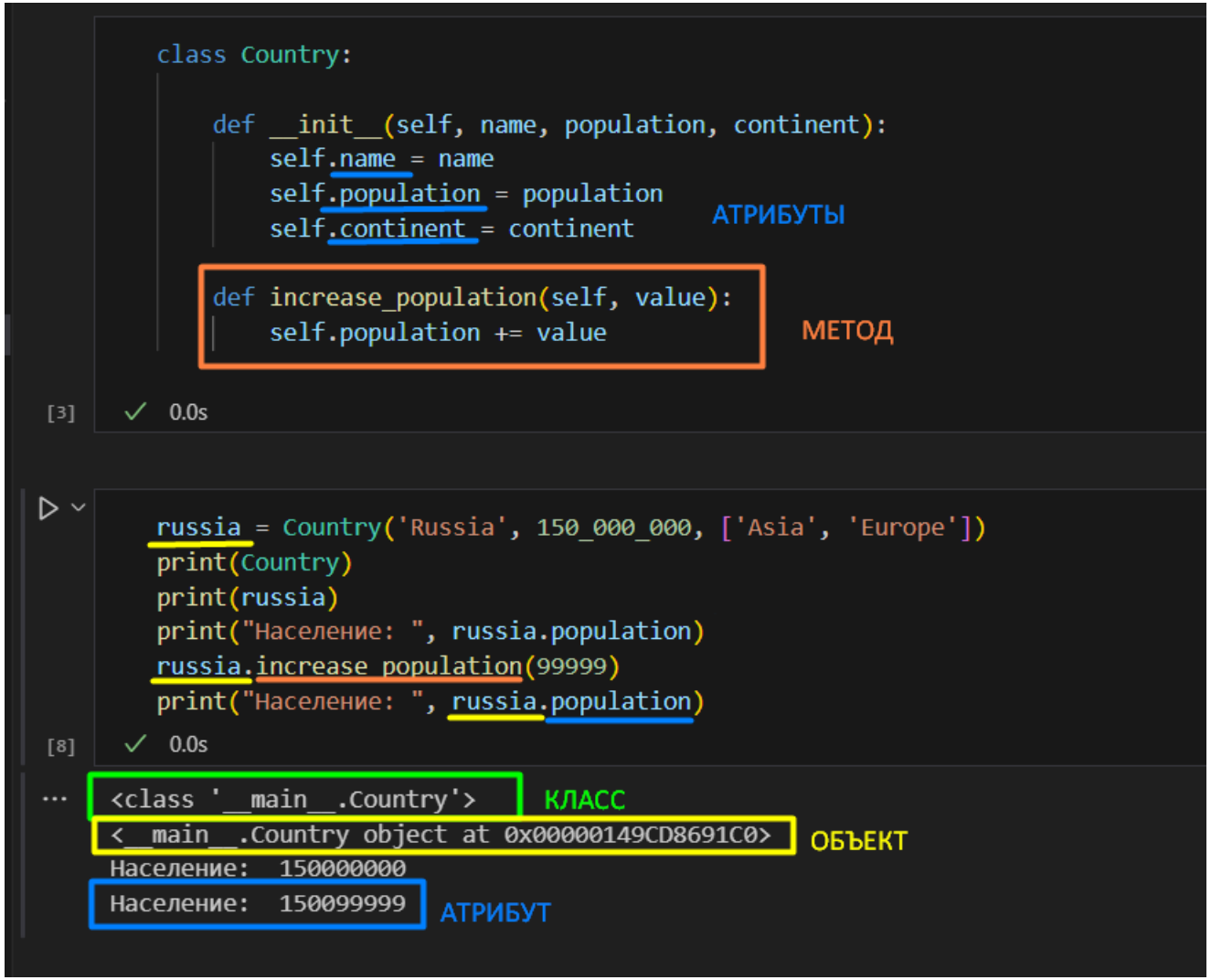
Основные принципы ООП:

- **Полиморфизм** – разное поведение одного и того же метода в разных классах
- **Инкапсуляция** – ограничение доступа к методам и переменным объекта
- **Наследование** – наследование дочерним классом атрибутов родительского класса
- **Абстракция** – скрытие внутренних реализаций процесса или метода от пользователя

Основные понятия:

- **Класс** – модель для создания объектов определённого типа, описывающая их структуру.
- **Объект** – экземпляр класса.
- **Атрибут** – свойства (переменные), принадлежащие классу или объекту класса
- **Метод** – функции, принадлежащие классу или объекту класса

Python. Пример объявления и вызова объекта класса



Определение класса:

class *NewClass*:

def *method1*(self, x):

self.attribute = x

def *method2*(self, …):

<инструкции метода>

def *method3*(self, …):

<инструкции метода>

Создание объекта класса:

class_object = NewClass()

Вызов метода класса:

class_object.method1(<аргументы>)

Доступ к атрибуту класса:

class_object.attribute

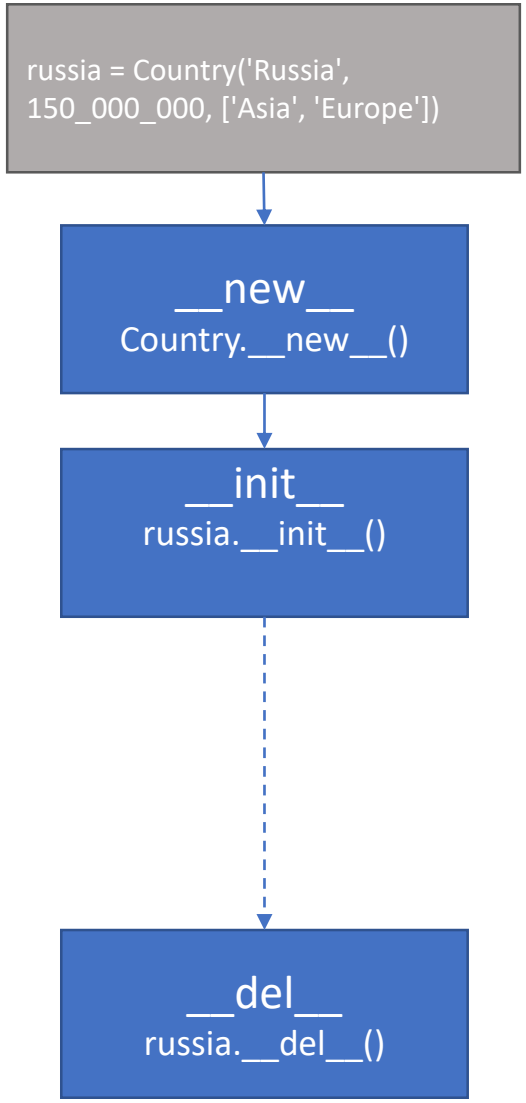
Python. Встроенные (магические) методы класса

Метод	Когда вызывается
__new__(cls, [...)	При создании объекта класса
__init__(self, [...)	При инициализации объекта
__del__(self)	Перед удалением объекта
__eq__(self, other)	При сравнении через оператор == *(Операторы !=, <, > <=, >= : __ne__, __lt__, __gt__, __le__, __ge__)*
__call__(self, [args...])	При вызове объекта
__str__(self)	При вызове через функцию **str(***<объект>***)**
__abs__(self)	При вызове через функцию **abs(***<объект>***)**
__len__(self)	При вызове через функцию **len(***<объект>***)**
__setattr__(self, name, value)	При изменении атрибута объекта *(При обращении к атрибуту или его удалении: __getattr__, __delattr__)*
__add__(self, other)	При сложении через оператор + *(Операторы -, *, / : __sub__, __mul__, __div__)*
__hash__(self)	При вызове через функцию hash()
__next__(self)	При вызове через функцию hash()

Это далеко не все методы!

https://habr.com/ru/articles/186608/

Python. Жизненный цикл объекта класса



После объявления объекта класса:

- вызывается магический метод __new__, который возвращает ссылку на объект нового класса
- вызывается магический метод __init__ для объекта класса, где выполняются инициализирующие процедуры
- происходит работа с объектом
- при отсутствии ссылок на объект или при вызове инструкции del вызывается магический метод __del__, после чего объект удаляется.

Python. Атрибуты объекта и класса

Для получения атрибутов объекта или класса можно воспользоваться методом **__dict__.**

```
print("russia:", russia.__dict__)
Country.__dict__
```

✓ 0.0s

```
russia: {'name': 'Russia', 'population': 150099999, 'continent': ['Asia', 'Europe']}
```

Атрибуты объекта

```
mappingproxy({'__module__': '__main__',
              'NAME': 'class_Country',
              '__init__': <function __main__.Country.__init__(self, name, population, continent)>,
              'increase_population': <function __main__.Country.increase_population(self, value)>,
              '__dict__': <attribute '__dict__' of 'Country' objects>,
              '__weakref__': <attribute '__weakref__' of 'Country' objects>,
              '__doc__': None})
```

Атрибуты класса

Получение значений атрибута:

При обращении к несуществующему атрибуту объекта этот объект будет искаться в атрибутах класса.

При отсутствии атрибута в пространстве имен объекта или класса будет вызвано исключение.

Для получения значения атрибута объекта / класса можно использовать функцию **getattr().**

Изменение значений атрибута:

При обращении к несуществующему атрибуту объекта будет создан новый атрибут объекта, даже если имя атрибута есть в пространстве имен класса.

Для изменения значения атрибута объекта / класса можно использовать функцию **setattr().**

```
print(Country.NAME)
print(russia.NAME)
```

✓ 0.0s

```
class_Country
class_Country
```

```
russia.NAME = 'Russia'
russia.__dict__
```

✓ 0.0s

```
{'name': 'Russia',
 'population': 150099999,
 'continent': ['Asia', 'Europe'],
 'NAME': 'Russia'}
```

```
print(Country.NAME)
print(russia.NAME)
```

✓ 0.0s

```
class_Country
Russia
```

Python. Функции для работы с атрибутами объекта/класса

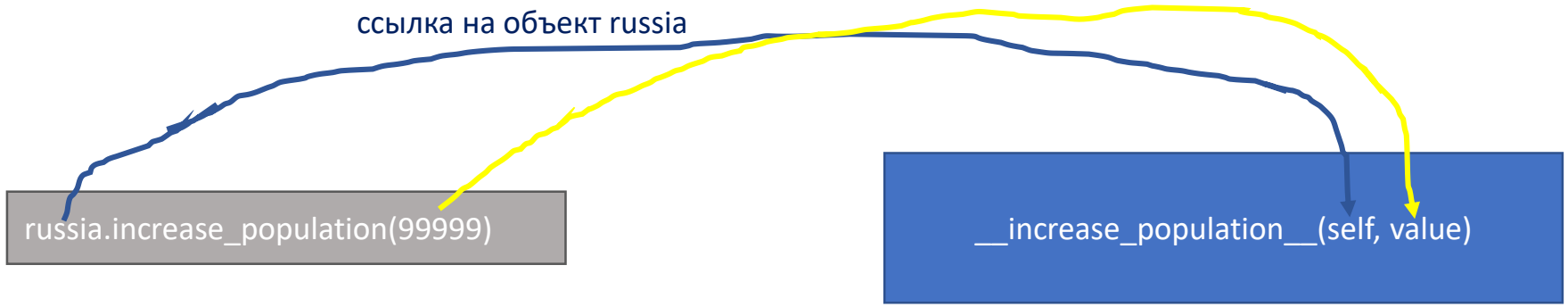
Метод	Описание
setattr(*<класс/объект>, <имя атрибута>, <значение атрибута>***)**	Присваивание значения атрибуту класса/объекта
getattr(*<класс/объект>, <имя атрибута>, <значение при отсутствии>***)**	Получение значения атрибута класса/объекта
delattr(*<класс/объект>, <имя атрибута>***)**	Удаление атрибута класса/объекта
hasattr(*<класс/объект>, <имя атрибута>***)**	Наличие указанного атрибута у класса/объекта. Возвращает True/False. В случае отсутствия атрибута у объекта, но присутствия его в класса все равно возвратит True.

Python. Вызов методов класса, параметр self

При определении методов класса первым параметром всегда идет параметр **self**, который **указывает на объект класса**.

При вызове метода класса интерпретатор автоматически передает в метод в качестве первого аргумента ссылку на объект, который вызвал этот метод.

Передача ссылки на объект, вызвавший метод, нужны для работы с объектом внутри метода.



Python. Статические методы и методы класса

	Статический метод	Метод класса
Ссылка на объект/класс	Не требуется	Нужна ссылка на класс
Доступ к атрибутам объекта	НЕТ	НЕТ
Доступ к атрибутам класса	НЕТ	ДА
Доступ внутри класса	через **self.**	через **self.**
Доступ извне	через **<объект класса>.** через **<имя класса>.**	через **<объект класса>.** через **<имя класса>.**
Декоратор	@staticmethod	@classmethod

В метод класса вместо обычной ссылки на объект класса (self)

передается ссылка на сам класс: **cls**

```python
class Country:
    favorite_country = 'Russia'

    def __init__(self, name, population, continent):
        self.name = name
        self.population = population
        self.continent = continent

    def increase_population(self, value):
        self.population += value

    @classmethod
    def is_favorite(cls, country_name):
        return country_name in cls.favorite_country

    @staticmethod
    def calc_millions(population):
        return population / 1000000

russia = Country('Russia', 150_000_000, ['Asia', 'Europe'])
```

✓ 0.0s

```python
print(russia.is_favorite('Russia'))
print(russia.calc_millions(5900000))
```

Python. ООП. Задание 1

1. Создайте класс Employee, который будет описывать сотрудника компании.
2. В качестве начальных атрибутов класса задайте позиционные параметры «name», «surname», «position», «age», «currency» и именованный параметр «salary», по умолчанию равный 50000.
3. Определите метод change_currency(), который меняет валюту зарплаты, по аналогии с заданием из части «Python. Функции».
4. Определите атрибут класса AGE_LIMITS и метод класса verify_age(), который будет возвращать True или False в зависимости от того, попадает ли возраст сотрудника в указанный диапазон.
5. Добавьте в инициализатор класса проверку параметра age и в случае, если возраст сотрудника выходит за пределы диапазона, выведите в консоль предупреждение о том, что возраст сотрудника не удовлетворяет корпоративным стандартам.
6. Создайте объект employee1 класса Employee с произвольными параметрами
7. Выведите атрибуты объекта в консоль в формате «атрибут»: «значение».

Python. ООП. Задание 2

1. В ранее созданном классе Employee определите статический метод convert_currency(), который принимает в качестве параметров денежную сумму, текущую валюту и требуемую валюту и возвращает значение денежной суммы в требуемой валюте.
2. Для пересчета валюты определите внутри метода convert_currency() словарь exchange_rate, где ключом будет являться валюта, а значением – коэффициент перевода. В качестве основной валюты (которая будет иметь значение коэффициента перевода = 1) выберите «rub» и определите еще несколько других валют, например {‘rub’: 1, ‘usd’: 90, ‘eur’:100}.
3. Замените код в методе change_currency(), оставив там только проверку, что валюта изменилась. И при изменении валюты получайте новое значение зарплаты с использованием функции convert_currency(). В случае отсутствия курса валюты в словаре exchange_rate, выводите в консоль предупреждение о невозможности изменения валюты и не меняйте атрибуты объекта. В противном случае, измените зарплату и валюту объекта.
4. Создайте еще несколько сотрудников (объектов класса Employee) и посчитайте их суммарную зарплату и средний возраст

Python. ООП. Режимы доступа к атрибутам

В Python предусмотрено 3 режима доступа к объектам:

- **private (публичный)** – доступ извне разрешен. Имена публичных атрибутов не содержат «_» в начале: **attribute**
- **protected (защищенный)** – не ограничивает доступ, но предупреждает разработчика о том, что атрибут является защищенным и его не следует менять. Имена защищенных атрибутов начинаются с «_»: **_attribute**
- **private (локальный)** – доступ есть только внутри класса. Имена локальных атрибутов начинаются с «__»: **__attribute**

Режимы доступа обеспечивают один из принципов ООП – **инкапсуляцию**.

Правила наименования атрибутов справедливы как для переменных, так и для функций внутри класса. Для изменения режима доступа к функциям можно использовать декораторы **@private** или **@protected** (предварительно нужно их импортировать: from accessify import protected, private).

На самом деле доступ к локальным атрибутам класса извне есть, но для этого нужно обращаться не по имени атрибута, а по кодовому имени «_<имя класса><имя атрибута>».

```python
class Country:
    favorite_country = 'Russia'

    def __init__(self, name, population, continent):
        self.name = name
        self.population = population
        self.continent = continent
        self.__private_attribute = 'пример локального атрибута'
        self._protected_attribute = 'пример защищенного атрибута'
```

```
russia._protected_attribute = russia._protected_attribute + ' да пофиг'
✓ 0.0s
```

```
russia.__dict__
✓ 0.0s
{'name': 'Russia',
 'population': 150000000,
 'continent': ['Asia', 'Europe'],
 '_Country__private_attribute': 'пример локального атрибута',
 '_protected_attribute': 'пример защищенного атрибута да пофиг'}
```

Python. ООП. Доступ к атрибутам. Сеттеры и геттеры

Для изменения и получения значений локальных атрибутов рекомендуется использовать специальные методы – сеттеры (изменение атрибута) и геттеры (получение значения). Можно их создавать как отдельные методы, например, для переменной x: **set_x()** и **get_x()**, но тогда придется обращаться к различным методам для получения и изменения атрибута. Также можно использовать специальный декоратор **@property**, который позволяет обращаться к сеттеру и геттеру по названию переменной. Для этого геттер называется по имени локальной переменной, а к геттеру добавляется декоратор @property. Сеттер также называется по имени переменной и к нему добавляется декоратор **@<имя геттера>.setter**.

```python
class Country:
    favorite_country = 'Russia'

    def __init__(self, name, population, continent):
        self.name = name
        self.population = population
        self.continent = continent
        self.__private_attribute = 'пример локального атрибута'
        self._protected_attribute = 'пример защищенного атрибута'

    @property
    def private_attribute(self):
        return self.__private_attribute

    @private_attribute.setter
    def private_attribute(self, value):
        self.__private_attribute = value
```

```python
print(russia.private_attribute)
russia.private_attribute = "его можно менять"
russia.private_attribute
```

✓ 0.0s

```
пример локального атрибута

'его можно менять'
```

Python. ООП. Задание 3

1. В ранее созданном классе Employee сделайте атрибуты name, surname, position, age, salary и currency локальными (измените имена). Добавьте для этих атрибутов геттеры с использованием декоратора @property, а для атрибутов position, age, salary добавьте сеттеры.
2. В геттере для age добавьте проверку типа передаваемого значения (int) и то, что он больше 0 и меньше 100. В случае, если это не int или возраст не корректный, выдавайте предупреждение о некорректном возрасте и не меняйте локальную переменную. Также добавьте проверку его нахождения внутри диапазона AGE_LIMITS, в случае выхода за его пределы формируйте сообщение, аналогичное сообщению из задания 1, но изменение возраста выполните.
3. В геттере для salary также добавьте проверку значения на корректность (по вашим критериям) и в случае, если передаваемое значение не проходит по критериям, выводите в консоль предупреждение и не меняйте локальную переменную.
4. Сделайте метод verify_age() защищенным (измените имя).

Python. ООП. Магические методы. __setattr__

__setattr__(self, name, value)

Данный метод вызывается при каждом изменении атрибута класса. Это можно использовать, например, для проверки передаваемых значений или если нужно запретить создание атрибутов с определенным именем. Если переопределять этот метод, то он должен возвращать **object.__setattr__(self, key, value)**, иначе атрибут не будет изменен.

```python
class Country:
    favorite_country = 'Russia'

    def __setattr__(self, key, value):
        if key=='id':
            if value != 'id':
                return

        return object.__setattr__(self, key, value)

    def __init__(self, name, population, continent):
        self.id = 'id'
        self.name = name
```

```
russia.id = 'idd'
russia.id
✓ 0.0s
'id'
```

Python. ООП. Задание 4

1. Перенесите проверку типов и границ передаваемых значений для атрибутов age и salary в магический метод __setattr__(). Для age проверку нахождения возраста в диапазоне AGE_LIMITS оставьте в сеттере.

Python. ООП. Магические методы. __call__, __str__, __eq__

__call__(self, [args...])

Данный метод вызывается при обращению к объекту как к методу (**()**). Классы, которые могут себя вести как функции называются функторы. Это может применяться, например для использования класса в качестве декоратора.

__str__(self)

Данный метод вызывается при обращению к объекту через функцию str().

__eq__(self, other)

Данный метод вызывается при сравнении объекта с другим объектом (оператор ==). При определении метода __eq__ обычно также будет возможно применение к объектам оператора !=.

```python
class Country:
    favorite_country = 'Russia'

    def __eq__(self, other):
        return self.name == other.name

    def __call__(self, *args, **kwargs):
        print(f'{self.name} нуждается в вас!')

    def __str__(self):
        return f"Страна {self.name} с населением {self.population}"
```

```
russia == Country('Russia', 150, ['Asiaaa', 'Europe'])
✓ 0.0s

True
```

```
print(russia)
russia()
✓ 0.0s

Страна Russia с населением 150000000
Russia нуждается в вас!
```

Python. ООП. Задание 5

1. Добавьте магический метод __call__(). Пусть при обращении к объекту как к функции в консоль выводится сообщение «Сотрудник <имя сотрудника> уже бежит к вам».
2. Добавьте магический метод __str__(). Определите формат возвращаемой информации об объекте.
3. Добавьте магический метод __eq__(). Пусть сотрудники будут равны, если у них одинаковые имена и фамилии
4. Добавьте магический метод __le__(). Этот метод позволяет выполнять сравнение объектов оператором <=. Придумайте, по какому атрибуту вы будете сравнивать сотрудников и реализуйте сравнение.

Python. ООП. Наследование

Наследование – механизм создания класса на основе другого существующего класса.

Основные понятия:

- **Базовый (родительский) класс** – класс, от которого производится наследование
- **Подкласс (дочерний класс)** – класс, который наследуется

В подклассе доступны все атрибуты и методы родительского класса.

В подклассе возможно переопределить атрибуты и методы родительского класса.

Все объекты в Python в конечном итоге наследуются от базового класса **object**.

Для определения того, что какой-либо класс является подклассом другого класса, можно использовать функцию **issubclass(**<подкласс>, <базовый класс>**)**.

Для определения того, что какой-либо класс/объект наследуется от другого класса, можно использовать функцию **isinstance(**<подкласс/объект>, <базовый класс>**)**.

Если подкласс переопределяет атрибуты базового класса – **переопределение (overriding) класса.**

Если подкласс дополняет атрибуты базового класса – **расширенный (extended) класс.**

```
print(isinstance(russia, object))
print(isinstance(russia, Country))
print(issubclass(Country, object))
print(issubclass(russia, object))
```

0.0s

```
True
True
True

---------------------------------------------------------------
TypeError                                 Traceback
C:\Users\ASKORO~1\AppData\Local\Temp/ipykernel_605
      2 print(isinstance(russia, Country))
      3 print(issubclass(Country, object))
----> 4 print(issubclass(russia, object))

TypeError: issubclass() arg 1 must be a class
```

Python. ООП. Задание 6

1. Создайте на основе встроенного класса dict новый класс superdict, в котором определите метод get_. Этот метод будет дублировать функции метода get базового класса, но дополнительно будет выводить в консоль предупреждение, что ключ отсутствует.

Python. ООП. Задание 7

1. Создайте классы Engineer и Manager, унаследованные от класса Employee.
2. В классе Engineer переопределите атрибут базового класса AGE_LIMITS на [20, 65].
3. В классе Manager переопределите значение по умолчанию параметра salary.
4. В классе Engineer определите метод change_salary(), добавив проверку на процент увеличения зарплаты. Если зарплата увеличилась более чем на 50%, нужно выводить в консоль предупреждение о подозрении на коррупционные схемы.
5. Добавьте в класс Engineer атрибут manager – ссылка на объект класса Manager.
6. Добавьте в класс Manager атрибут engineers – список объектов класса Engineer.
7. Добавьте в класс Manager метод add_engineer(), который будет принимать на вход ссылку на объект Engineer и добавлять ее в список engineers, если данного объекта еще нет в списке. Кроме того, при добавлении ссылки на объект Engineer необходимо в этом объекте присвоить значению manager ссылку на объект класса Manager, в чей список попал инженер.
8. Создайте 5 объектов класса Engineer и 2 объекта класса Manager и распределите инженеров по менеджерам. Выведите в консоль атрибут engineers объектов класса Manager.

Python. ООП. Делегирование и функция super()

Если в подклассе переопределен метод базового класса, то при вызове метода будет вызван метод подкласса. Это касается в т.ч. магических методов.

При определении в подклассе метода __init__ метод инициализатор базового класса не будет вызван без явной инструкции, поэтому, если нам нужно вызвать инициализатор базового класса, в инициализаторе подкласса необходимо использовать метод **super()**, который ссылается на базовый класс: **super().__init__(**<аргументы>**). Причем его нужно вызывать в самом начале инициализации!** Вызов методов базового класса через функцию super() называет **делегированием**.

```
class A:

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

class B(A):

    def __init__(self, x, y, z):
        #super().__init__(x, y)

        self.z = z
```

```
b = B(1, 2, 3)
b.__dict__
```

✓ 0.0s

```
{'z': 3}
```

```
class A:

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

class B(A):

    def __init__(self, x, y, z):
        super().__init__(x, y)

        self.z = z
```

```
b = B(1, 2, 3)
b.__dict__
```

✓ 0.0s

```
{'x': 1, 'y': 2, 'z': 3}
```

Python. ООП. Полиморфизм

Полиморфизм – возможность работы единым образом с различными объектами, через единый интерфейс.

Для создания единого интерфейса необходимо, чтобы методы интерфейса были определены во всех классах.

Для реализации этого требования можно воспользоваться абстрактными методами, которые определить в базовом классе.

Абстрактный метод – метод, который не содержит реализации. Для реализации абстрактного метода в Python есть декоратор **abstractmethod** модуля **abc** (from abc import ABC, abstractmethod).

```
from abc import ABC, abstractmethod

class A(ABC):

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

    @abstractmethod
    def do_something(self):
        pass

class B(A):

    def __init__(self, x, y, z):
        super().__init__(x, y)

        self.z = z
```

```
b = B(1, 2, 3)
0.0s

TypeError                                 Trace
C:\Users\ASKORO~1\AppData\Local\Temp/ipykernel_
----> 1 b = B(1, 2, 3)

TypeError: Can't instantiate abstract class B
```

```
class A(ABC):

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

    @abstractmethod
    def do_something(self):
        pass

class B(A):

    def __init__(self, x, y, z):
        super().__init__(x, y)

        self.z = z
    def do_something(self):
        print('okay')
```

```
b = B(1, 2, 3)
b.do_something()
✓ 0.0s

okay
```

Python. ООП. Задание 8

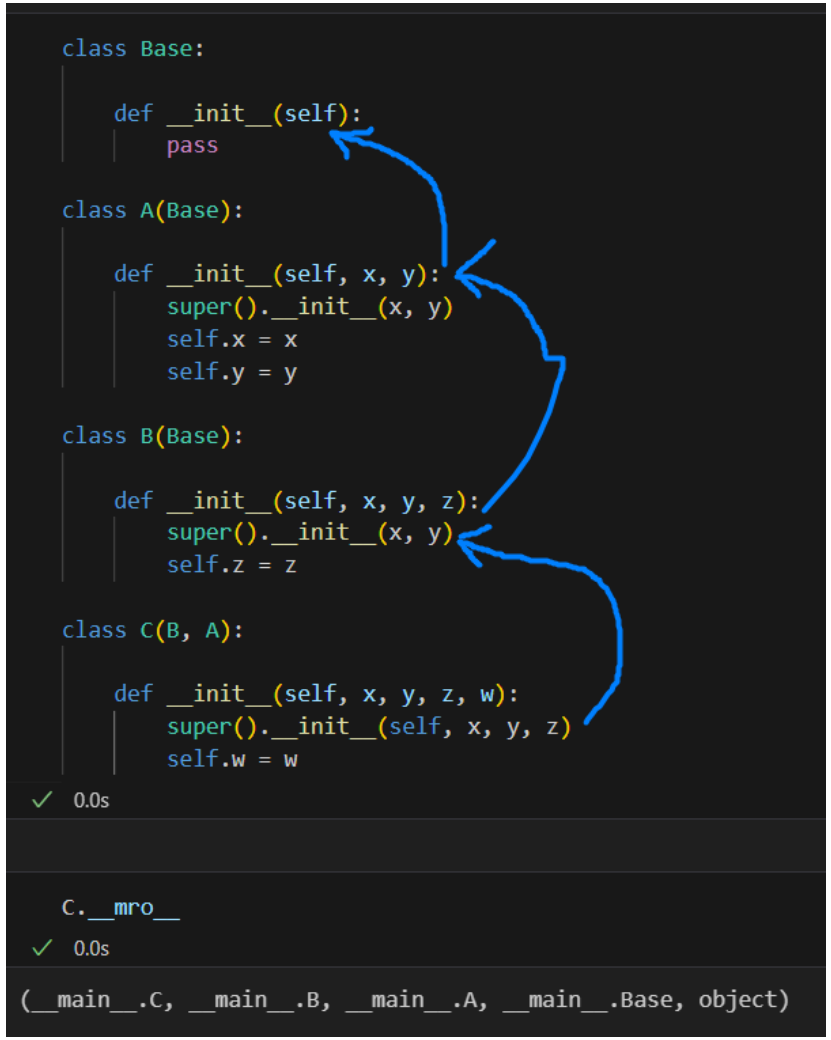
1. В классе Employee добавьте абстрактный метод give_premium().
2. Определите метод give_premium() в классах Engineer и Manager. Метод должен выводить в консоль сообщение типа <position> <name> <surname> получил премию в размере <размер премии>. Размер премии для объекта класса Engineer должен равняться 10% от его зарплаты, а для объекта класса Manager – 15% от суммы всех зарплат инженеров, находящихся в его подчинении.
3. Пересоздайте объекты классов Engineer и Manager и в цикле дайте каждому из них премию.

Python. ООП. Множественное наследование

В Python существует возможность наследования от нескольких родительских классов – такой механизм называется множественным наследованием.

Для разрешения конфликтов инициализации объектов реализован механизм MRO – Method Resolution Order. Для получения порядка наследования подкласса нужно выполнить команду:

<имя класса>.__mro__



Python. ООП. Задание 9

1. Объявите класс Person, от которого будет наследоваться класс Employee.
2. Переместите в Person все атрибуты и методы, рассматривающие сотрудника как человека: возраст, имя и фамилия.
3. Пересоздайте объекты классов Engineer и Manager и убедитесь в корректной работе всех функций.