УДК 004.423 **Информационные технологии**

Халевин Тимофей Анатольевич, *студент направления подготовки информатика и вычислительной техники, Хакасский государственный университет имени Н.Ф. Катанова, г. Абакан, Россия*

Голубничий Артем Александрович, *научный руководитель, старший преподаватель кафедры ПОВТиАС, Хакасский государственный университет имени Н.Ф. Катанова, г. Абакан, Россия*

КЛАССЫ В ЯЗЫКЕ ПРОГРАММИРОВАНИЯ PYTHON

Аннотация: Данная статья рассматривает классы в Python, описывает их методы и принципы наследования.

Ключевые слова: Python, классы, наследование, сокрытие.

Annotation: This article looks at classes in Python, describing their methods and inheritance principles.

Keywords: Python, classes, inheritance, hiding.

Python – это простой, читаемый и мощный язык программирования. Его чистый синтаксис и интерпретируемость делают его идеальным для начинающих и опытных разработчиков. С его мультипарадигмальной поддержкой, Python позволяет создавать разнообразные приложения, включая веб-сайты, научные вычисления, анализ данных и даже искусственный интеллект [1].

Благодаря обширной стандартной библиотеке и активному сообществу разработчиков, Python предлагает множество инструментов и ресурсов, облегчающих разработку. Его простота и универсальность сделали Python одним из самых популярных языков программирования в мире, используемым в различных сферах от разработки приложений до научных исследований.

Python отлично подходит для объектно-ориентированного программирования (ООП) и активно использует эту концепцию. ООП в Python основан на создании классов, которые определяют структуру и поведение объектов. Классы в Python позволяют организовать данные и функции, связанные с этими данными, в единый объект.

Python поддерживает основные принципы ООП, такие как наследование, инкапсуляцию и полиморфизм. Наследование позволяет создавать новые классы, наследуя свойства и методы существующих классов, что способствует повторному использованию кода и организации данных. Инкапсуляция позволяет скрыть детали реализации от пользователя, предоставляя только интерфейс для работы с объектом. Полиморфизм позволяет использовать объекты разных классов с общим интерфейсом, что упрощает работу с ними.

Объектно-ориентированное программирование в Python облегчает разработку более модульных и масштабируемых программ, упрощает понимание и обслуживание кода, а также способствует повышению его повторного использования. Комбинация лаконичного синтаксиса и мощных инструментов ООП делает Python прекрасным выбором для разработчиков, желающих использовать объектно-ориентированный подход в своих проектах.

Классы в Python являются основой объектно-ориентированного программирования. Они представляют собой структуру, описывающую атрибуты (переменные) и методы (функции), которые объект данного класса может иметь. Классы создаются с использованием ключевого слова *class*, за которым следует имя класса.

Атрибуты класса представляют данные, хранящиеся в объекте, и могут быть доступны через экземпляр класса. Методы класса – это функции, определенные внутри класса, которые могут выполнять операции над объектами данного класса.

В Python конструктор класса представляется методом с именем *__init__* Этот метод вызывается при создании нового объекта класса и инициализирует его. Он не является обязательным, но чрезвычайно полезен для инициализации

атрибутов объекта [2].

Конструктор выполняет инициализацию атрибутов объекта при его создании, позволяя установить начальные значения для этих атрибутов. Он имеет доступ к объекту, который он инициализирует, через параметр *self*. На рисунке 1 представлен пример создания класса.

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

# Создание экземпляра класса Human
person = Human("Иван", 30)

# Вывод информации об экземпляре
print(f"Имя: {person.name}, Возраст: {person.age}") # Имя: Иван, Возраст: 30
```

Рисунок 1 – Создание класса с использованием конструктора

В данном примере класс *Human* имеет конструктор *__init__*, который принимает параметры *name* и *age*, устанавливает их в соответствующие атрибуты *self.name* и *self.age*.

Метод *__init__* не является обязательным для определения класса. Если метод *__init__* не определен в классе, Python автоматически создаст пустой конструктор по умолчанию, который не делает ничего, кроме инициализации пустого объекта.

Следует отметить, что отсутствие метода *__init__* может быть допустимым в некоторых случаях, особенно если класс не требует явной инициализации атрибутов при создании экземпляров.

Пример класса без явного указания конструктора представлен на рисунке 2.

```python
class Car:
    type = "Легковой"

    def display_info(self):
        return f"Тип транспортного средства: {self.type}"

# Создание экземпляра класса Car
my_car = Car()

# Вызов метода для отображения информации об автомобиле
print(my_car.display_info()) # Тип транспортного средства: Легковой
```

Рисунок 2 – Создание класса без явного указания конструктора

Ключевое слово *self* используется в методах класса для ссылки на экземпляр, к которому применяется данный метод. Оно представляет собой ссылку на сам объект, позволяя методам класса получать доступ к атрибутам и другим методам этого объекта.

Когда вызывается метод класса, первым параметром (кроме статических методов) автоматически передается ссылка на сам экземпляр этого класса, и этот параметр обычно называется *self*. Это позволяет методам работать с конкретным экземпляром, к которому они применяются.

В большинстве случаев явно создавать конструктор __init__ будет правильным решением.

В Python имена атрибутов класса могут быть скрыты с использованием нижних подчеркиваний. Это не изменяет функциональности атрибутов, но является соглашением между программистами о том, что атрибут не должен быть использован вне класса. Существуют два основных способа скрытия атрибутов с помощью подчеркиваний, защищенный (protected) и приватный (private).

Атрибуты, которые начинаются с одного нижнего подчеркивания, считаются защищенными. Это соглашение о том, что эти атрибуты не должны использоваться за пределами класса или подклассов, хотя технически к ним можно получить доступ. Создание защищенного атрибута представлена на

рисунке 3.

```python
class MyClass:
    def __init__(self):
        self._protected_attr = 10

    def get_protected_attr(self):
        return self._protected_attr

# Создание экземпляра класса
obj = MyClass()

# Доступ к "protected" атрибуту
print(obj._protected_attr) # 10, но так лучше не делать
print(obj.get_protected_attr())  # 10, лучше делать так
```

Рисунок 3 – Создание защищенного атрибута класса

Атрибуты, которые начинаются с двух нижних подчеркиваний, имеют переименование (name mangling). Python изменяет имя атрибута, делая его менее доступным извне класса. Это предотвращает конфликты имен в подклассах. На рисунке 4 представлен пример создания приватного атрибута класса.

```python
class MyClass:
    def __init__(self):
        self.__private_attr = 20

    def get_private_attr(self):
        return self.__private_attr

# Создание экземпляра класса
obj = MyClass()

# Доступ к "private" атрибуту
print(obj.get_private_attr())  # 20

# Попытка доступа к "private" атрибуту извне класса вызовет ошибку
print(obj.__private_attr)  # Ошибка: 'MyClass' object has no attribute '__private_attr'
print(obj._MyClass__private_attr) # 20, но так делать не нужно
```

Рисунок 4 – Создание приватного атрибута класса

Хотя доступ к атрибутам с нижними подчеркиваниями извне класса все еще возможен, эти соглашения помогают указать другим программистам, что эти атрибуты предполагаются для использования только внутри класса.

В Python можно также создавать защищенные и приватные методы, которые предполагаются для использования только внутри класса или его подклассов. Создаются они по аналогии с атрибутами используя одиночное или двойное нижнее подчеркивание перед названием метода.

Так же у класса есть специальные методы (dunder methods):

1. __init__(self, *args, **kwargs): Вызывается при создании нового экземпляра класса.
2. __str__(self): Возвращает строковое представление объекта.
3. __repr__(self): Возвращает «представление» объекта, используется для отладки, вызывается функцией repr().
4. __len__(self): Возвращает длину объекта, вызывается функцией len().
5. __add__(self, other): Определяет поведение для оператора сложения +.
6. __sub__(self, other): Определяет поведение для оператора вычитания -.
7. __eq__(self, other): Определяет поведение для оператора равенства ==.

Пример использования этих методов представлен на рисунке 5.

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.example_list = [1, 2, 3, 4, 5]

    def __str__(self):
        return f"Первая точка: {self.x}, вторая точка: {self.y}"

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __len__(self):
        return len(self.example_list)

    def __sub__(self, other):
        return Point(self.x - other.x, self.y - other.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

point1 = Point(1, 2)
point2 = Point(3, 4)

print(point1)  # Первая точка: 1, вторая точка: 2
print(point1 + point2)  # Первая точка: 4, вторая точка: 6
print(len(point1)) # 5
print(point1 - point2) # Первая точка: -2, вторая точка: -2
print(point1 == point2)  # Результат: False
```

Рисунок 5 – Использование специальных методов класса

В Python наследование применяется не только к методам, но и к атрибутам класса. Дочерний класс наследует все атрибуты от родительского класса.

Если дочерний класс не имеет своего собственного определения атрибута, он обращается к родительскому классу для доступа к этому атрибуту. Однако, если дочерний класс имеет собственное определение атрибута с тем же именем, то используется атрибут дочернего класса. На рисунке 6 представлено наследование классов.

```python
class Cat:
    def __init__(self):
        self.family = "Кошачьи"

class Tiger(Cat):
    def __init__(self, name):
        super().__init__() # передача полей родительского класса Cat дочернему классу Tiger
        self.name = name

    def __str__(self):
        return f'Имя: {self.name}, Семейство: {self.family}'
```

Рисунок 6 – Наследование классов

В случае множественного наследования в Python, дочерний класс может наследовать атрибуты как от нескольких родительских классов, так и их последовательных потомков. Порядок, в котором классы указаны в скобках при определении класса, определяет приоритет наследования, если у них есть одинаковые имена атрибутов или методов.

Если атрибут есть в нескольких родительских классах, Python будет искать его в порядке, в котором родительские классы указаны в скобках при определении дочернего класса. При обращении к атрибуту, Python будет использовать значение из первого класса в списке, в котором он найдет данный атрибут.

Пример множественного наследования представлен на рисунке 7.

```python
class Animal:
    def __init__(self, weight, height):
        self.weight = weight
        self.height = height

class Family:
    def __init__(self, family):
        self.family = family

class Dog(Animal, Family):
    def __init__(self, name, weight, height, family):
        Animal.__init__(self, weight, height)
        Family.__init__(self, family)
        self.name = name

    def __str__(self):
        return f'Вес: {self.weight}, Рост: {self.height}, Семейство: {self.family}, Кличка: {self.name}'

dog = Dog(name="Жулик", family="Псовые", weight=30, height=70)
print(dog)
```

Рисунок 7 – Множественное наследование класса

В этом примере класс Dog наследует атрибуты классов Animal и Family. Для наследования этих полей мы используем конструкцию *Class.__init__(self, поля)*.

Заключение

В мире программирования объектно-ориентированный подход играет важную роль, а Python, как язык, предоставляет обширные средства для реализации этого подхода. Одним из ключевых элементов в объектно-ориентированном программировании является наследование, позволяющее создавать иерархии классов, обеспечивая повторное использование кода и структурирование программы.

Библиографический список:

1. Бэрри П. Изучаем программирование на Python [Текст] / П. Бэрри. – М.: Вильямс, 2014. – 243 с.
2. Гэддис Т. Начинаем программировать на Python [Текст] / Т. Гэддис. – СПб.: БХВ-Петербург, 2021. – 768 с.