Программирование на Python.

Тема №2. Атрибуты, свойства.
Модификаторы доступа

Механизм создания экземпляров

Рассмотрим простой пример:

```
class Example:
    mode = 1
    color = 'green'

a = Example() #a.mode = 1, a.color = 'green'
b = Example() #b.mode = 1, b.color = 'green'
```

Рассмотрим детальнее, что происходит при создании экземпляров

Пространство имен

При создании, объекты **a** и **b** создают собственные пространства имен – пространства имен **ЭКЗЕМПЛЯРОВ КЛАССА**.

Так же – они не содержат собственных атрибутов. Атрибуты **mode** и **color** принадлежат самому классу Example и находятся в нем. А объекты **a** и **b** содержат ссылки на эти атрибуты класса. В этом можно легко убедиться используя команду **id()**:

```
print(id(a.mode))         #140730201596712
print(id(Example.mode)) #140730201596712
```

Атрибуты класса

Таким образом – мы приходим к пониманию того, что такое **Атрибуты класса** (иногда их так же называют **свойствами**)**.**

Это общие для всех экземпляров атрибуты, то есть каждый экземпляр класса может получить доступ к этим атрибутам.

Атрибуты экземпляров

Попробуйте присвоить значение новое значение атрибуту **mode**, и проверьте еще раз id(a.mode):

```
print(id(a.mode))          #140730201596712
a.mode = 10
print(id(a.mode))          #140730201597000
```

Мы видим другой id, значит a.mode содержит ссылку на другую область памяти, нежели Example.mode

Атрибуты экземпляров

Таким образом, мы создали переменную с именем **mode** уже в пространстве имен экземпляра класса. Так же это можно оценить, заглядывая в **__dict__** экземпляра до и после присвоения значения переменной **mode**

Такие атрибуты называются **атрибутами экземпляров**

Добавление новых атрибутов

Вы можете добавлять новые атрибуты как классам, так и экземплярам:

```
Example.new_attr = 'value'                  #таким образом
setattr(Example,'new_attr','value') #или таким
a.new_value = 10                            #для экземпляров
```

Если вы попытаетесь присвоить значение атрибуту с именем, которого еще не было в пространстве имен – Python добавит новый атрибут с эти значением.

Доступ к атрибутам

Для доступа к атрибутам – достаточно обратиться к их имени:

```
print(Example.new_attr) #value
```

Важно! Если обратиться к несуществующему атрибуту, вы получите ошибку. Этого можно избежать, воспользовавшись специальной функцией **getattr** (возвращает третий атрибут функции (None для примера), если не найден атрибут с запрашиваемым именем:

```
print(getattr(Example,'attr',None))
```

Удаление атрибутов

Для удаления атрибутов можно воспользоваться командой **del** или командой **delattr(ClassName, ‘attr_name’):**

```
del Example.new_attr
delattr(Example,'new_attr')
```

При попытке удаления атрибута, имени которого нет в пространстве имен – вы получите Attribute error

Тонкости

Тип данных Example.mode – **int (Целое число).** Это неизменяемый тип данных. При попытке изменения значения переменной с таким типом данных, Python просто создает в памяти новый объект с новым значением, и заменяет старый адрес из переменной на новый в этой переменной, например:

```
demo_var = 10
print(id(demo_var)) #40730218636360
demo_var = 20
print(id(demo_var)) #140730218636680
```

Тонкости

Создадим атрибут класса – список, и попробуем добавить в него новый элемент, обращаясь к атрибуту класса от имени экземпляра:

```
class Example:
    mode = []
    color = 'green'

a = Example()
a.mode.append('new_value')
print(Example.mode) #['new_value']
```

Новое значение попало внутрь списка на уровне атрибута класса. Эта особенность вытекает из того, как python работает с изменяемыми типами данных **(list).**

Такие нюансы нужно знать, чтобы избежать ошибок при работе с данными.

Инициализация экземпляров

При создании новых экземпляров **python** вызывает специальный магический метод **__init__** (в следующей теме мы обсудим их подробнее), называемый **инициализатором объекта класса (или конструктором экземпляра класса).** По своей сути, это функция, которая вызывается единожды в момент создания экземпляра. Давайте рассмотрим на примере, для чего он нужен.

__init__

```
class Example:
    mode = []
    color = 'green'
    def __init__(self):
        print('вызов __init__')
        self.value = 100

a = Example()     #вызов __init__
print(a.value)    #100
```

В момент создания экземпляра с именем **a** в терминале появляется сообщение «вызов __init__», и создается атрибут экземпляра **a.value** со значением **100.** Ключевое слово **self** используется для указания ссылки на создаваемый экземпляр класса

__init__ с доп. аргументами

При создании новых экземпляров, часто возникает необходимость передавать значения в __init__, это работает как с обычными функциями в **python**:

```
class Example:
    mode = []
    color = 'green'
    def __init__(self,value, name):
        print('вызов __init__')
        self.value = value
        self.name = name

a = Example( value: 100, name: 'Новый')
print(a.value)    #100
```

Дополнительные аргументы перечисляются через запятую в конструкторе класса

Упражнение

- Реализуйте **атрибут класса**: счетчик количества созданных экземпляров для класса Example.
- Реализуйте **метод** для отображения информации о текущем «порядковом» номере экземпляра
- Реализуйте **метод** для отображения информации об общем количестве экземпляров.

Финализатор

Метод **__del__** автоматически вызывается перед уничтожением экземпляра класса:

```
class Example:
    mode = []
    color = 'green'
    def __init__(self,value, name):
        print('вызов __init__')
        self.value = value
        self.name = name

    def __del__(self):
        print("Удаление экземпляра: " + str(self))

a = Example( value: 100, name: 'Новый')
del(a)  #Удаление экземпляра: <__main__.Example object at 0x0000028B2CC28110>
```

Инкапсуляция. Геттеры и сеттеры

Инкапсуляция

Инкапсуляция – механизм ограничения доступа к данным и метода класса извне. По сути – это изолирование данных.

Рассмотрите пример:

```
class Person:
    def __init__(self, name, money):
        self.name = name
        self.money = money
new = Person( name: 'Иван Иванович',  money: 100_000)
print(new.money)          #100000
new.money = 0
print(new.money)          #0
```

Инкапсуляция

В данном примере мы можем напрямую получить доступ к атрибуту **new.money** и изменить его значение в том числе, на недопустимое, или поменять тип данных.

```
class Person:
    def __init__(self, name, money):
        self.name = name
        self.money = money
new = Person( name: 'Иван Иванович',  money: 100_000)
print(new.money)          #100000
new.money = 0
print(new.money)          #0
```

Инкапсуляция

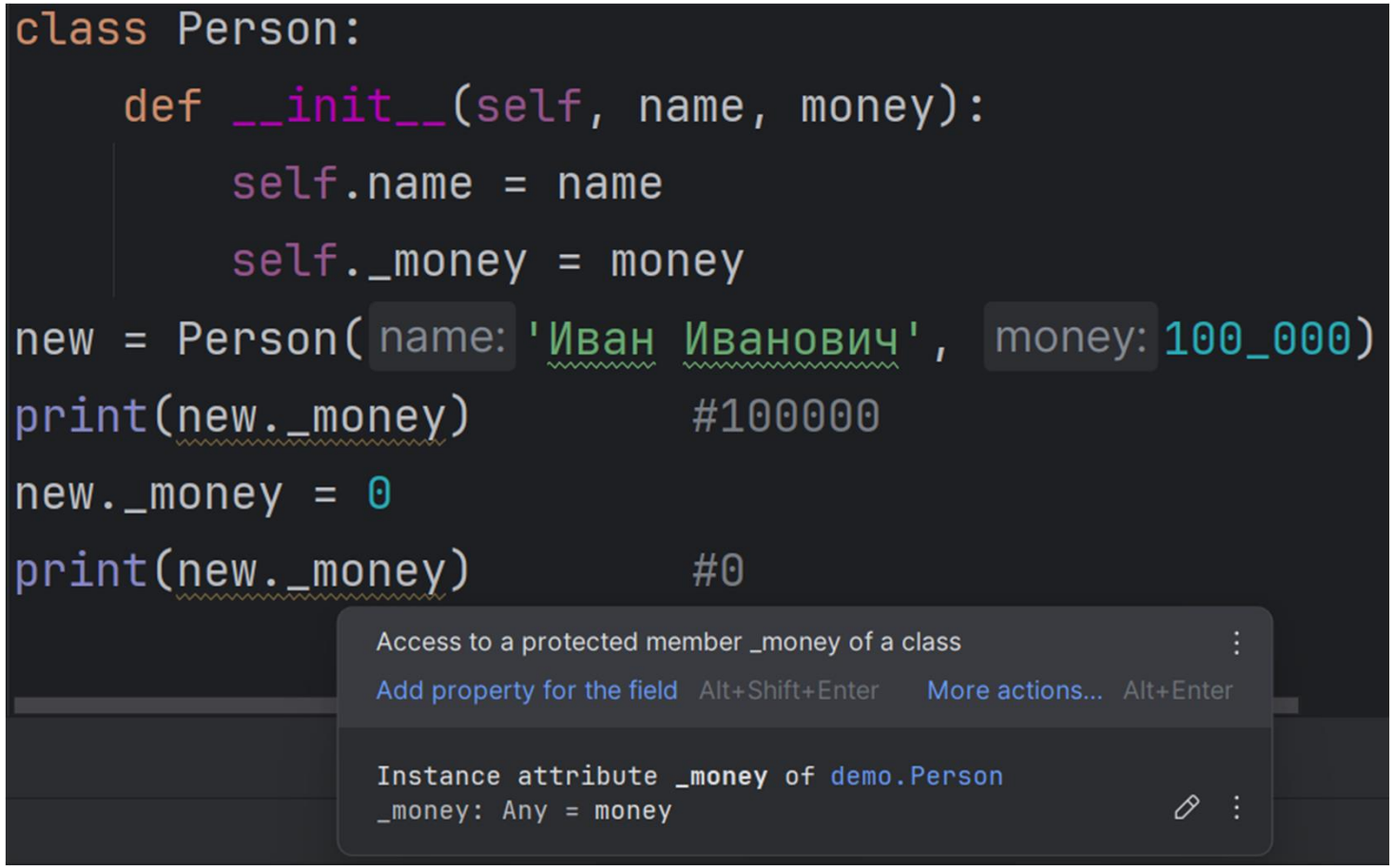
Такая ситуация – недопустима! Это может привести не только к программным ошибкам, но и к материальным убыткам пользователя.

Для ограничения доступа к данным, вы можете установить режим **protected** (данные доступны для обращения внутри класса и в дочерних классах) или **private** (служит для обращения только внутри класса)**.**

До сих пор все создаваемые атрибуты были **публичными**.

protected

Для создания защищенного атрибута, необходимо использовать **одно подчеркивание:**



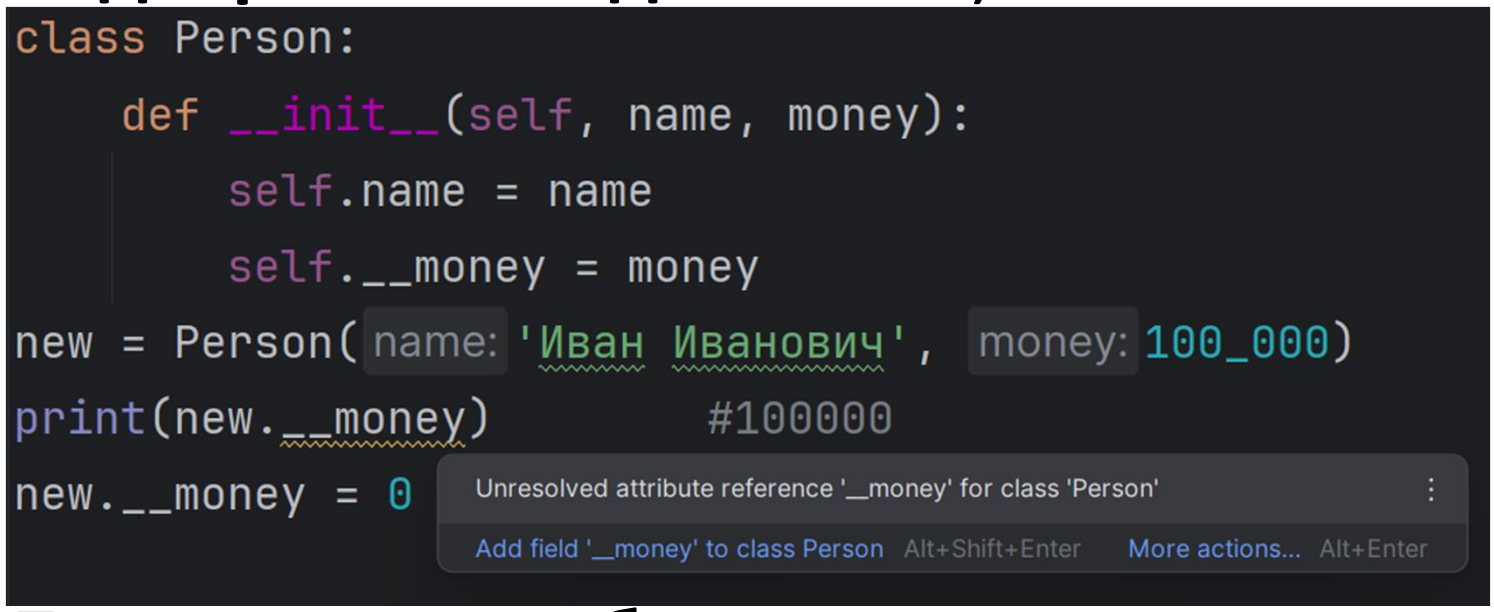
protected

Как вы видите, при изменение защищенных значений ошибки не появляются. Возникает вопрос – для чего они нужны?

Нижнее подчеркивание **предостерегает** программиста от использования этого свойства вне класса, поэтому к такому атрибуту лучше не обращаться напрямую. Одно подчеркивание указывает, что переменная является служебной, для корректной работы класса.

private

Создадим приватный атрибут (**два подчеркивания до имени**):



Приватные атрибуты не доступны в основной области видимости переменных, и мы получаем ошибку – свойство __money не определено.

Setter - Сеттер

Однако, внутри области видимости класса доступ к этим переменным имеется.

Создадим метод **set_balance:**

```
class Person:
    def __init__(self, name, money):
        self.name = name
        self.__money = money
    def set_balance(self, money):
        self.__money = money
new = Person( name: 'Иван Иванович',  money: 100_000)
new.set_balance(0) #Ошибок не возникает
```

Setter - Сеттер

Ошибок не возникает, все в порядке. Такие методы, которые позволяют задавать значения приватных свойств, называются **СЕТТЕРЫ** (от англ. set - Устанавливать)

Такие методы позволяют создать управляемый интерфейс для доступа к данным, внутри которого можно проверять правильность вводимых данных и выполнять прочие необходимые действия (преобразование типов, округление, логирование и т.д.)

Getter - геттер

Создадим метод для отображения приватных атрибутов:

```
class Person:
    def __init__(self, name, money):
        self.name = name
        self.__money = money
    def set_balance(self, money):
        self.__money = money
    def show_balance(self):
        return self.__money
new = Person( name: 'Иван Иванович',  money: 100_000)
print(new.show_balance())
```

Getter - геттер

Ошибок так же не возникло. Такие методы, которые позволяют получить значения приватных атрибутов называются **Геттеры** (от англ. Слова **get** – получать).

С помощью методов – геттеров, вы можете обеспечить ограниченный доступ к данным – например, разрешить просмотр, но не изменение приватного значения.

Сеттеры и **геттеры** называются **интерфейсными методами**

Целостность класса

Сеттеры и геттеры – инструменты для реализации механизма инкапсуляции. Эти инструменты позволяют обеспечить правильную и безошибочную работу классов, их методов и сохранность данных атрибутов.

Упражнение

Реализуйте класс **TaxPayer**, представляющий налогоплательщика. Каждый налогоплательщик имеет ФИО (атрибут экземпляра **name)**,ИНН (атрибут экземпляра **ITIN**) и баланс (Атрибут экземпляра balance). Все атрибуты должны быть приватными. Реализуйте сеттер и геттер методы для каждого атрибута с именами set_name/get_name и т.д.

Упражнение

Реализуйте класс **Point**, представляющий точку в декартовой системе координат. Точка должна иметь два приватных атрибута экземпляра x и y – координаты точки.

Реализуйте **сеттер** и **геттер** методы для отображения координат точки.

Создайте метод **move** с аргументами **dx, dy,** смещающий точку на расстояние относительно текущих координат.

Так же – реализуйте метод **length,** принимающий на вход экземпляр Класса Point, расстояние до которого от текущей точки необходимо вычислить.