Основы Python

Урок 8.

ООП. Полезные дополнения

**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](#_unoqtckto9oa)

[@staticmethod](#_3qe46a1in63i)

[@classmethod](#_unoyyzikxg33)

[Атрибуты и встроенные методы объектов классов](#_sefz9zqhi84e)

[Стандартные атрибуты и методы](#_mdhqwelsvg78)

[Пример ООП-программы](#_cawa7yvavlto)

[Создание собственных исключений](#_r3yid2m3bqj6)

[Pip и virtualenv. Особенности использования](#_v5rsn5bynesz)

[Работа с pip](#_6bsbpzs9qp2o)

[Работа с virtualenv](#_7fry1dl9mprj)

[Библиотека psutil](#_z5nxe5gj5uim)

[Библиотека requests](#_zdzgr9235msw)

[Создание запроса](#_v2eqgkk72oh7)

[Передача аргументов в запросе](#_ppw4tfikmqo6)

[Содержимое объекта response](#_lu1ufptte9pr)

[Коды состояний и заголовки](#_tjdnedu1gap9)

[Дополнительные материалы](#_vqapgkhnyfqp)

[Используемая литература](#_tnflastqfeho)

# 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](http://www.ilnurgi1.ru/docs/python/modules/traceback.html).

# 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) |

# 

# Дополнительные материалы

1. [Статические методы и методы класса](http://python-3.ru/page/staticheskie-metody-i-metody-klassa).
2. [Создание классов-исключений](https://pythoner.name/exceptions-example).
3. [Библиотека psutil](http://www.ilnurgi1.ru/docs/python/modules_user/psutil.html).
4. [Краткое руководство по библиотеке Python Requests](https://pythonru.com/biblioteki/kratkoe-rukovodstvo-po-biblioteke-python-requests).

# Используемая литература

Для подготовки этого методического пособия были использованы следующие ресурсы:

1. [Язык программирования Python 3 для начинающих и чайников](https://pythonworld.ru/).
2. [Программирование в Python](https://python-scripts.com/).
3. [Учим Python качественно (habr)](https://habrahabr.ru/post/150302/).
4. [Самоучитель по Python](http://pythonworld.ru/samouchitel-python).
5. [Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 (4-е издание)](http://www.proklondike.com/books/python/lutz_python_2011.html).