Что такое инкапсуляция в Python

Инкапсуляция является одним из основных принципов объектно-ориентированного программирования (ООП), который позволяет скрыть внутреннюю реализацию объекта и предоставить контролируемый доступ к его данным через специальные методы. В Python инкапсуляция реализуется с помощью модификаторов доступа и методов-аксессоров.

Основы инкапсуляции в Python

В Python для создания приватных атрибутов используется двойное подчеркивание __ перед именем атрибута. Это делает атрибуты недоступными для прямого обращения извне класса, обеспечивая защиту данных.

Пример 1: Класс Car с инкапсуляцией

```
class Car:
    def __init__(self, make, model, year):
        self.__make = make  # Приватный атрибут
        self.__model = model
        self.__year = year

    def get_make(self):
        return self.__make

    def get_model(self):
        return self.__model

    def get_year(self):
        return self.__year

    def set_make(self, make):
        self.__make = make

    def set_model(self, model):
        self.__model = model

    def set_year(self, year):
        if year > 1900:  # Добавляем валидацию
            self.__year = year
        else:
            print("Некорректный год выпуска")

    def display_info(self):
        print(f"Марка: {self.__make}, Модель: {self.__model},
Год: {self.__year}")

# Использование класса
car = Car("Toyota", "Camry", 2020)
car.display_info()  # Вывод: Марка: Toyota, Модель: Camry, Год:
```

```
2020

# Изменение атрибутов через методы
car.set_year(2021)
car.display_info()  # Вывод: Марка: Toyota, Модель: Camry, Год:
2021

# Попытка прямого доступа к атрибутам вызовет ошибку
# print(car.__make)  # AttributeError: 'Car' object has no
attribute '__make'
```

Инкапсуляция с проверкой данных

Одним из главных преимуществ инкапсуляции является возможность добавления проверок при изменении данных объекта.

Пример 2: Класс Employee с валидацией

```
class Employee:
    def __init__(self, name, age, salary):
        self.__name = name
        self.__age = age
        self.__salary = salary

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def get_salary(self):
        return self.__salary

    def set_name(self, name):
        if name and len(name.strip()) > 0:
            self.__name = name.strip()
        else:
            print("Имя не может быть пустым")

    def set_age(self, age):
        if 18 <= age <= 65:
            self.__age = age
        else:
            print("Возраст должен быть от 18 до 65 лет")

    def set_salary(self, salary):
        if salary > 0:
            self.__salary = salary
```

```
        else:
            print("Зарплата должна быть положительной")

    def display_info(self):
        print(f"Имя: {self.__name}, Возраст: {self.__age},
Зарплата: {self.__salary} руб.")

# Создание и использование объекта
employee = Employee("Иван", 30, 50000)
employee.display_info()  # Вывод: Имя: Иван, Возраст: 30,
Зарплата: 50000 руб.

# Изменение через валидированные методы
employee.set_salary(60000)
employee.set_age(35)
employee.display_info()  # Вывод: Имя: Иван, Возраст: 35,
Зарплата: 60000 руб.
```

Практический пример: Банковский счет

Рассмотрим более сложный пример с банковским счетом, где инкапсуляция критически важна для безопасности операций.

Пример 3: Класс BankAccount

```
class BankAccount:
    def __init__(self, account_number, initial_balance=0):
        self.__account_number = account_number
        self.__balance = initial_balance
        self.__transaction_history = []

    def get_account_number(self):
        return self.__account_number

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            self.__transaction_history.append(f"Пополнение:
+{amount}")
            print(f"Счет пополнен на {amount}. Текущий баланс:
{self.__balance}")
        else:
            print("Сумма пополнения должна быть положительной")

    def withdraw(self, amount):
```

```
        if amount <= 0:
            print("Сумма снятия должна быть положительной")
        elif amount > self.__balance:
            print("Недостаточно средств на счете")
        else:
            self.__balance -= amount
            self.__transaction_history.append(f"Снятие: -
{amount}")
            print(f"Снято {amount}. Текущий баланс:
{self.__balance}")

    def get_transaction_history(self):
        return self.__transaction_history.copy()

    def display_info(self):
        print(f"Номер счета: {self.__account_number}, Баланс:
{self.__balance} руб.")

# Использование класса
account = BankAccount("12345678", 1000)
account.display_info()  # Вывод: Номер счета: 12345678, Баланс:
1000 руб.

# Операции со счетом
account.deposit(500)     # Счет пополнен на 500. Текущий баланс:
1500
account.withdraw(300)    # Снято 300. Текущий баланс: 1200
account.withdraw(2000)   # Недостаточно средств на счете

# Просмотр истории операций
history = account.get_transaction_history()
for transaction in history:
    print(transaction)
```

Инкапсуляция в классе Student

Пример 4: Класс Student с расширенной валидацией

```
class Student:
    def __init__(self, name, age, grade):
        self.__name = name
        self.__age = age
        self.__grade = grade
        self.__subjects = {}

    def get_name(self):
        return self.__name
```

```
    def get_age(self):
        return self.__age

    def get_grade(self):
        return self.__grade

    def set_name(self, name):
        if name and len(name.strip()) > 0:
            self.__name = name.strip()
        else:
            print("Имя не может быть пустым")

    def set_age(self, age):
        if 16 <= age <= 25:
            self.__age = age
        else:
            print("Возраст студента должен быть от 16 до 25
лет")

    def set_grade(self, grade):
        if 0 <= grade <= 100:
            self.__grade = grade
        else:
            print("Оценка должна быть от 0 до 100")

    def add_subject(self, subject_name, score):
        if 0 <= score <= 100:
            self.__subjects[subject_name] = score
        else:
            print("Оценка по предмету должна быть от 0 до 100")

    def get_subjects(self):
        return self.__subjects.copy()

    def calculate_average(self):
        if self.__subjects:
            return sum(self.__subjects.values()) /
len(self.__subjects)
        return 0

    def display_info(self):
        print(f"Имя: {self.__name}, Возраст: {self.__age}, Общая
оценка: {self.__grade}")
        if self.__subjects:
            print("Предметы:")
```

```
            for subject, score in self.__subjects.items():
                print(f"  {subject}: {score}")
            print(f"Средний балл:
{self.calculate_average():.2f}")

# Создание и использование объекта
student = Student("Анна", 20, 90)
student.display_info()

# Добавление предметов
student.add_subject("Математика", 95)
student.add_subject("Физика", 88)
student.add_subject("Химия", 92)

student.display_info()

# Изменение оценки с валидацией
student.set_grade(95)
student.set_grade(105)  # Некорректное значение, не изменит
оценку
```

Преимущества инкапсуляции

1. Защита данных: предотвращает случайное или некорректное изменение важных атрибутов объекта
2. Контроль доступа: позволяет добавлять проверки и ограничения при изменении данных
3. Модульность: упрощает поддержку и изменение кода без влияния на другие части программы
4. Безопасность: скрывает внутреннюю реализацию от внешнего воздействия

Заключение

Инкапсуляция в Python является мощным инструментом для создания надежных и безопасных объектно-ориентированных программ. Использование приватных атрибутов и методов-аксессоров позволяет контролировать доступ к данным объекта и обеспечивает целостность программы. Правильное применение инкапсуляции делает код более читаемым, безопасным и легким для сопровождения.