Разработка интернет-приложений

Python

Основные понятия объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) является методологией разработки программного обеспечения, в основе которой лежит понятие класса и объекта, при этом сама программа создается как некоторая совокупность объектов, которые взаимодействую друг с другом и с внешним миром.

Объектно-ориентированная парадигма имеет несколько принципов:

- Данные структурируются в виде объектов, каждый из которых имеет определенный тип, то есть принадлежит к какому-либо классу.
- Классы – результат формализации решаемой задачи, выделения главных ее аспектов.
- Внутри объекта инкапсулируется логика работы с относящейся к нему информацией.
- Объекты в программе взаимодействуют друг с другом, обмениваются запросами и ответами.
- При этом объекты одного типа сходным образом отвечают на одни и те же запросы.
- Объекты могут организовываться в более сложные структуры, например, включать другие объекты или наследовать от одного или нескольких объектов.

Объектно-ориентированное программирование используется, чтобы:

- структурировать информацию и не допускать путаницы;
- точно определять взаимодействие одних элементов с другими;
- повышать управляемость программы;
- быстрее масштабировать код под различные задачи;
- лучше понимать написанное;
- эффективнее поддерживать готовые программы;
- внедрять изменения без необходимости переписывать весь код.

Выделяют четыре основных «столпа» ООП – это абстракция, инкапсуляция, наследование и полиморфизм.

Классы в Python. Создание классов и объектов

Создание класса в Python начинается с инструкции class. Класс состоит из объявления (инструкция class), имени класса и тела класса, которое содержит атрибуты и методы.

```
class название_класса:
    атрибуты_класса
    методы_класса
```

Вот так будет выглядеть минимальный класс.

```
class Rectangle:
    pass   #  в нашем минимальном классе есть только одна инструкция pass
```

Для того чтобы создать объект класса необходимо воспользоваться следующим синтаксисом:

```
имя_объекта = ИмяКласса()
```

```
r1 = Rectangle()
r2 = Rectangle()
```

Инициализатор класса

Для создания объекта класса используется конструктор. Если конструктор не указывать явно, то используется конструктор по умолчанию, который не принимает параметров и который неявно имеют все классы. Если мы хотим явным образом определить действия при создании объекта, то сделать это можно с помощью специального метода, который называется `__init__()` (по два прочерка с каждой стороны). Это инициализатор класса, который вызывает неявный конструктор класса и проводит необходимый набор действий для задания параметров объекта.

```
class Rectangle:
    def __init__(self, width, height ) -> None: #  "-> None" необязательная
        # часть, которая аннотирует, что метод конструктора класса
        # не должен ничего возвращать
        self.width = width
        self.height = height
        print("Прямоугольник создан")

r1 = Rectangle(3, 5)
r2 = Rectangle(6, 8)
```

Конструктор должен принимать как минимум один параметр – ссылку на текущий объект – **`self`**. Обычно конструкторы применяются для определения действий, которые должны производиться при создании объекта.

Конструктор фактически представляет обычную функцию, только для вызова конструктора используется не `__init__`, а название класса. Кроме того, при вызове конструктора параметру `self` явным образом не передается никакого значения. При выполнении программы `Python` динамически будет определять `self`.

Статические и динамические атрибуты класса

Класс может содержать атрибуты и методы. Атрибут может быть статическим и динамическим (уровня объекта класса). Суть в том, что для работы со статическим атрибутом, вам не нужно создавать экземпляр класса, а для работы с динамическим – нужно.

Абстрагирование — это способ выделить набор наиболее важных атрибутов и методов и исключить незначимые. Соответственно, абстракция — это использование всех таких характеристик для описания объекта. Важно представить объект минимальным набором полей и методов без ущерба для решаемой задачи.

```
class Rectangle:
    default_color = "green"
    def __init__(self, width, height ):
        self.width = width
        self.height = height

r1 = Rectangle(3, 5)
r2 = Rectangle(6, 8)

print(r1.default_color) # green
print(Rectangle.default_color) # green
```

В представленном выше классе, атрибут `default_color` – это статический атрибут, и доступ к нему, как было сказано выше, можно получить не создавая объект класса `Rectangle`. `width` и `height` – это динамические атрибуты, при их создании было использовано ключевое слово `self`. Для доступа к `width` и `height` предварительно нужно создать объект класса `Rectangle`

Методы классов

Методы класса представляют функции, которые определенны внутри класса и которые определяют его поведение. При определении методов любого класса, как и инициализатора, первый параметр метода представляет ссылку на текущий объект, который для методов уровня объекта класса называется `self`. Через эту ссылку внутри класса мы можем обратиться к функциональности текущего объекта. Но при самом вызове метода этот параметр не учитывается. Если метод должен принимать другие параметры, то они определяются после параметра `self` и при вызове подобного метода для них необходимо передать значения

Используя имя объекта, мы можем обратиться к его методам. Для обращения к методам применяется нотация точки – после имени объекта ставится точка и после нее идет вызов метода.

```
class Rectangle:
    def __init__(self, width, height ):
        self.width = width
        self.height = height

    def square(self):
        return self.width*self.height

    def grow(self, k):
        self.width *=k
        self.height *= k

r1 = Rectangle(3, 5)

print(r1.square()) # 15
r1.grow(2)
print(r1.square()) # 60
```

Методы классов

Помимо методов объектов класса (обычных методов), есть еще статические методы и классовые методы. Для реализации этих методов необходимо добавлять декораторы `@staticmethod` и `@classmethod` перед определением метода. В случае реализации обычных методов декораторы не добавляются. Использовать статические и классовые методы можно и к классу и к объекту класса, а обычные методы только к объекту.

```
class Rectangle:
    default_color="green"
    def __init__(self, width, height ):
        self.width = width
        self.height = height
        print(f"{ self.default_color = }")

    @classmethod
    def help(cls):
        print(f"Some help text. Use {cls.default_color}")

    @staticmethod
    def hint():
        print(f"Some hint text")

r1 = Rectangle(3, 5)  # self.default_color = 'green'
Rectangle.help()   # Some help text. Use green
Rectangle.hint()   # Some hint text
r1.hint()          # Some hint text
```

Специальные методы

Строковое представление объекта `__str__(self)` – вызывается при печати объекта.

```
class Rectangle:
    def __init__(self, width, height ):
        self.width = width
        self.height = height

    def __str__(self) -> str:
        return f"Прямоугольник шириной  {self.width},  высотой {self.height}"

r1 = Rectangle(3, 5)
print(r1) # Прямоугольник шириной  3,  высотой 5
```

Конструктор класса и инициализация экземпляра класса

В Python разделяют конструктор класса и метод для инициализации экземпляра класса.

Метод `__init__(self)` является инициализатором класса, а конструктором класса является метод `__new__(cls, *args, **kwargs)`.

При этом, `__new__` – это классовый метод, а `__init__` таким не является. Метод `__new__` редко переопределяется, чаще используется реализация от базового класса `object`, `__init__` же наоборот является очень удобным способом задать параметры объекта при его создании.

```
class Rectangle:
    def __new__(cls, *args, **kwargs):
        print("Hello from __new__")
        return super().__new__(cls)

    def __init__(self, width, height):
        print("Hello from __init__")
        self.width = width
        self.height = height

r1 = Rectangle(3, 5)
```

```
----------------
Hello from __new__
Hello from __init__
```

Инкапсуляция

Под инкапсуляцией понимается сокрытие деталей реализации, данных и т.п. от внешней стороны.

Например, можно определить класс «холодильник», который будет содержать следующие данные: производитель, объем, количество камер хранения, потребляемая мощность и т.п., и методы: открыть/закрыть холодильник, включить/выключить, но при этом реализация того, как происходит непосредственно включение и выключение пользователю вашего класса не доступна, что позволяет ее менять без опасения, что это может отразиться на использующей класс «холодильник» программе. При этом класс становится новым типом данных в рамках разрабатываемой программы и можно создавать объекты этого нового типа.

Инкапсуляция

Все объекты в Python инкапсулируют внутри себя данные и методы работы с ними, предоставляя публичные интерфейсы для взаимодействия.

Атрибут, как и метод, может быть объявлен приватным (внутренним) с помощью нижнего подчеркивания перед именем, но настоящего скрытия на самом деле не происходит – все на уровне соглашений.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self._special_calculations()

    # не предназначена для вызова из-вне, но вызвать можно
    def _special_calculations(self):
        self.__very_special_calculations()

    # можно вызвать только внутри класса
    def __very_special_calculations(self):
        pass

r1 = Rectangle(3, 5)
r1._special_calculations()        # выполнится
r1.__very_special_calculations() # будет ошибка
```

Если поставить перед именем атрибута два подчеркивания, к нему нельзя будет обратиться напрямую (на самом деле есть обходной путь, но мы про него говорить не будем).

Наследование

Под наследованием понимается возможность создания нового класса на базе существующего.

Когда один класс становится наследником другого, то все атрибуты класса-предка (надкласса, superclass) становятся доступны классу-потомку (подклассу, subclass) — наследуются (достаются в наследство).

Примером базового класса, демонстрирующего наследование, можно определить класс «автомобиль», имеющий атрибуты: масса, мощность двигателя, объем топливного бака и методы: завести и заглушить. У такого класса может быть потомок – «грузовой автомобиль», он будет содержать те же атрибуты и методы, что и класс «автомобиль», и дополнительные свойства: количество осей, мощность компрессора и т.п.

В Python следующие две записи равнозначны:

```
class MyClass:
    pass

class MyClass(object):
    pass
```

То есть, если класс-предок не указан, то таковым считается `object` — самый базовый класс в Python.

Наследование

```
class Counter:
    def __init__(self):
        self.value = 0

    def inc(self):
        self.value += 1

    def dec(self):
        self.value -= 1

#  Наследник Counter
class NonDecreasingCounter(Counter):  # в скобках указан класс-предок
    def dec(self): # переопределяем метод dec
        pass
    def inc(self): # переопределяем метод inc
        super().inc() # вызываем родительский метод
        super().inc()

c = NonDecreasingCounter()
c.inc()
print(c.value) # 2
c.dec()
print(c.value) # 2
```

Полиморфизм

Полиморфизм (polymorphism) — это понятие из объектно-ориентированного программирования, которое позволяет разным сущностям выполнять одни и те же действия. При этом неважно, как эти сущности устроены внутри и чем они различаются.

```
class Apple:
    def use(self):
        return "Вы съели яблоко. Ваше здоровье увеличилось."

class Stone:
    def use(self):
        return "Вы бросили камень. Он улетел далеко."

class Cloud:
    def use(self):
        return "Вы попытались использовать облако... Ничего не произошло."

def interact_with_object(object):
    print(object.use())

# Создание объектов
apple = Apple()
stone = Stone()
cloud = Cloud()

# Взаимодействие с объектами
interact_with_object(apple)  # Вы съели яблоко. Ваше здоровье увеличилось.
interact_with_object(stone)  # Вы бросили камень. Он улетел далеко.
interact_with_object(cloud)  # Вы попытались использовать облако... Ничего не произошло.
```

Основные виды полиморфизма

Полиморфизм не является чем-то однородным. Он имеет несколько разновидностей, каждая из которых служит своей цели. В программировании выделяют три ключевых вида полиморфизма: подтипов, параметрический и ad-hoc.

- Полиморфизм подтипов позволяет объектам дочерних классов быть использованными вместо объектов родительского класса. Это значит, что функции или методы, ожидающие объект родительского класса, могут работать с объектами дочерних классов без изменений.
- Параметрический полиморфизм дает возможность функции обрабатывать данные любого типа. В этом случае функция определяется без указания конкретных типов данных, которые она может принимать. Пример – функция `len()`
- Ad-hoc полиморфизм, или перегрузка функций, позволяет использовать одно и то же имя функции для различных типов данных, изменяя поведение функции в зависимости от принимаемых аргументов.

Подключение пакетов и модулей

В языке программирования Python подключение пакетов и модулей осуществляется с помощью `import`. Это позволяет распределять код по логическим "узлам" приложения(модели данных, обработчики, и тп.), что позволяет получить менее нагруженные кодом файлы.

Импортирование модулей

```
import math
import random
import datetime

print(math.pi)
print(random.randint(0, 100))
print(datetime.datetime.now())
```

Изменение названия модуля и функций

```
import math as m
print(m.sin(m.pi/2))
```

Использование from … import

Оператор `from` полезен тем, что нам не обязательно указывать название модуля перед использованием функции. Данный процесс упрощает скорость написания кода. В начале указываем какую функцию импортируем из модуля и пишем код.

```
from math import pi
from random import randint
print(pi)
print(randint(0, 100))
```

```
from random import randint as rdi
print(rdi(0, 100))
```

Свои модули и пакеты

В языке программирования Python модулями являются все файлы с расширением *.py (* обозначает, что на этом месте может стоять любой символ или любое их количество). Исключением является служебный файл __init__.py

Файл my_module.py

```
class MyClass:
    def __str__(self):
        return "Hello!"
```

Файл test.py

```
import my_module
c = my_module.MyClass()
print(c)


from my_module import MyClass
c = MyClass()
print(c)
```

Свои модули и пакеты

В языке программирования Python пакетами являются все директории(вне зависимости от наличия в них модулей), содержащие файл __init__.py, который исполняется при импорте пакета. __init__.py – это файл зависимостей, который помогает Python находить доступные модули в каталоге пакета. Если мы удалим этот файл, то Python не сможет импортировать модули.

Пакет с единственным модулем __init__.py при импорте ведёт себя как обычный модуль. Содержимое инициализирующего модуля определяет атрибуты объекта пакета.

Прочие модули пакета и вложенные пакеты не импортируются автоматически вместе с пакетом-родителем, но могут быть импортированы отдельно с указанием полного имени.

Задачи

1. Создайте класс Student, который имеет:
   - атрибуты name, age, group_name, pass_scores (имя, возраст, группа, оценки);
   - метод average_pass_score – для вычисления среднего балла успеваемости.
   - метод add_pass_scores – для добавления новых оценок к имеющимся.

   Оценки могут быть только 3, 4, 5.

2. Создайте класс Sphere, который имеет:
   - атрибуты c_x, c_y, c_z, r (координаты центра и радиус);
   - метод volume – для вычисления объема сферы.
   - метод is_inside, который для координат x, y, z проверяет попадает ли точка в сферу.

   Радиус строго больше нуля.