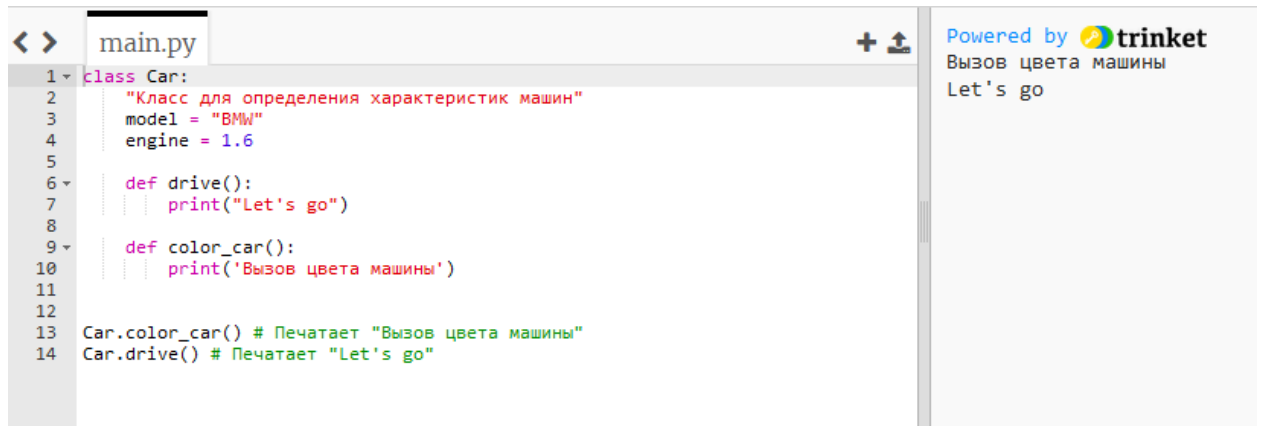



Лекция №2. Методы

Слайд №2

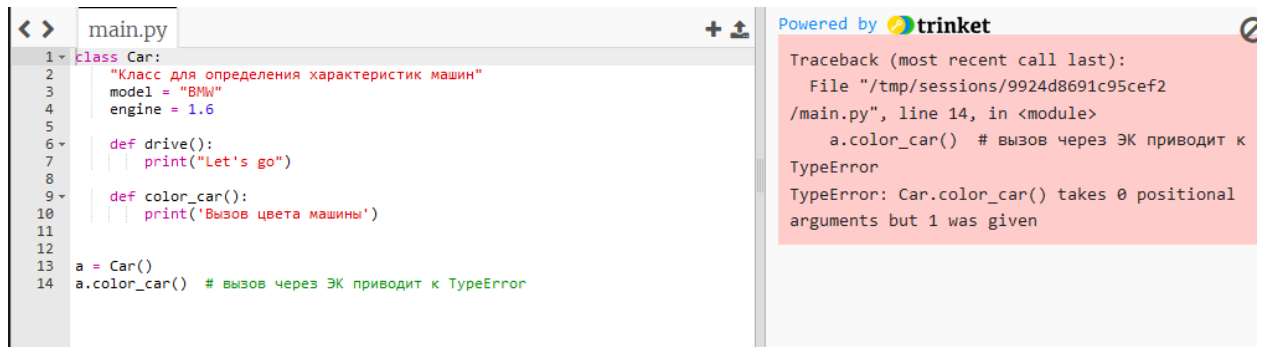
Если обратиться к функции через класс, то ошибки не возникнет:



```
1 class Car:
2     "Класс для определения характеристик машин"
3     model = "BMW"
4     engine = 1.6
5
6     def drive():
7         print("Let's go")
8
9     def color_car():
10        print('Вызов цвета машины')
11
12
13 Car.color_car() # Печатает "Вызов цвета машины"
14 Car.drive() # Печатает "Let's go"
```

Powered by  trinket
Вызов цвета машины
Let's go

Если обратиться к функции через экземпляр класса, то возникнет ошибка TypeError:



```
1 class Car:
2     "Класс для определения характеристик машин"
3     model = "BMW"
4     engine = 1.6
5
6     def drive():
7         print("Let's go")
8
9     def color_car():
10        print('Вызов цвета машины')
11
12
13 a = Car()
14 a.color_car() # вызов через ЭК приводит к TypeError
```

Traceback (most recent call last):
File "/tmp/sessions/9924d8691c95cef2/main.py", line 14, in <module>
a.color_car() # вызов через ЭК приводит к
TypeError
TypeError: Car.color_car() takes 0 positional
arguments but 1 was given

Слайд №3

Все это связано с методом. Метод обозначает функцию, которая определена внутри класса. Отличительная особенность метода от обычной функции в том, что

1. метод связан с ЭК и вызывается от него
2. при вызове метода первым параметром будет передан тот ЭК, от которого метод был вызван.

И вот именно из-за пункта 2 мы получали ошибку: вызывая метод `color_car()` через ЭК, интерпретатор Python автоматически подставляет объект, хранящийся в переменной `a`, в вызываемый метод, а в самом методе мы этот параметр не принимали. Это и приводило к возникновению исключения `TypeError`.

Для вызова метода через экземпляр класса необходимо прописать один аргумент для принятия самого ЭК, от которого будет вызван метод. В следующем примере я назову этот аргумент `self`

Мы можете дать любое название этому аргументу, но в пер8 принято называть его `self`.

```
main.py
1 class Car:
2     "Класс для определения характеристик машин"
3     model = "BMW"
4     engine = 1.6
5
6     def drive(self):
7         print(f"Let's go: {self}")
8
9
10 a = Car() # Создаём экземпляр класса
11 b = Car() # Создаём экземпляр класса
12 a.drive() # Вызываем метод класса
13 b.drive() # Вызываем метод класса
```

Powered by trinket

Let's go: <__main__.Car object at 0x7f8ef75510f0>
Let's go: <__main__.Car object at 0x7f8ef7551120>

Когда запустите данный код, увидите что на экране будут распечатаны два объекта. Обратите внимание на разные адреса объектов в памяти.

Слайд №4

Получение и изменение атрибутов экземпляра в методах

Мы можем использовать ЭК, который передается в метод, для проставления ему новых атрибутов.

ЭК, которые мы с вами создаем, являются изменяемыми объектами, и те атрибуты, что мы им проставляем внутри метода, будут сохранены в самом ЭК. Вот пример:

```
main.py
1 class Car:
2     "Класс для определения характеристик машин"
3     model = "VAZ"
4     engine = 1.6
5     horse_power = 100
6     color = ''
7
8     def drive(instance):
9         print(f"Let's go {instance.model} {instance.engine}")
10
11     def power(instance):
12         print(f"Мощность {instance.model} - {instance.horse_power} лошадиных сил")
13
14     def color(instance):
15         print(f'Цвет {instance.model} - {instance.color}')
16
17     def set_values(instance, new_model, new_engine, new_horse_power, new_color):
18         instance.model = new_model
19         instance.engine = new_engine
20         instance.horse_power = new_horse_power
21         instance.color = new_color
22
23 auto = Car()
24 auto.drive() # Let's go VAZ 1.6
25 auto.model = 'BMW' # Меняем машину
26 auto.drive() # Let's go BMW 1.6
27 auto.power() # Мощность BMW - 100 лошадиных сил
28 auto.horse_power = 350 # Добавим лошадей к мощности
29 auto.power() # Мощность BMW - 350 лошадиных сил
30 auto.set_values('AUDI', 5, 300, 'blue')
31 auto.drive() # Let's go AUDI 5
32 auto.power() # Мощность AUDI - 300 лошадиных сил
```

Powered by trinket

Let's go VAZ 1.6
Let's go BMW 1.6
Мощность BMW - 100 лошадиных сил
Мощность BMW - 350 лошадиных сил
Let's go AUDI 5
Мощность AUDI - 300 лошадиных сил

Слайд №5

Вызов методов через класс

Теперь мы не можем вызывать наш метод от класса, такая попытка приведёт к исключению (ошибке):

```
main.py
1 class Car:
2     "Класс для определения характеристик машин"
3     model = "BMW"
4     engine = 1.6
5
6     def drive(instance):
7         print("Let's go")
8
9     def color_car(instance):
10         print('Вызов цвета машины')
11
12 Car.color_car()
```

Powered by trinket

Traceback (most recent call last):
File "/tmp/sessions/1d2f8988da2adb7b/main.py", line 12, in <module>
Car.color_car()
TypeError: Car.color_car() missing 1 required positional argument: 'instance'

Мы добавили в наш метод один обязательный аргумент, но он не передаётся, что и приводит к ошибке. Для выхода из такой ситуации необходимо при вызове метода передавать переменную с ссылкой на имеющийся экземпляр класса:

```
<> main.py + ↕
1 class Car:
2     "Класс для определения характеристик машин"
3     model = "BMW"
4     engine = 1.6
5
6     def drive(instance):
7         print("Let's go")
8
9     def color_car(instance):
10        print('Вызов цвета машины')
11
12 a = Car()
13 Car.color_car(a) # Вызов цвета машины
14 Car.drive(a) # Let's go
```

Powered by trinket
Вызов цвета машины
Let's go

По сути вызов `Car.color_car(a)` становится эквивалентным `a.color_car()`. Это делается для передачи различных параметров в метод класса, где они обрабатываются и возвращаются в соответствующий экземпляр класса не копируя сам метод:

Слайд №6

```
<> main.py + ↕
1 class Car:
2     "Класс для определения характеристик машин"
3     model = "BMW"
4     engine = 1.6
5
6     def set_color_car(my_obj, color):
7         my_obj.color = color
8         print(f'Теперь у машины {my_obj.color} цвет')
9
10    def get_color_car(my_obj):
11        return my_obj.color
12
13
14 a = Car() # Создаём экземпляр класса a
15 b = Car() # Создаём экземпляр класса b
16 a.set_color_car('black') # Теперь у машины black цвет
17 print('Атрибуты ЭК a:', a.__dict__) # проверяем наличие данного атрибута в a
18 b.set_color_car('red') # Теперь у машины red цвет
19 print('Атрибуты ЭК b:', b.__dict__) # проверяем наличие данного атрибута в b
20 print(a.get_color_car(), b.get_color_car()) # Получаем значения из метода
```

Powered by trinket
Теперь у машины black цвет
Атрибуты ЭК a: {'color': 'black'}
Теперь у машины red цвет
Атрибуты ЭК b: {'color': 'red'}
black red

Теперь при вызове метода `set_color_car()` с дополнительным параметром(аргументом) `color` нам необходимо также передавать значение, а первый аргумент, отвечающий за экземпляр класса, передается по прежнему автоматически.

Слайд №7

Инициализация

Мы создаем ЭК, они обладают своим поведением(методами) и данными(атрибутами). Поведение экземпляра получает от своего класса, для этого мы в классе описываем методы. А вот атрибуты в экземпляр попадают только через присвоение. Ниже пример, в нем есть метод `set_values`, который создает атрибуты для ЭК

```
<> main.py + ↕
1 class Car:
2     "Класс для определения характеристик машин"
3
4
5     def set_values(self, new_model, new_engine, new_horse_power):
6         self.model = new_model
7         self.engine = new_engine
8         self.horse_power = new_horse_power
9
10
11 bmw_3 = Car()
12 print(bmw_3.__dict__)
13 bmw_3.set_values('BMW', 3, 500)
14 print(bmw_3.__dict__)
```

Powered by trinket
{}
{'model': 'BMW', 'engine': 3, 'horse_power': 500}

Но проблема заключается в том, что этот метод нужно вызывать. Что если мы хотим, чтобы при создании ЭК проставлять ему необходимые атрибуты. И для этого в python есть метод инициализации `__init__`.

Слайд №8

Название данного метода начинается и заканчивается двумя нижними подчеркиваниями, в python так обозначаются специальные названия, именуемые магическими методами. Существует достаточно большое количество магических методов. Объединяет их одно - *каждый магический метод будет вызываться автоматически при наступлении своего события.*

Вот конкретно наш магический метод `__init__` будет срабатывать при создании объекта.

```
main.py
1 class Car:
2     "Класс для определения характеристик машин"
3     def __init__(self, new_model, new_engine, new_horse_power):
4         print('method init')
5         self.model = new_model
6         self.engine = new_engine
7         self.horse_power = new_horse_power
8
9     bmw_3 = Car('BMW', 3, 500) # вызывается метод __init__
10    print(bmw_3.__dict__)
11    print('-----')
12    audi_q4 = Car('Audi', 2.5, 400) # вызывается метод __init__
13    print(audi_q4.__dict__)
```

```
Powered by trinket
method init
{'model': 'BMW', 'engine': 3, 'horse_power': 500}
-----
method init
{'model': 'Audi', 'engine': 2.5, 'horse_power': 400}
```

И цель метода инициализации как раз проставить наш вновь созданный ЭК нужными атрибутами.

Необходимо отметить, что при создании параметров в `__init__` мы должны передавать их одновременно с созданием экземпляра класса, иначе в процессе выполнения программы возникнет ошибка:

```
main.py
1 class Car:
2     "Класс для определения характеристик машин"
3     def __init__(self, new_model, new_engine, new_horse_power):
4         print('method init')
5         self.model = new_model
6         self.engine = new_engine
7         self.horse_power = new_horse_power
8
9     bmw_3 = Car() # не передали атрибуты
10    print(bmw_3.__dict__)
11
```

```
Traceback (most recent call last):
  File "/tmp/sessions/8d50c4b1bfdc8537/main.py", line 9, in <module>
    bmw_3 = Car() # не передали атрибуты
TypeError: Car.__init__() missing 3 required positional arguments: 'new_model', 'new_engine', and 'new_horse_power'
```

Слайд №9

При нашей реализации эти аргументы являются обязательными при вызове. Если же вы не хотите передавать некоторые атрибуты при инициализации можно использовать значения по умолчанию внутри метода:

```
main.py
1 class Car:
2     "Класс для определения характеристик машин"
3     def __init__(self, new_model='Car', new_engine=None, new_horse_power=0):
4         print('method init')
5         self.model = new_model
6         self.engine = new_engine
7         self.horse_power = new_horse_power
8
9     bmw_3 = Car()
10    print(bmw_3.__dict__)
11    print('-----')
12    audi_q4 = Car(new_horse_power=350)
13    print(audi_q4.__dict__)
```

```
method init
{'model': 'Car', 'engine': None, 'horse_power': 0}
-----
method init
{'model': 'Car', 'engine': None, 'horse_power': 350}
```

Стоит отметить, что параметры в методе можно называть любыми именами, но желательно применять те названия, которые дают понимание об их назначении. При передаче в `__init__` надо сохранять позиционность передаваемых параметров. Нужно помнить, что устанавливая атрибуты необходимо сопоставлять имена параметров соответствующим атрибутам для их инициализации:

```
#Ранее у нас был такая реализация
def __init__(self, new_model, new_engine, new_horse_power):
    self.model = new_model
```

```
self.engine = new_engine
self.horse_power = new_horse_power
```

#В реальности вы столкнетесь с такой реализацией

```
def __init__(self, model, engine, horse_power):
    self.model = model
    self.engine = engine
    self.horse_power = horse_power
```

Слайд №10

Поскольку данные хранятся в словаре и вызываются методом `__dict__`, можно использовать методы словарей для вывода атрибутов из экземпляра класса, например:

```
main.py
1 class Car:
2     """Класс для определения характеристик машин"""
3     def __init__(self, new_model='Car', new_engine=None, new_horse_power=0):
4         print('method init')
5         self.model = new_model
6         self.engine = new_engine
7         self.horse_power = new_horse_power
8
9
10 auto = Car('BMW', 2.5, 350)
11 print(*auto.__dict__.values()) # получение значений атрибутов
12 print(*auto.__dict__.items()) # получение распакованных атрибутов в форме кортежа
13 print(*auto.__dict__.keys()) # получение ключей словаря
```

Powered by trinket

```
method init
BMW 2.5 350
('model', 'BMW') ('engine', 2.5) ('horse_power', 350)
model engine horse_power
```

Слайд №11

Экземпляры как атрибуты

При моделировании явлений реального мира в программах классы нередко дополняются все большим количеством подробностей. Списки атрибутов и методов растут, и через какое-то время файлы становятся длинными и громоздкими. В такой ситуации часть одного класса нередко можно записать в виде отдельного класса. Большой код разбивается на меньшие классы, которые работают во взаимодействии друг с другом.

Например, у нас есть класс `ElectricCar`, в котором реализованы методы.

```
main.py
1 class ElectricCar:
2     """Класс для создания электромобиля"""
3
4     def __init__(self, maker, model, year, battery_size=70):
5         self.maker = maker
6         self.model = model
7         self.year = year
8         self.battery_size = battery_size
9
10     def describe_battery(self):
11         """Выводит информацию о мощности аккумулятора."""
12         print("This car has a " + str(self.battery_size) + "-kWh battery.")
13
14     def describe_car_info(self):
15         print(f'{self.maker} {self.model} {self.year}'.title())
16
17
18 my_tesla = ElectricCar('tesla', 'model s', 2016)
19 my_tesla.describe_car_info()
20 my_tesla.describe_battery()
```

Powered by trinket

```
Tesla Model S 2016
This car has a 70-kWh battery.
```

В этот класс можно продолжать добавлять новые методы и со временем он разрастется и станет огромным. Есть такой антипаттерн, который называется «[Божественный объект](#)»

Слайд №12

Антипаттерны — полная противоположность паттернам. Если паттерны проектирования — это примеры практик хорошего программирования, то есть шаблоны решения определённых задач. То антипаттерны — их полная противоположность, это — шаблоны ошибок, которые совершаются при решении различных задач.

Божественный объект — анти-паттерн, который довольно часто встречается у ООП разработчиков. Такой объект берет на себя слишком много функций и/или хранит в себе практически все данные. В итоге мы имеем непереносимый код, в котором, к тому же, сложно разобраться. Так же, подобный код довольно сложно поддерживать, учитывая, что вся система зависит практически только от него.

Слайд №13

Бороться с антипаттерном «Божественный объект» следует путем разбивания одного большого класса на несколько небольших. Например, вы видим что есть атрибут и метод, который к нему относится. Мы можем вынести все, что касается аккумулятора в отдельный класс Battery. И при инициализации автомобиля в качестве атрибута класса хранить экземпляра класса Battery

```
1 class Battery:
2     """Простая модель аккумулятора электромобиля."""
3
4     def __init__(self, battery_size=70):
5         self.battery_size = battery_size
6
7     def describe_battery(self):
8         print("This car has a " + str(self.battery_size) + "-kWh battery.")
9
10
11 class ElectricCar:
12     """Класс для создания электромобиля"""
13
14     def __init__(self, maker, model, year):
15         self.maker = maker
16         self.model = model
17         self.year = year
18         self.battery = Battery()
19
20     def describe_car_info(self):
21         print(f'{self.maker} {self.model} {self.year}'.title())
22
23
24 my_tesla = ElectricCar('tesla', 'model s', 2016)
25 my_tesla.describe_car_info()
26 my_tesla.battery.describe_battery()
27 print(my_tesla.battery.battery_size)
```

Tesla Model S 2016
This car has a 70-kWh battery.
70

Слайд №14

МОНОСОСТОЯНИЕ

Как уже мы знаем, все атрибуты экземпляров класса и их значения хранятся в индивидуальных словарях и при изменении значений в ЭК они обновляются только в этих же ЭК, например:

```
<> main.py +
1 class Cat:
2     """Программа о кошках"""
3
4     def __init__(self, breed='', color=''):
5         self.breed = breed
6         self.color = color
7
8 cat1 = Cat()
9 cat2 = Cat()
10 print(cat1.__dict__) # Выведем словарь атрибутов cat1
11 print(cat2.__dict__) # Выведем словарь атрибутов cat2
```

Powered by trinket
{'breed': '', 'color': ''}
{'breed': '', 'color': ''}

Как видно из вывода, индивидуальные словари ЭК пока заполнены одинаковыми данными. Передадим в инициализацию параметры:

```
<> main.py +
1 class Cat:
2
3     def __init__(self, breed='', color=''):
4         self.breed = breed
5         self.color = color
6
7 cat1 = Cat('pers', 'black')
8 cat2 = Cat('siam', 'gray')
9 print(cat1.__dict__) # Выведем словарь атрибутов cat1
10 print(cat2.__dict__) # Выведем словарь атрибутов cat2
```

Powered by trinket
{'breed': 'pers', 'color': 'black'}
{'breed': 'siam', 'color': 'gray'}

В итоге мы видим индивидуальное наполнение словарей ЭК.

У каждого ЭК свои собственные атрибуты и изменение атрибута `breed` у ЭК `cat1` никак не влияет на этот атрибут у экземпляра `cat2`

Но если вы хотите, чтобы у всех ваших экземпляров были одни общие атрибуты, вы можете воспользоваться паттерном «Моносостояние». Он позволяет реализовать одно состояние для атрибутов всех наших ЭК

Слайд №15

Моносостояние можно создать при помощи одного общего словаря `__shared_attr`

```
main.py
1 class Cat:
2     __shared_attr = {          # Формирование единого словаря атрибутов кл
3         'breed': 'pers',      # При изменении значений - эти значения обн
4         'color': 'black'      # во всех экземплярах класса
5     }
6
7     def __init__(self):
8         self.__dict__ = Cat.__shared_attr # Передаём в инициализатор
9                                           # созданный моно-словарь
10
11
12 cat1 = Cat()
13 cat2 = Cat()
14 print(cat1.__dict__) # Проверим словарь атрибутов ЭК cat1
15 print(cat2.__dict__) # Проверим словарь атрибутов ЭК cat
16 cat1.breed, cat1.color = 'pers', 'white' # Меняем атрибуты у cat1
17 print('cat1:', cat1.__dict__)
18 print('cat2:', cat2.__dict__) # Видим, что у cat2 тоже поменялись атри
19 cat2.breed, cat2.color = 'siam', 'gray' # Меняем атрибуты у cat2
20 print('cat1:', cat1.__dict__) # Меняется и у cat1
21 print('cat2:', cat2.__dict__)
```

Powered by trinket

```
{'breed': 'pers', 'color': 'black'}
{'breed': 'pers', 'color': 'black'}
cat1: {'breed': 'pers', 'color': 'white'}
cat2: {'breed': 'pers', 'color': 'white'}
cat1: {'breed': 'siam', 'color': 'gray'}
cat2: {'breed': 'siam', 'color': 'gray'}
```

`__shared_attr` - общий словарь для всего пространства имён, к которому имеет доступ каждый создаваемый экземпляр класса и изменения которого отражаются во всех ЭК. При инициализации мы подменяем личный словарь атрибутов у ЭК на наш общий словарь `__shared_attr`

Обратите внимание на то, как меняются значения в словаре - в независимости под каким ЭК были внесены новые параметры, они меняются во всех экземплярах класса.

Слайд № 16

При этом, мы можем добавлять новые параметры через экземпляр класса, которые также будут отражены в других ЭК, пример:

```
main.py
1 class Cat:
2     __shared_attr = {
3         'breed': 'pers',
4         'color': 'black'
5     }
6
7     def __init__(self):
8         self.__dict__ = Cat.__shared_attr
9
10
11 cat1 = Cat()
12 cat2 = Cat()
13 cat1.weight = 5 # Добавляем параметр в ЭК
14 print('cat1:', cat1.__dict__)
15 print('cat2:', cat2.__dict__)
```

Powered by trinket

```
cat1: {'breed': 'pers', 'color': 'black',
'weight': 5}
cat2: {'breed': 'pers', 'color': 'black',
'weight': 5}
```

Из вывода, становится понятно, что любое изменение моно-словаря приводит к коррекции во всех экземплярах класса.

Слайд №17

Модификаторы доступа в Python ограничивают доступ к методам класса через экземпляры класса. Их использование помогает скрыть данные в классе так, чтобы извне невозможно было нарушить работу этого класса.

Уровни доступа к атрибутам и методам класса в Python можно разделить на:


- ПУБЛИЧНЫЙ (public)
- ПРИВАТНЫЙ(private)
- ЗАЩИЩЁННЫЙ(protected)

Слайд №18

Публичный доступ

В Python каждый атрибут класса или экземпляра класса является публичным по умолчанию. Они могут быть доступны из любой точки за пределами класса. Рассмотрим публичный доступ к атрибутам:

```
main.py
1 class BankAccount:
2
3     def __init__(self, name, balance, passport):
4         self.name = name
5         self.balance = balance
6         self.passport = passport
7
8     def print_data(self): # эти данные доступны из общих вызовов
9         print(self.name, self.balance, self.passport)
10
11
12 account1 = BankAccount('Bob', 100000, 45484564654)
13 account1.print_data()
14 print(account1.name)
15 print(account1.balance)
16 print(account1.passport)
```

Powered by  trinket

Bob 100000 45484564654

Bob
100000
45484564654


В итоге, мы смогли получить доступ к методу класса и к каждому атрибуту ЭК - так функционирует публичный доступ. Это не всегда правильно, т.к. любой человек может получить доступ к данным, которые не должны быть доступны: персональные данные, данные счетов и баланса и др. Для этого в Python существуют и другие режимы доступа.

Слайд №19

Защищенный (protected) режим

Несмотря на свое название, защищенный режим не закрывает доступ к методам и атрибутам класса. Однако существует соглашение в сообществе - имя с префиксом подчеркивания, например `_money`, следует рассматривать как закрытую часть API (будь то функция, метод или элемент данных) для внутреннего служебного использования. И именно нижним подчеркиванием в названии атрибута или метода мы передаём информацию другому разработчику о том, что перед ним `protected` атрибут или метод и его не следует использовать вне класса. Но все это работает только на уровне соглашений, и `protected` атрибуты и методы остаются доступными вне классов.

```
main.py
1 class BankAccount:
2
3     def __init__(self, name, balance, passport):
4         self._name = name # нижним подчеркиванием указываем
5         self._balance = balance # что атрибут должен быть защищен
6         self._passport = passport # от публичного доступа
7
8     def print_protected_data(self):
9         print(self._name, self._balance, self._passport)
10
11
12 account1 = BankAccount('Bob', 100000, 45484564654)
13 account1.print_protected_data()
14 print(account1._name)
15 print(account1._balance)
16 print(account1._passport)
```

Powered by  trinket

Bob 100000 45484564654

Bob
100000
45484564654

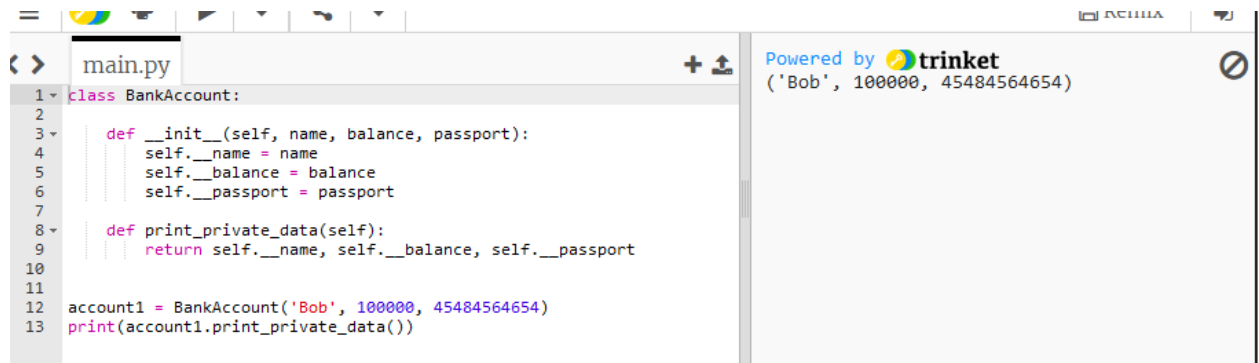
Как видно из вывода, данные всё также доступны для использования вне класса.

Слайд №20


Приватный (private) режим

Как же нам все таки ограничить доступ к атрибутам и методам? Для этого применяется приватный режим доступа (private), который ограничивает доступ вне класса. Приватные методы создаются при помощи двух нижних подчёркиваний, тем самым указывая на приватность атрибута или метода.

Давайте попробуем получить доступ к защищенным атрибутам через специально разрешенный метод:



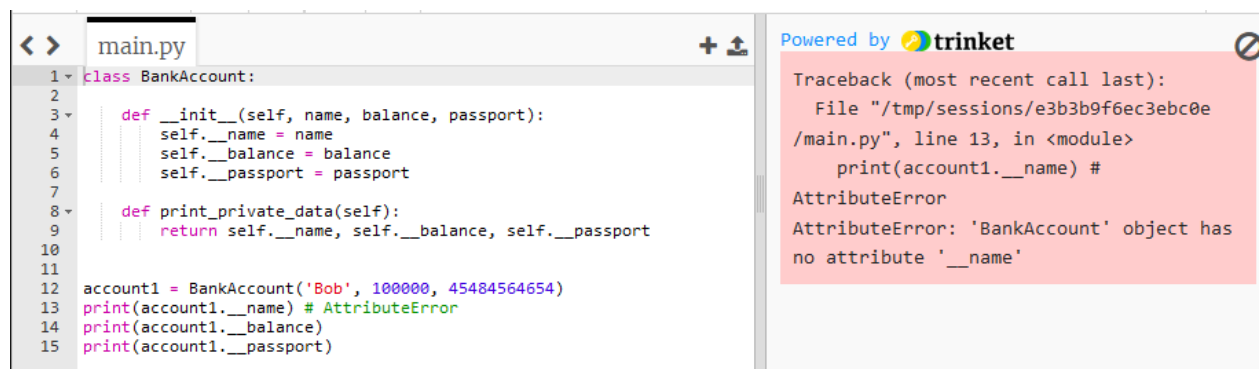
```
1 class BankAccount:
2
3     def __init__(self, name, balance, passport):
4         self.__name = name
5         self.__balance = balance
6         self.__passport = passport
7
8     def print_private_data(self):
9         return self.__name, self.__balance, self.__passport
10
11
12 account1 = BankAccount('Bob', 100000, 45484564654)
13 print(account1.print_private_data())
```

Powered by  trinket
('Bob', 100000, 45484564654)


В результате мы можем получать доступ к данным через специально организованный метод `print_private_data`. Но доступ к атрибутам здесь осуществляется внутри самого класса.

Слайд №21

Если мы попробуем обратиться к приватным атрибутам вне класса, то получим `AttributeError`:



```
1 class BankAccount:
2
3     def __init__(self, name, balance, passport):
4         self.__name = name
5         self.__balance = balance
6         self.__passport = passport
7
8     def print_private_data(self):
9         return self.__name, self.__balance, self.__passport
10
11
12 account1 = BankAccount('Bob', 100000, 45484564654)
13 print(account1.__name) # AttributeError
14 print(account1.__balance)
15 print(account1.__passport)
```

Powered by  trinket

Traceback (most recent call last):
File "/tmp/sessions/e3b3b9f6ec3ebc0e/main.py", line 13, in <module>
 print(account1.__name) #
AttributeError
AttributeError: 'BankAccount' object has no attribute '__name'

Python сообщает, что класс `BankAccount` не имеет атрибута `__name`, тем самым показывая нам сокрытие атрибутов класса от внешнего доступа. Такое сокрытие обработки защищенных атрибутов называется **ИНКАПСУЛЯЦИЯ**, т.е. мы предоставляем пользователю метод `print_private_data` для работы с нашими данными, без возможности непосредственного воздействия на защищенные атрибуты.

Если мы работаем с приватным методом и также не можем обратиться к нему, то мы можем организовать получение данных через публичный метод, например:

```
1 class BankAccount:
2
3     def __init__(self, name, balance, passport):
4         self.__name = name
5         self.__balance = balance
6         self.__passport = passport
7
8     def public_call(self):
9         print('work public method')
10        self.__print_private_data()
11
12    def __print_private_data(self):
13        print('work private method')
14        print(self.__name, self.__balance, self.__passport)
15
16
17 account1 = BankAccount('Bob', 100000, 45484564654)
18 account1.public_call()
```

```
work public method
work private method
Bob 100000 45484564654
```

В заключение необходимо отметить, что приватность доступа к атрибутам и методам класса - это лишь общее соглашение.

Слайд №22

И при желании, обратиться к приватным атрибутам в обход разрешенного доступа через метод `print_private_data` вполне возможно. Для этого достаточно узнать в каком виде хранятся переменные в нашем классе. Это делается при помощи функции `dir`:

```
< > main.py + -
1 class BankAccount:
2
3     def __init__(self, name, balance, passport):
4         self.__name = name
5         self.__balance = balance
6         self.__passport = passport
7
8     def public_call(self):
9         self.__print_private_data()
10
11    def __print_private_data(self):
12        print(self.__name, self.__balance, self.__passport)
13
14
15 account1 = BankAccount('Bob', 100000, 45484564654)
16 print(dir(account1)) # просмотр атрибутов нашего экземпляра класса
```

```
['_BankAccount__balance',
'_BankAccount__name',
'_BankAccount__passport',
'_BankAccount__print_private_data',
'__class__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__eq__',
'__format__', '__ge__',
'__getattr__', '__gt__', '__hash__',
'__init__', '__init_subclass__',
'__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__',
'public_call']
```

Обратите внимание на такие названия как:

- `_BankAccount__balance`
- `_BankAccount__name`
- `_BankAccount__passport`
- `_BankAccount__print_private_data`

Именно через них мы сможем получить доступ к приватным данным вне класса:

```
< > main.py + -
1 class BankAccount:
2
3     def __init__(self, name, balance, passport):
4         self.__name = name
5         self.__balance = balance
6         self.__passport = passport
7
8     def public_call(self):
9         self.__print_private_data()
10
11    def __print_private_data(self):
12        print(self.__name, self.__balance, self.__passport)
13
14
15 account1 = BankAccount('Bob', 100000, 45484564654)
16 print(account1._BankAccount__balance)
17 print(account1._BankAccount__name)
18 print(account1._BankAccount__passport)
19 account1._BankAccount__print_private_data()
```

```
100000
Bob
45484564654
Bob 100000 45484564654
```

Слайд №23

Управление атрибутами в ваших классах

Когда вы определяете класс на объектно-ориентированном языке программирования, вы, вероятно, в конечном итоге получите некоторые атрибуты экземпляра и класса. Другими словами, вы получите переменные, доступные через экземпляр, класс или даже и то, и другого, в зависимости от языка. Атрибуты представляют или содержат внутреннее состояние данного объекта, к которому вам часто потребуется обращаться и изменять его.

Как правило, у вас есть как минимум два способа управления атрибутом. Либо вы можете получить доступ и изменить атрибут напрямую, либо вы можете использовать методы. Методы — это функции, прикрепленные к данному классу. Они обеспечивают поведение и действия, которые объект может выполнять со своими внутренними данными и атрибутами.

Если вы предоставляете свои атрибуты пользователям вашей программы, они становятся частью общедоступного API (*Application Programming Interface* — «программный интерфейс приложения») ваших классов.

Слайд №24

Пользователь вашего класса будет получать к ним доступ и изменять их непосредственно в своем коде. И тут могут возникнуть ситуации, что пользователь попытается сохранить недопустимое значение в атрибуты экземпляров вашего класса. Через сеттер(setter) вы можете повлиять на значение, которое сохраняется в ваш атрибут. А геттер(getter) поможет управлять доступом к вашему атрибуту. А для создания геттеров и сеттеров в Python вам может пригодиться `property`

`property` - эта функция позволяет вам превращать атрибуты класса в свойства или управляемые атрибуты. Поскольку `property()` — это встроенная функция, вы можете использовать ее, ничего не импортируя.

Примечание. Обычно `property` называют встроенной функцией. Однако `property` — это класс, предназначенный для работы как функция, а не как обычный класс. Вот почему большинство разработчиков Python называют это функцией. Это также причина, по которой `property()` не следует соглашению Python по именованию классов.

С помощью `property` вы можете прикрепить методы получения(getter) и установки(setter) к заданным атрибутам класса. Таким образом, вы можете обрабатывать внутреннюю реализацию этого атрибута, не раскрывая методы получения и установки в вашем API. Вы также можете указать способ обработки удаления атрибута и предоставить соответствующую строку документации для ваших свойств.

Слайд №25

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Параметры:

- `fget=None` - функция для получения значения атрибута
- `fset=None` - функция для установки значения атрибута
- `fdel=None` - функция для удаления значения атрибута
- `doc=None` - строка, для [строки документации](#) атрибута

Возвращаемое значение `property` — это сам управляемый атрибут. Если вы обращаетесь к управляемому атрибуту, как в `obj.attr`, тогда Python автоматически вызывает `fget()`. Если вы присваиваете атрибуту новое значение, как в `obj.attr = value`, тогда Python вызывает `fset()`, используя входное значение в качестве аргумента. Наконец, если вы запустите оператор `del obj.attr`, то Python автоматически вызовет `fdel()`.

Примечание. Первые три аргумента функции `property` должны принимать функциональные объекты. Вы можете думать об объекте функции как об имени функции без вызывающей пары круглых скобок

Четвертым аргументом вы можете передать строку документации doc для вашего свойства.

Слайд №26

```
class Person:
    def __init__(self, name):
        self._name = name

    def _get_name(self):
        print("Get name")
        return self._name

    def _set_name(self, value):
        print("Set name")
        self._name = value

    def _del_name(self):
        print("Delete name")
        del self._name

    name = property(
        fget=_get_name,
        fset=_set_name,
        fdel=_del_name,
        doc="The name property."
    )
```

В этом фрагменте создаете класс Person. Инициализатор класса `__init__()` принимает имя в качестве аргумента и сохраняет его в защищенном атрибуте с именем `._name`. Затем вы определяете три непубличных метода:

1. `_get_name()` возвращает текущее значение `._name`
2. `_set_name()` принимает `value` и присваивает его в атрибут экземпляра `._name`
3. `_del_name()` удаляет у экземпляра атрибут `._name`

Слайд №27

```
>>> person = Person('Jack')

>>> person.name
Get name
Jack

>>> person.name = 'Jamal'
Set name
>>> person.name
Get name
Jamal

>>> del person.name
Delete name
>>> person.name
Get name
Traceback (most recent call last):
...
AttributeError: 'Person' object has no attribute '_name'

>>> help(person)
Help on Person in module __main__ object:

class Person(builtins.object)
...
```

```
| name
|     The name property.
```

Слайд №28

Предоставление атрибутов только для чтения

Пожалуй самый элементарный вариант использования `property` — предоставить атрибуты только для чтения в ваших классах. Для достижения этой цели вы можете создать `Person`, как в следующем примере:

```
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    @property
    def name(self):
        return self._name

    @property
    def age(self):
        return self._age
```

Здесь вы сохраняете входные аргументы в атрибутах `_name` и `_age`. Используем подчеркивания (`_`) в именах для того, чтобы сообщить другим разработчикам, что они являются закрытыми атрибутами и к ним нельзя обращаться с помощью записи через точку, например, в `person._age`. Далее, определяем два метода получения через декоратор `@property`. Теперь у вас есть два свойства только для чтения

```
>>> person = Person('Jack', 33)

>>> # Считываем значения
>>> person.name
Jack
>>> person.age
33

>>> # Пытаемся записать новое значение
>>> person.age = 42
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

Слайд №29

Предоставление атрибутов для чтения и записи

Вы также можете использовать `property`, чтобы предоставить управляемым атрибутам возможность и для чтения и для записи. На практике вам просто нужно предоставить соответствующий метод получения («чтение» он же «getter») и метод установки («запись» он же «setter») для ваших свойств, чтобы создать управляемые атрибуты для чтения и записи.

Допустим, вы хотите, чтобы ваш класс `Square` имел атрибут `.area`. Однако получение стороны и площади в инициализаторе класса квадрата кажется ненужным, потому что вы можете вычислить одно, используя другое. Вот `Square`, который управляет `.side` и `.area` как атрибутами для чтения и записи:

```
class Square:
    def __init__(self, side):
        self.side = side

    @property
```

```

def side(self):
    return self._side

@side.setter
def side(self, value):
    self._side = float(value)

@property
def area(self):
    return self.side ** 2

@area.setter
def area(self, value):
    self.side = value ** 0.5

```

Здесь мы создаем класс Square со свойством-атрибутом .side для чтения и записи. В этом случае метод получения(getter) просто возвращает значение стороны квадрата. Метод setter преобразует входное значение стороны и присваивает его закрытой переменной ._side, которую вы используете для хранения окончательных данных.

В этой новой реализации Square и его свойства .side следует отметить одну тонкую деталь. В этом случае инициализатор класса присваивает входное значение свойству .side напрямую, а не сохраняет его в выделенном непубличном атрибуте, таком как ._side.

Почему? Потому что вам нужно убедиться, что каждое значение, предоставленное как сторона квадрата, включая значение инициализации, проходит через метод установки и преобразуется в число с плавающей запятой.

Square также реализует атрибут .area как свойство. Метод getter вычисляет площадь, используя сторону квадрата. Метод setter делает нечто любопытное. Вместо сохранения входного значения площади в специальном атрибуте он вычисляет сторону квадрата и записывает результат вновь в свойство .side.

```

>>> sq = Square(42)

>>> # Считываем значения
>>> sq.side
42.0

>>> sq.area
1764.0

>>> # запишем новое значение
>>> sq.area = 100

>>> sq.side
10.0

```

Предоставление атрибутов только для записи

Вы также можете создать атрибуты только для записи, изменив способ реализации метода получения ваших свойств. Например, вы можете заставить свой метод получения вызывать исключение каждый раз, когда пользователь получает доступ к базовому значению атрибута.

Вот пример обработки паролей со свойством только для записи:

```

class User:
    def __init__(self, name, password):
        self.name = name
        self.password = password

    @property

```

```

def password(self):
    raise AttributeError("Пароль можно только менять, нельзя смотреть")

@password.setter
def password(self, plaintext):
    salt = os.urandom(32)
    self._hashed_password = hashlib.pbkdf2_hmac(
        "sha256", plaintext.encode("utf-8"), salt, 100_000
    )

```

Инициализатор класса User принимает в качестве аргументов имя пользователя и пароль и сохраняет их в атрибут `.name` и в свойство `.password` соответственно. Мы используем свойство для управления тем, как ваш класс обрабатывает входной пароль. Метод получения вызывает `AttributeError` всякий раз, когда пользователь пытается получить текущий пароль. Это превращает `.password` в атрибут-свойство только для записи:

```

>>> jack = User("Jack", "secret_key")

>>> jack._hashed_password
b'7\x1f+\x02\xc4q\x93\xb6\x98\xb3\r\x9f\x9e\xa4v\nI\xde\x10\x11\x98\xb7\xcf\xff\x9c\x83f\xe4\x07\x8c\xce\xc8'

>>> jack.password
Traceback (most recent call last):
...
raise AttributeError("Пароль можно только менять, нельзя смотреть")

>>> jack.password = "new_secret"

>>> jack._hashed_password
b'H\xd3f\xe5\x92,\xc4\xe6\xf29g\xe0\x96I\xd1\xf3^\xd6D\xb4\xbd\x89\xc8\x85s\x13\xa6YA\x08\x89\x89'

```

В этом примере мы создаем экземпляр пользователя `john` с начальным паролем через свойство `password`. Метод установки хэширует пароль и сохраняет его в защищенном атрибуте `._hashed_password`. Обратите внимание, что когда вы пытаетесь получить доступ к `.password` напрямую, вы получаете `AttributeError`. Наконец, присвоение нового значения `.password` запускает метод установки и создает новый хешированный пароль.

В методе установки `.password` мы используем `os.urandom()` для генерации 32-байтовой случайной строки в качестве соли вашей хеш-функции. Чтобы сгенерировать хешированный пароль используем `hashlib.pbkdf2_hmac()`. Затем вы сохраняете полученный хешированный пароль в закрытом атрибуте `._hashed_password`. Это гарантирует, что вы никогда не сохраните открытый текстовый пароль в каком-либо извлекаемом атрибуте.