GeekBrains

Основы Python

Урок 8.
ООП. Полезные дополнения

Восьмой урок курса посвящён важным дополнениям, расширяющим возможности парадигмы ООП в Python. К этим дополнениям относятся статические методы и методы класса. Поговорим о встроенных атрибутах и методах. В этом уроке мы рассмотрим процесс написания несложной программы на основе парадигмы ООП. Далее научимся создавать собственные исключения. В конце урока мы познакомим вас с трюками для создания лаконичного кода и расскажем о полезных библиотеках.

На этом уроке

1. Узнаем, как реализовать статический метод и метод класса.
2. Познакомимся с атрибутами и встроенными методами объектов классов.
3. Увидим пример реализации ООП-программы.
4. Научимся создавать и применять собственные исключения.
5. Узнаем о полезных хитростях в Python.
6. Познакомимся с библиотекой psutil, системой управления пакетами pip, инструментом virtualenv.
7. Познакомимся с библиотекой requests.

Оглавление

Статические методы и методы класса

@staticmethod

Обычный метод позволяет выполнять операции с объектами класса. Чтобы получить к нему (методу) доступ, необходимо указать ссылку на соответствующий объект. Однако существуют методы, которые вызываются напрямую через имя класса (статические методы).

Для определения метода в качестве статического используется декоратор **@staticmethod.** Его помещают перед именем метода.

Декоратор — это функция, расширяющая возможности другой функции или метода класса.

Пример:

```
class Auto:
    @staticmethod
    def get_class_info():
        print("Детальная информация о классе")


Auto.get_class_info()
```

Результат:

```
Детальная информация о классе
```

В этом примере реализован класс **Auto,** который имеет один статический метод — **get_class_info()**. Вызов этого метода осуществляется через название класса. Судя по коду, создавать объект класса **Auto** для вызова метода **get_class_info()** не требуется. В примере для получения доступа мы используем имя класса.

Важно! *Статические методы имеют доступ только к атрибутам классов, к ним нельзя обратиться через **self**. По сути, статические методы ничего не знают ни о классе, ни об экземпляре, на который вызываются.*

Рассмотрим ещё один пример:

```
class MyClass:
    @staticmethod
    def on_sum_1(param_1, param_2):  # Статический метод
        return param_1 + param_2

    def on_sum_2(self, param_1, param_2):  # Обычный метод класса
        return param_1 + param_2

    def on_sum_3(self, param_1, param_2):
        return MyClass.on_sum_1(param_1, param_2)  # Вызов статического метода
```

Попробуем вызвать статический метод **on_sum_1()**:

```
print(MyClass.on_sum_1(20, 30))
```

Результат:

```
50
```

Согласно логике работы метода **on_sum_1()**, в качестве первого параметра передаётся ссылка на сам класс, а не на его объект. В этом примере в рамках статического метода **on_sum_1()** получить доступ к атрибутам и методам объекта класса нельзя. Соответственно, мы не можем использовать **self**.

Теперь проверим работу метода **on_sum_2()**:

```
mc = MyClass()
print(mc.on_sum_2(20, 10))
```

Результат:

```
30
```

Поскольку **on_sum_2()** — обычный метод класса, для его вызова сначала необходимо создать объект класса. Метод **on_sum_2()** в качестве первого параметра принимает ссылку объекта класса.

Теперь попробуем вызвать статический метод **on_sum_1()** через объект класса:

```
print(mc.on_sum_1(40, 30))
```

Результат:

```
70
```

И вызвать статический метод **on_sum_1()** через обычный метод класса:

```
print(mc.on_sum_3(50, 50))
```

Результат:

```
100
```

@classmethod

Таким декоратором дополняется метод, который получает класс в качестве первого аргумента.

Пример:

```
class MyClass:
    @classmethod
    def my_method(cls, param):  # Метод класса
        print(cls, param)


MyClass.my_method(30)  # Вызов метода через название класса
mc = MyClass()
mc.my_method(70)  # Вызов метода класса через экземпляр
```

Результат:

```
<class '__main__.MyClass'> 30
<class '__main__.MyClass'> 70
```

В приведённом примере служебная переменная **cls** указывает на класс, а не экземпляр.

Важно! *Через **cls** мы обращаемся к методу класса, а через **self** — к экземпляру класса (объекту).*

Чтобы определить, какой из декораторов использовать (**@staticmethod** или **@classmethod**), необходимо проанализировать логику метода класса. Метод оперирует атрибутами и методами в пределах класса — используем декоратор **@classmethod**. Но если он (метод класса) не выполняет какие-либо операции с другими частями класса, можно использовать декоратор **@staticmethod**.

Атрибуты и встроенные методы объектов классов

Стандартные атрибуты и методы

Атрибут	Описание
`__name__`	Имя класса
`__module__`	Имя модуля
`__dict__`	Словарь с атрибутами класса
`__bases__`	Кортеж с базовыми классами
`__doc__`	Строка документации класса
`__class__`	Объект-класс, экземпляром которого является этот инстанс
`__init__`	Конструктор
`__del__`	Деструктор
`__hash__`	Возвращает хеш-значение объекта, равное 32-битному числу
`__getattr__`	Возвращает атрибут, недоступный обычным способом
`__setattr__`	Присваивает значение атрибуту
`__delattr__`	Удаляет атрибут
`__call__`	Выполняется при вызове экземпляра класса
`__str__`	Строковое представление объекта
`__repr__`	Формальное строковое представление объекта
`__getitem__`	Получение элемента по индексу или ключу
`__setitem__`	Присваивание элемента с данным ключом или индексом
`__delitem__`	Удаление элемента с данным ключом или индексом

Пример:

```
class User:
    def __init__(self, name, login, passwd, email):
        self.name = name
        self.login = login
        self.passwd = passwd
        self.email = email
    def on_get_data(self):
        print(f"имя: {self.name}, логин: {self.login}, "
            f"пароль: {self.passwd}, email: {self.email}")


u = User("Ivan Ivanov", "IvIv", "11111", "iviv@mail.ru")
u.on_get_data()
print(f"__name__ - {User.__name__}, \n __module__ - {User.__module__}, \n"
    f"__dict__ - {User.__dict__}, \n __bases__ - {User.__bases__}, \n"
    f"__doc__ - {User.__doc__}, \n __class__ - {User.__class__}, \n"
    f"__init__ - {User.__init__}, \n __hash__ - {User.__hash__}")
```

Результат:

```
имя: Ivan Ivanov, логин: IvIv, пароль: 11111, email: iviv@mail.ru
__name__ - User,
__module__ - __main__,
__dict__ - {'__module__': '__main__', '__init__': <function User.__init__ at
0x0000007F30BB71E0>, 'on_get_data': <function User.on_get_data at
0x0000007F30BB7268>, '__dict__': <attribute '__dict__' of 'User' objects>,
'__weakref__': <attribute '__weakref__' of 'User' objects>, '__doc__': None},
__bases__ - (<class 'object'>,),
__doc__ - None,
__class__ - <class 'type'>,
__init__ - <function User.__init__ at 0x0000007F30BB71E0>,
__hash__ - <slot wrapper '__hash__' of 'object' objects>
```

Пример ООП-программы

В рамках концепции ООП большую роль играет этап предварительного проектирования. Он включает следующие шаги:

1. Сформулировать задачу.

2. Определить объекты предметной области, участвующие в решении задачи.
3. Выделить классы, на основе которых генерируются объекты. При необходимости определить базовые классы и классы-потомки.
4. Установить основные атрибуты и методы объектов.
5. Создать классы, их атрибуты и методы.
6. Создать объекты классов.
7. Выполнить итоговое решение задачи, организовав взаимодействие объектов.

Разработаем виртуальную модель образовательного процесса. Для этого в программе выделим следующие объекты: студенты, преподаватель, знания.

Для реализации задачи необходимо создать три класса: «Преподаватель», «Студент», «Данные». У групп «Преподаватели» и «Студенты» есть общие параметры, например, имя и фамилия. Получаем надкласс «Персона». Далее определяем у этого класса атрибуты, общие для преподавателя и студента: имя и фамилия.

В классе «Преподаватель» реализуем наследование от группы «Персона». Определяем метод **to_teach()**, который принимает ссылку на экземпляр класса «Предмет», и список студентов, изучающих этот предмет. В методе **to_teach()** для каждого студента вызываем метод **to_take().** Он будет фиксировать усвоение студентом предмета или набора предметов (заполнение списка **knowledges**).

В группе «Студент» также реализуем наследование от класса «Персона» и определяем метод **to_take().** Он будет вносить в список освоенные студентом предметы (полученные знания), новый предмет или список предметов.

Создадим ещё один класс и дадим ему имя «Предмет». Его основная функция — принимать набор названий предметов. Класс будет содержать метод **my_list()**, возвращающий атрибут, то есть список предметов.

Пример:

```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
    def __str__(self):
        return f"Name and surname: {self.name} {self.surname}"
```

```
class Teacher(Person):
    def to_teach(self, subj, *pupils):
        for pupil in pupils:
            pupil.to_take(subj)

class Pupil(Person):
    def __init__(self, name, surname):
        super().__init__(name, surname)
        self.knowledges = []
    def to_take(self, subj):
        self.knowledges.append(subj)

class Subject:
    def __init__(self, *subjects):
        self.subjects = list(subjects)
    def my_list(self):
        return self.subjects
```

Проверим работу кода:

```
s = Subject("maths", "physics", "chemistry")
t = Teacher("Ivan", "Ivanov")
print(t)

p_1 = Pupil("Petr", "Petrov")
p_2 = Pupil("Sergey", "Sergeev")
p_3 = Pupil("Vladimir", "Vladimirov")
print(f"{p_1}; {p_2}; {p_3}")

t.to_teach(s, p_1, p_2, p_3)
print(p_1.knowledges[0].my_list())
```

Результат:

```
Name and surname: Ivan Ivanov
Name and surname: Petr Petrov; Name and surname: Sergey Sergeev; Name and
surname: Vladimir Vladimirov
['maths', 'physics', 'chemistry']
```

Создание собственных исключений

Список основных исключений и их описания приведены в таблице ниже:

Исключение	Описание

Exception	Любое исключение, не являющееся системным
ZeroDivisionError	Попытка деления на ноль
IndexError	Индекс не входит в диапазон элементов
KeyError	Несуществующий ключ
FileExistsError	Попытка создания существующего файла или директории
FileNotFoundError	Файл или директория не существует
IndentationError	Неправильные отступы
TypeError	Несоответствие объекта и типа данных
ValueError	Некорректное значение аргумента функции

Пример:

```
print(100/0)
```

Результат:

```
ZeroDivisionError: division by zero
```

Пример:

```
my_dict = {"k_1": "v_1", "k_2": "v_2", "k_3": "v_3"}
val = my_dict["k_4"]
```

Результат:

```
KeyError: 'k_4'
```

Пример:

```
my_list = [10, 20, 30]
print(my_list[3])
```

Результат:

```
IndexError: list index out of range
```

В представленных примерах возникают исключения, и выполнение кода завершается с ошибкой. Для обработки исключений применяются конструкции **try/except**.

Пример:

```
try:
    print(100/0)
except:
    print("Деление на ноль недопустимо")
```

Результат:

```
Деление на ноль недопустимо
```

Блок **try** содержит инструкции, которые могут привести к возникновению исключения, а в блоке **except** реализован его перехват.

В обработке исключений могут быть задействованы инструкции **else** и **finally**. Первая выполняется при отсутствии исключения, вторая — всегда, независимо, было исключение или нет.

Пример:

```
try:
    res = 100/0
except ZeroDivisionError:
    print("На ноль делить нельзя")
else:
    print(f"Все хорошо. Результат - {res}")
finally:
    print("Программа завершена")
```

В Python существует возможность создания собственных классов-исключений — потомков класса **Exception**.

Пример:

```
class OwnError(Exception):
    def __init__(self, txt):
        self.txt = txt

inp_data = input("Введите положительное число: ")

try:
    inp_data = int(inp_data)
    if inp_data < 0:
        raise OwnError("Вы ввели отрицательное число!")
except ValueError:
    print("Вы ввели не число")
except OwnError as err:
    print(err)
else:
    print(f"Все хорошо. Ваше число: {inp_data}")
```

Результат:

```
Введите положительное число: 5
Все хорошо. Ваше число: 5

Введите положительное число: text
Вы ввели не число

Введите положительное число: -65
Вы ввели отрицательное число!
```

В этом примере в выражении **OwnError («вы ввели отрицательное число!»)** создаётся объект собственного класса-исключения. С помощью оператора **raise** происходит возбуждение исключения. Оно перехватывается во второй ветке **except** и присваивается переменной **err**.

У экземпляров класса **Exception** (и его производных) доступен метод **__str__().** Он предназначен для вывода значений атрибутов. Поэтому обращаться к атрибутам объекта можно следующим образом: **err.txt**.

В процессе работы над Python-программами возникают ситуации, когда код не отрабатывает команды разработчика. При этом явная информация об ошибках отсутствует. Чтобы их найти, нужно воспользоваться следующим механизмом:

Пример:

```
import traceback

def incorrect(a, b):
    return a / b

try:
    res = incorrect(5, 0)
except Exception as e:
    print('Ошибка:\n', traceback.format_exc())
```

Результат:

```
Ошибка:
 Traceback (most recent call last):
    res = incorrect(5, 0)
    return a / b
ZeroDivisionError: division by zero
```

В этом примере используются возможности модуля traceback. Он применяется для сбора и вывода трассировочной информации о программе после появления исключения. Функции в модуле работают с объектами, которые содержат трассировочную информацию. Чаще всего он (модуль) обеспечивает нестандартный механизм вывода информации об ошибках. О возможностях модуля можно узнать по ссылке.

Pip и virtualenv. Особенности использования

Работа с pip

Это популярная система управления пакетами. Она участвует в установке и управлении программными пакетами, реализованными с помощью Python. Начиная с интерпретатора версии 3.4, установка системы pip не требуется.

Пример:

```
pip install numpy
```

Список основных команд pip:

Команда	Описание

pip help	Получить подсказку о доступных командах
pip install package_name	Установить пакет
pip uninstall package_name	Удалить пакет
pip list	Получить список установленных пакетов
pip search package_name	Найти пакет по имени
pip install -U package_name	Обновить указанный пакет
pip show package_name	Получить информацию об установленном пакете

Работа с virtualenv

Под виртуальной средой понимают директорию, содержащую необходимые для работы приложения пакеты. Они позволяют выполнять изолированный запуск приложения. Виртуальная среда автоматически поставляется с собственным интерпретатором Python и отдельным инструментом pip. Та же копия интерпретатора используется при создании среды.

Virtualenv позволяет:

- создавать новую изолированную среду для Python-проекта;
- выполнять простую и быструю упаковку приложений;
- образовывать зависимости для одного проекта;
- обеспечивать портативность между системами.

Установка virtualenv:

```
pip install virtualenv
```

Создание виртуальной среды для проекта:

```
virtualenv my_proj
```

Альтернативная команда создания виртуальной среды для проекта:

```
python -m venv my_proj
```

Активация виртуальной среды для Windows:

```
my_proj\Scripts\activate
```

Активация виртуальной среды для Linux и MacOs:

```
source my_proj/venv/bin/activate
```

Теперь все устанавливаемые приложения будут размещаться в текущей виртуальной среде.

Деактивация виртуальной среды для Windows:

```
my_proj\Scripts\deactivate
```

Деактивация виртуальной среды для Linux и MacOs:

```
source deactivate
```

Библиотека psutil

Позволяет получить информацию о параметрах процессора, памяти, дисков. Это отличная библиотека для управления системой и ресурсами.

Пример:

```
import psutil

#  Информация о системных вызовах и контекстных переключателях
print(psutil.cpu_stats())

#  Информация о диске
print(psutil.disk_usage("D:"))

#  Информация о состоянии памяти
print(psutil.virtual_memory())
```

Результат:

```
scpustats(ctx_switches=230863362, interrupts=176391588, soft_interrupts=0,
syscalls=1151270777)
```

```
sdiskusage(total=892722536448, used=99323600896, free=793398935552,
percent=11.1)

svmem(total=8547123200, available=3777404928, percent=55.8, used=4769718272,
free=3777404928)
```

Библиотека requests

Сторонний инструмент для выполнения запросов и обработки ответов. Одно из ключевых звеньев для парсинга веб-страниц.

Установка:

```
pip install requests
```

Создание запроса

Пример:

```
import requests

resp = requests.get('https://github.com/requests')
print(resp)
print(type(resp))

resp = requests.put('https://github.com/requests/put')
print(resp)
resp = requests.delete('https://github.com/requests/delete')
print(resp)
resp = requests.head('https://github.com/requests/get')
print(resp)
resp = requests.options('https://github.com/requests/get')
print(resp)
```

Результат:

```
<Response [200]>
<class 'requests.models.Response'>
<Response [422]>
<Response [422]>
```

```
<Response [404]>
<Response [404]>
```

В этом примере создаётся подключение. Переменная resp содержит ссылку на объект Response. Средствами библиотеки requests можно выполнять стандартные запросы: PUT, DELETE, HEAD, OPTIONS.

Передача аргументов в запросе

При необходимости передачи аргументов в URL, т. е., формирования запроса, например, https://github.com/requests/get?key=val, можно использовать словарь.

Пример:

```
data = {'key1': 'value1'}
resp = requests.get("https://github.com/requests/get", params=data)
```

Содержимое объекта response

Пример:

```
import requests
resp = requests.get("https://github.com/requests/")
print(resp.text)
```

Коды состояний и заголовки

Пример:

```
import requests
resp = requests.get("https://github.com/requests/")
print(resp.status_code)
print(resp.headers)
```

Практическое задание

1. Реализовать класс «Дата», функция-конструктор которого должна принимать дату в виде строки формата «день-месяц-год». В рамках класса реализовать два метода. Первый, с

декоратором @classmethod. Он должен извлекать число, месяц, год и преобразовывать их тип к типу «Число». Второй, с декоратором @staticmethod, должен проводить валидацию числа, месяца и года (например, месяц — от 1 до 12). Проверить работу полученной структуры на реальных данных.

2. Создайте собственный класс-исключение, обрабатывающий ситуацию деления на ноль. Проверьте его работу на данных, вводимых пользователем. При вводе нуля в качестве делителя программа должна корректно обработать эту ситуацию и не завершиться с ошибкой.
3. Создайте собственный класс-исключение, который должен проверять содержимое списка на наличие только чисел. Проверить работу исключения на реальном примере. Запрашивать у пользователя данные и заполнять список необходимо только числами. Класс-исключение должен контролировать типы данных элементов списка.

   Примечание: длина списка не фиксирована. Элементы запрашиваются бесконечно, пока пользователь сам не остановит работу скрипта, введя, например, команду «stop». При этом скрипт завершается, сформированный список с числами выводится на экран.

   Подсказка: для этого задания примем, что пользователь может вводить только числа и строки. Во время ввода пользователем очередного элемента необходимо реализовать проверку типа элемента. Вносить его в список, только если введено число. Класс-исключение должен не позволить пользователю ввести текст (не число) и отобразить соответствующее сообщение. При этом работа скрипта не должна завершаться.

4. Начните работу над проектом «Склад оргтехники». Создайте класс, описывающий склад. А также класс «Оргтехника», который будет базовым для классов-наследников. Эти классы — конкретные типы оргтехники (принтер, сканер, ксерокс). В базовом классе определите параметры, общие для приведённых типов. В классах-наследниках реализуйте параметры, уникальные для каждого типа оргтехники.
5. Продолжить работу над первым заданием. Разработайте методы, которые отвечают за приём оргтехники на склад и передачу в определённое подразделение компании. Для хранения данных о наименовании и количестве единиц оргтехники, а также других данных, можно использовать любую подходящую структуру (например, словарь).
6. Продолжить работу над вторым заданием. Реализуйте механизм валидации вводимых пользователем данных. Например, для указания количества принтеров, отправленных на склад, нельзя использовать строковый тип данных.

   Подсказка: постарайтесь реализовать в проекте «Склад оргтехники» максимум возможностей, изученных на уроках по ООП.

7. Реализовать проект «Операции с комплексными числами». Создайте класс «Комплексное число». Реализуйте перегрузку методов сложения и умножения комплексных чисел. Проверьте работу проекта. Для этого создаёте экземпляры класса (комплексные числа), выполните

сложение и умножение созданных экземпляров. Проверьте корректность полученного результата.

Дополнительные материалы

1. Статические методы и методы класса.
2. Создание классов-исключений.
3. Библиотека psutil.
4. Краткое руководство по библиотеке Python Requests.

Используемая литература

Для подготовки этого методического пособия были использованы следующие ресурсы:

1. Язык программирования Python 3 для начинающих и чайников.
2. Программирование в Python.
3. Учим Python качественно (habr).
4. Самоучитель по Python.
5. Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 (4-е издание).