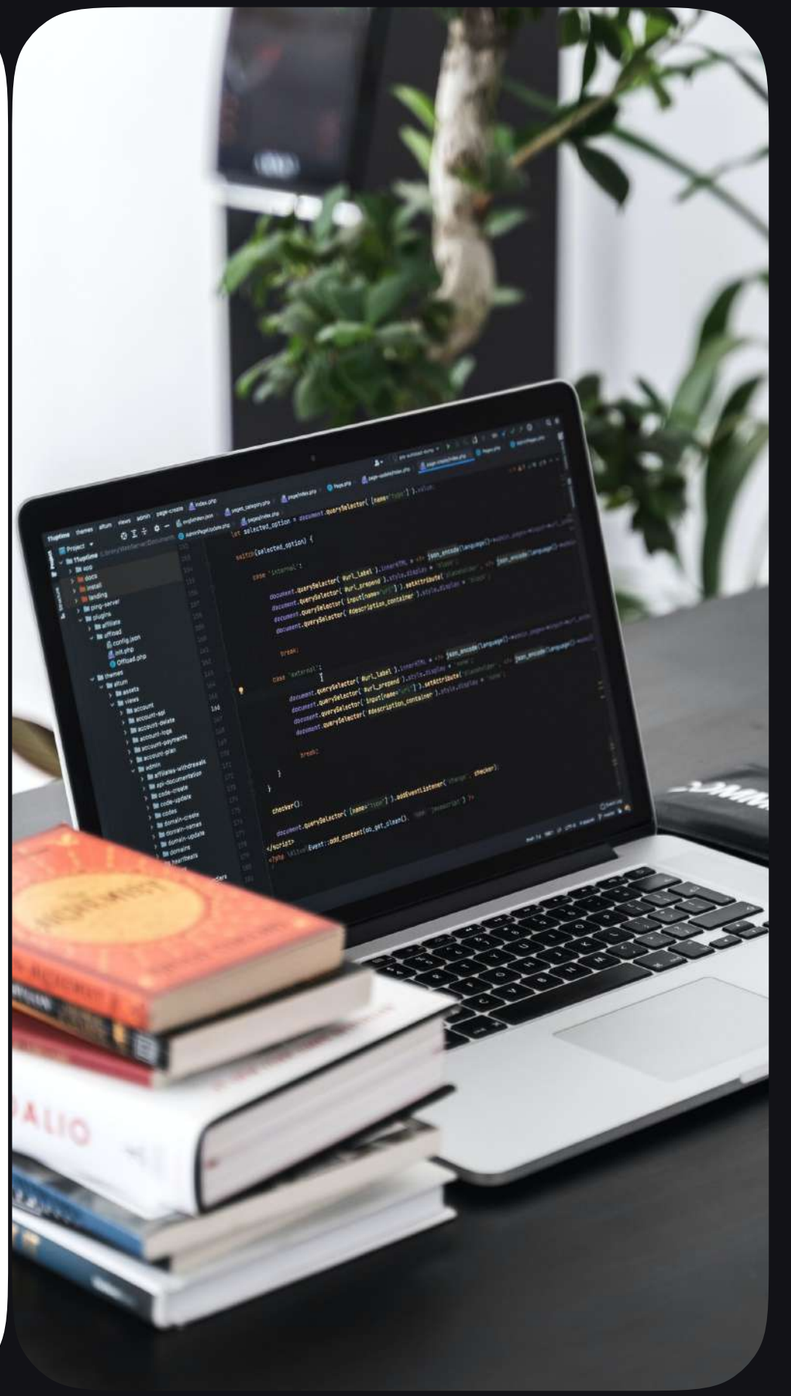
Модуль 2 Занятие 8

Инкапсуляция



Инкапсуляция

Режимы доступа

Геттеры
и сеттеры

Магические
методы для
атрибутов

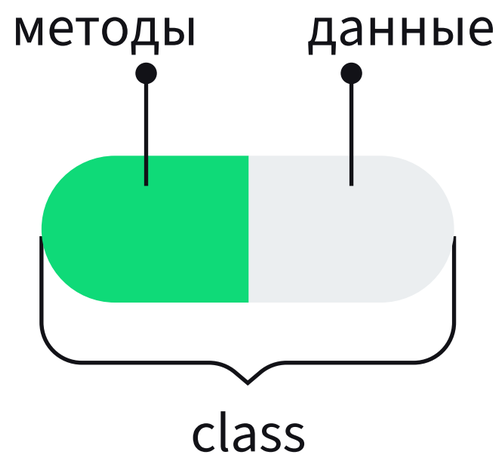
Инкапсуляция

Инкапсуляция — это принцип ООП, который заключается в сокрытии внутреннего устройства класса за **интерфейсом** — разрешенных методов и свойств.

На практике это означает помещение данных и методов для их обработки в одном месте, в «капсуле», и изменение данных только с помощью методов. Для этого можно использовать ограничения на прямой доступ к данным.

Класс является примером инкапсуляции, поскольку он содержит данные и методы в одном месте.

Инкапсуляция



Пример

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


person = Person('Иван', 15)
print(person.name)
print(person.age)
person.age = 166
print(person.age)
```

Создадим класс Person с атрибутами name и age.

Согласно концепции инкапсуляции, необходимо предоставить методы для работы с атрибутами объекта и, если возможно, ограничить прямой доступ к атрибутам.

Вывод:

```
Иван
15
166
```

Пример

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

    def change_age(self):
        self.age += 1

    def rename(self, newname):
        self.name = newname

    def get_info(self):
        return f'Имя: {self.name}, возраст: {self.age}'


person = Person('Иван', 15)
person.change_age()
person.age = 166
print(person.age)
```

Добавим методы для управления атрибутами.

Однако мы все равно можем получить прямой доступ к атрибутам.

Вывод:

```
166
```

Ограничение доступа

```python
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def change_age(self):
        self.__age += 1

    def rename(self, newname):
        self.__name = newname

    def get_info(self):
        return f'Имя: {self.__name}, возраст: {self.__age}'


person = Person('Иван', 15)
person.change_age()
print(person.get_info())
print(person.__age)
```

Ограничить прямой доступ к атрибутам можно, используя различные режимы доступа к атрибутам.

Например: __age — приватный атрибут, получить доступ к нему можно только внутри класса.

Ошибка:

```
Имя: Иван, возраст: 16
AttributeError: 'Person' object has no attribute '__age'
```

Режимы доступа

Сокрытие данных может быть достигнуто путем объявления атрибутов и методов класса как защищенных и приватных. Для этого используется одинарное подчеркивание и двойное подчеркивание перед именем атрибута или метода:

- **public** — публичный атрибут, доступен вне класса
- **_protected** — защищенный атрибут, доступен внутри класса и его дочерних классах (начинается с подчеркивания)
- **__private** — приватный атрибут, доступен только внутри класса (начинается с двойного подчеркивания)

Пример

```
class Person:
    def __init__(self, name, age, passport):
        self.name = name                # публичный атрибут (доступен вне класса)
        self._age = age                 # защищенный атрибут (доступен внутри класса и дочерних классах)
        self.__passport = passport      # приватный атрибут (доступен только внутри класса)



person = Person('Иван', 15, 1234567890)
print(person.__passport)
```

Ошибка:

```
AttributeError: 'Person' object has no attribute '__passport'
```

В приведенном выше примере номер паспорта (__passport) – это приватный атрибут. Чтобы получить доступ к нему, необходимо создать публичный метод.

Режим доступа на уровне соглашений

На самом деле в Python нет механизма, который реально ограничивал бы доступ к атрибутам и методам. В Python режим доступа определяется только на уровне соглашений:

- Атрибут с одним подчеркиванием (_protected) следует рассматривать как внутреннюю деталь реализации, которая должна предостерегать от использования этого атрибута или метода вне класса или дочерних классов.
- Атрибут с двумя подчеркиваниями (__private) явно предостерегает от использования его вне класса, и попытка сделать это приведет к ошибке AttributeError.

Пример

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


person = Person('Иван', 15, 1234567890)
person._age = 116
print(person._age)
```

Вывод:

```
116
```

К защищенному атрибуту все равно можно получить доступ.

Но одинарное подчеркивание перед атрибутом должно предостерегать от прямого использования этого атрибута вне класса.

Пример

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


person = Person('Иван', 15, 1234567890)
person._age = 116
# print(person.__passport)
print(person._Person__passport)
```

Вывод:

```
1234567890
```

Атрибут с двумя подчеркиваниями (__private) на самом деле буквально заменяется на _classname__private, где classname это имя класса.

То есть по-прежнему можно получить доступ или изменить переменную, которая считается приватной.

Сеттеры и геттеры

Реализовать инкапсуляцию в Python можно также с помощью **сеттеров** и **геттеров** — это методы, которые работают с приватными и защищенными атрибутами.

Сеттеры и геттеры используются не только чтобы избежать прямого доступа к приватным и защищенным атрибутам, но и для дополнительной проверки при установке значений атрибутов.

Пример

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

    def get_passport(self):
        return self.__passport

    def set_passport(self, passport):
        if isinstance(passport, int) and len(str(passport)) == 10:
            self.__passport = passport
        else:
            print('Неверный номер паспорта')


person = Person('Иван', 15, 1234567890)
person.set_passport('Иван')
print(person.get_passport())
```

Метод set_passport()
для изменения и проверки
корректности приватного
атрибута __passport.
А метод get_passport()
для получения этого атрибута.

Вывод:

```
Неверный номер паспорта
1234567890
```

@property

```python
class Person:
    def __init__(self, name, age, passport):
        self.name = name
        self._age = age
        self.__passport = passport

    @property
    def passport(self):
        return self.__passport

    @passport.setter
    def passport(self, passport):
        if isinstance(passport, int) and len(str(passport)) == 10:
            self.__passport = passport
        else:
            print('Неверный номер паспорта')


person = Person('Иван', 15, 1234567890)
person.passport = 123450
person.passport = 1111111111
print(person.passport)
```

Также в Python есть декоратор @property, который значительно упрощает использование геттеров и сеттеров.

Вывод:

```
Неверный номер паспорта
1111111111
```

@property

```
class Person:
    def __init__(self, name, age, passport):
        self.__passport = passport

    @property
    def passport(self):
        return self.__passport

    @passport.setter
    def passport(self, passport):
        ...
        self.__passport = passport


...
person.passport = 1111111111
print(person.passport)
```

1. Геттер находится выше сеттера и они называются одинаково
2. Над геттером ставится декоратор @property
3. Над сеттером ставится декоратор @имя_атрибута_геттера.setter

Теперь, при обращении к атрибуту passport, будет вызываться либо геттер, либо сеттер.

Магические методы для атрибутов

- **def __setattr__(self, key, value): ...** — метод вызывается при изменении атрибута key класса
- **def __getattribute__(self, item): ...** — метод вызывается при получении атрибута класса с именем item
- **def __getattr__(self, item): ...** — метод вызывается при получении несуществующего атрибута item класса
- **def __delattr__(self, item): ...** — метод вызывается при удалении атрибута item

Пример

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

    def __getattribute__(self, item):
        if item == 'passport':
            return 'Нет доступа'
        return super().__getattribute__(item)


person = Person('Иван', 15, 1234567890)
print(person.name)
print(person.passport)
```

Метод __getattribute__ вызывается при получении атрибута класса с именем name. Ограничим доступ к атрибуту passport. Для других атрибутов вызываем метод родительского класса, который вернет значение разрешенного атрибута.

Вывод:

```
Иван
Нет доступа
```

Пример

```
class Person:
    def __init__(self, name, age, passport):
        self.name = name
        self.age = age
        super().__setattr__('passport', passport)

    def __setattr__(self, key, value):
        if key == 'name' or key == 'age':
            super().__setattr__(key, value)
        else:
            print('Нет доступа')

person = Person('Иван', 15, 1234567890)
person.name = 'Иван Иванов'
person.passport = 1111111111
print(person.name, person.passport)
```

Метод __setattr__ вызывается при изменении атрибута класса с именем name. Ограничим доступ к атрибуту passport. Для других атрибутов вызываем метод родительского класса, который установит значение разрешенного атрибута.

Вывод:

```
Нет доступа
Иван Иванов 1234567890
```

Итоги

- Инкапсуляция — это принцип ООП, который заключается в сокрытии внутреннего устройства класса за интерфейсом — разрешенных методов и свойств.
- В Python режим доступа определяется только на уровне соглашений:
  - _protected — защищенный атрибут, доступен внутри класса и его дочерних классах
  - __private — приватный атрибут, доступен только внутри класса
- К приватному атриубут можно получить доступ по имени _classname__private, где classname это имя класса.
- сеттеры и геттеры — это методы которые работают с приватными и защищенными атрибутами.
- Декоратор @property упрощает использование геттеров и сеттеров.
- Ограничить доступ к атрибутам можно с помощью магических методов.