Python

- Декораторы
- Функции
- Разное
- Объектно ориентированное программирование на языке Python
    - Создание и использование вложенных классов
    - Классы и объекты
    - Основные понятия объектно-ориентированного программирования
- Модули и пакеты
- Copy in Python (Deep Copy and Shallow Copy)
- Полезное
- Проектирование в python
    - Разбираемся в REST API с примерами на Python
- Как перебрать словарь в Python
- Инструменты
    - IPython
    - Poetry

??????????

Декораторы — один из самых полезных инструментов в Python, однако новичкам они могут показаться непонятными. Возможно, вы уже встречались с ними, например, при работе с Flask, но не хотели особо вникать в суть их работы. Эта статья поможет вам понять, чем являются декораторы и как они работают.

??? ????? ??????????

Новичкам декораторы могут показаться неудобными и непонятными, потому что они выходят за рамки «обычного» процедурного программирования как в Си, где вы объявляете функции, содержащие блоки кода, и вызываете их. То же касается и объектно-ориентированного программирования, где вы определяете классы и создаёте на их основе объекты. Декораторы не принадлежат ни одной из этих парадигм и исходят из области функционального программирования. Однако не будем забегать вперёд, разберёмся со всем по порядку.

Декоратор — это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода. Вот почему декораторы можно рассматривать как практику метапрограммирования, когда программы могут работать с другими программами как со своими данными. Чтобы понять, как это работает, сначала разберёмся в работе функций в Python.

??? ???????? ???????

Все мы знаем, что такое функции, не так ли? Не будьте столь уверены в этом. У функций Python есть определённые аспекты, с которыми мы нечасто имеем дело, и, как следствие, они забываются. Давайте проясним, что такое функции и как они представлены в Python.

??????? ??? ?????????

С этим аспектом функций мы знакомы лучше всего. Процедура — это именованная последовательность вычислительных шагов. Любую процедуру можно вызвать в любом месте программы, в том числе внутри другой процедуры или даже самой себя. По этой части больше нечего сказать, поэтому переходим к следующему аспекту функций в Python.

??????? ??? ??????? ??????? ??????

В Python всё является объектом, а не только объекты, которые вы создаёте из классов. В этом смысле он (Python) полностью соответствует идеям объектно-ориентированного программирования. Это значит, что в Python всё это — объекты:

- числа;
- строки;
- классы (да, даже классы!);
- функции (то, что нас интересует).

Тот факт, что всё является объектами, открывает перед нами множество возможностей. Мы можем сохранять функции в переменные, передавать их в качестве аргументов и возвращать из других функций. Можно даже определить одну функцию внутри другой.

Иными словами, функции — это объекты первого класса. Из определения в Википедии:

> Объектами первого класса в контексте конкретного языка программирования называются элементы, с которыми можно делать всё то же, что и с любым другим объектом: передавать как параметр, возвращать из функции и присваивать переменной.

И тут в дело вступает функциональное программирование, а вместе с ним — декораторы.

?????????????? ???????????????? — ??????? ?????? ????????

В Python используются некоторые концепции из функциональных языков вроде Haskell и OCaml. Пропустим формальное определение функционального языка и перейдём к двум его характеристикам, свойственным Python:

- функции являются объектами первого класса;
- следовательно, язык поддерживает функции высших порядков.

Функциональному программированию присущи и другие свойства вроде отсутствия побочных эффектов, но мы здесь не за этим. Лучше сконцентрируемся на другом — функциях высших порядков. Что есть функция высшего порядка? Снова обратимся к Википедии:

> Функции высших порядков — это такие функции, которые могут принимать в качестве аргументов и возвращать другие функции.

Если вы знакомы с основами высшей математики, то вы уже знаете некоторые математические функции высших порядков порядка вроде дифференциального оператора *d/dx*. Он принимает на входе функцию и возвращает другую функцию, производную от исходной. Функции высших порядков в программировании работают точно так же — они либо принимают функцию(и) на входе и/или возвращают функцию(и).

???? ????????

Раз уж мы ознакомились со всеми аспектами функций в Python, давайте продемонстрируем их в коде:

```
def hello_world():
    print('Hello world!')
```

Здесь мы определили простую функцию. Из фрагмента кода далее вы увидите, что эта функция, как и классы с числами, является объектом в Python:

```
>>> def hello_world():
...     print('Hello world!')
...
>>> type(hello_world)
<class 'function'>
>>> class Hello:
...     pass
...
>>> type(Hello)
<class 'type'>
>>> type(10)
<class 'int'>
```

Как вы заметили, функция `hello_world` принадлежит типу `<class 'function'>`. Это означает, что она является объектом *класса* `function`. Кроме того, класс, который мы определили, принадлежит *классу* `type`. От этого всего голова может пойти кругом, но чуть поигравшись с функцией `type` вы со всем разберётесь.

Теперь давайте посмотрим на функции в качестве объектов первого класса.

Мы можем хранить функции в переменных:

```
>>> hello = hello_world
>>> hello()
Hello world!
```

Определять функции внутри других функций:

```
>>> def wrapper_function():
...     def hello_world():
...         print('Hello world!')
...     hello_world()
...
>>> wrapper_function()
Hello world!
```

Передавать функции в качестве аргументов и возвращать их из других функций:

```
>>> def higher_order(func):
...     print('Получена функция {} в качестве аргумента'.format(func))
...     func()
...     return func
...
>>> higher_order(hello_world)
Получена функция <function hello_world at 0x032C7FA8> в качестве аргумента
Hello world!
<function hello_world at 0x032C7FA8>
```

Из этих примеров должно стать понятно, насколько функции в Python гибкие. С учётом этого можно переходить к обсуждению декораторов.

??? ???????? ???????????

Повторим определение декоратора:

> Декоратор — это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода.

Раз мы знаем, как работают функции высших порядков, теперь мы можем понять как работают декораторы. Сначала посмотрим на пример декоратора:

```
def decorator_function(func):
    def wrapper():
```

```
        print('Функция-обёртка!')
        print('Оборачиваемая функция: {}'.format(func))
        print('Выполняем обёрнутую функцию...')
        func()
        print('Выходим из обёртки')
    return wrapper
```

Здесь `decorator_function()` является функцией-декоратором. Как вы могли заметить, она является функцией высшего порядка, так как принимает функцию в качестве аргумента, а также возвращает функцию. Внутри `decorator_function()` мы определили другую функцию, обёртку, так сказать, которая обёртывает функцию-аргумент и затем изменяет её поведение. Декоратор возвращает эту обёртку. Теперь посмотрим на декоратор в действии:

```
>>> @decorator_function
... def hello_world():
...     print('Hello world!')
...
>>> hello_world()
Оборачиваемая функция: <function hello_world at 0x032B26A8>
Выполняем обёрнутую функцию...
Hello world!
Выходим из обёртки
```

Магия, не иначе! Просто добавив `@decorator_function` перед определением функции `hello_world()`, мы модифицировали её поведение. Однако как вы уже могли догадаться, выражение с `@` является всего лишь синтаксическим сахаром для `hello_world = decorator_function(hello_world)`.

Иными словами, выражение `@decorator_function` вызывает `decorator_function()` с `hello_world` в качестве аргумента и присваивает имени `hello_world` возвращаемую функцию.

И хотя этот декоратор мог вызвать вау-эффект, он не очень полезный. Давайте взглянем на другие, более полезные (наверное):

```
def benchmark(func):
    import time

    def wrapper():
        start = time.time()
        func()
        end = time.time()
        print('[*] Время выполнения: {} секунд.'.format(end-start))
```

```
    return wrapper

@benchmark
def fetch_webpage():
    import requests
    webpage = requests.get('https://google.com')

fetch_webpage()
```

Здесь мы создаём декоратор, замеряющий время выполнения функции. Далее мы используем его на функции, которая делает GET-запрос к главной странице Google. Чтобы измерить скорость, мы сначала сохраняем время перед выполнением обёрнутой функции, выполняем её, снова сохраняем текущее время и вычитаем из него начальное.

После выполнения кода получаем примерно такой результат:

```
[*] Время выполнения: 1.4475083351135254 секунд.
```

К этому моменту вы, наверное, начали осознавать, насколько полезными могут быть декораторы. Они расширяют возможности функции без редактирования её кода и являются гибким инструментом для изменения чего угодно.

?????????? ????????? ? ?????????? ????????

В приведённых выше примерах декораторы ничего не принимали и не возвращали. Модифицируем наш декоратор для измерения времени выполнения:

```
def benchmark(func):
    import time

    def wrapper(*args, **kwargs):
        start = time.time()
        return_value = func(*args, **kwargs)
        end = time.time()
        print('[*] Время выполнения: {} секунд.'.format(end-start))
        return return_value
    return wrapper

@benchmark
```

```
def fetch_webpage(url):
    import requests
    webpage = requests.get(url)
    return webpage.text

webpage = fetch_webpage('https://google.com')
print(webpage)
```

Вывод после выполнения:

```
[*] Время выполнения: 1.4475083351135254 секунд.
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"........
```

Как вы видите, аргументы декорируемой функции передаются функции-обёртке, после чего с ними можно делать что угодно. Можно изменять аргументы и затем передавать их декорируемой функции, а можно оставить их как есть или вовсе забыть про них и передать что-нибудь совсем другое. То же касается возвращаемого из декорируемой функции значения, с ним тоже можно делать что угодно.

?????????? ? ???????????

Мы также можем создавать декораторы, которые принимают аргументы. Посмотрим на пример:

```
def benchmark(iters):
    def actual_decorator(func):
        import time

        def wrapper(*args, **kwargs):
            total = 0
            for i in range(iters):
                start = time.time()
                return_value = func(*args, **kwargs)
                end = time.time()
                total = total + (end-start)
            print('[*] Среднее время выполнения: {} секунд.'.format(total/iters))
            return return_value

        return wrapper
    return actual_decorator
```

```
@benchmark(iters=10)
def fetch_webpage(url):
    import requests
    webpage = requests.get(url)
    return webpage.text

webpage = fetch_webpage('https://google.com')
print(webpage)
```

Здесь мы модифицировали наш старый декоратор таким образом, чтобы он выполнял декорируемую функцию `iters` раз, а затем выводил среднее время выполнения. Однако чтобы добиться этого, пришлось воспользоваться природой функций в Python.

Функция `benchmark()` на первый взгляд может показаться декоратором, но на самом деле таковым не является. Это обычная функция, которая принимает аргумент `iters`, а затем возвращает декоратор. В свою очередь, он декорирует функцию `fetch_webpage()`. Поэтому мы использовали не выражение `@benchmark`, а `@benchmark(iters=10)` — это означает, что тут вызывается функция `benchmark()` (функция со скобками после неё обозначает вызов функции), после чего она возвращает сам декоратор.

Да, это может быть действительно сложно уместить в голове, поэтому держите правило:

> Декоратор принимает функцию в качестве аргумента и возвращает функцию.

В нашем примере `benchmark()` не удовлетворяет этому условию, так как она не принимает функцию в качестве аргумента. В то время как функция `actual_decorator()`, которая возвращается `benchmark()`, является декоратором.

???????-???????????

Напоследок стоит упомянуть, что не только функции, а любые вызываемые объекты могут быть декоратором. Экземпляры классов/объекты с методом `__call__()` тоже можно вызывать, поэтому их можно использовать в качестве декораторов. Эту функциональность можно использовать для создания декораторов, хранящих какое-то состояние. Например, вот декоратор для мемоизации:

```
from collections import deque


class Memoized:
```

```
    def __init__(self, cache_size=100):
        self.cache_size = cache_size
        self.call_args_queue = deque()
        self.call_args_to_result = {}

    def __call__(self, fn):
        def new_func(*args, **kwargs):
            memoization_key = self._convert_call_arguments_to_hash(args, kwargs)
            if memoization_key not in self.call_args_to_result:
                result = fn(*args, **kwargs)
                self._update_cache_key_with_value(memoization_key, result)
                self._evict_cache_if_necessary()
            return self.call_args_to_result[memoization_key]
        return new_func

    def _update_cache_key_with_value(self, key, value):
        self.call_args_to_result[key] = value
        self.call_args_queue.append(key)

    def _evict_cache_if_necessary(self):
        if len(self.call_args_queue) > self.cache_size:
            oldest_key = self.call_args_queue.popleft()
            del self.call_args_to_result[oldest_key]

    @staticmethod
    def _convert_call_arguments_to_hash(args, kwargs):
        return hash(str(args) + str(kwargs))


@Memoized(cache_size=5)
def get_not_so_random_number_with_max(max_value):
    import random
    return random.random() * max_value
```

Само собой, этот декоратор нужен в основном в демонстрационных целях, в реальном приложении для подобного кеширования стоит использовать `functools.lru_cache`.

??????????

Тут будут перечислены некоторые важные вещи, которые не были затронуты в статье или были затронуты вскользь. Вам может показаться, что они расходятся с тем, что было написано в статье до этого, но на самом деле это не так.

- Декораторы не обязательно должны быть функциями, это может быть любой вызываемый объект.
- Декораторы не обязаны возвращать функции, они могут возвращать что угодно. Но обычно мы хотим, чтобы декоратор вернул объект того же типа, что и декорируемый объект. Пример:

```
>>> def decorator(func):
...     return 'sumit'
...
>>> @decorator
... def hello_world():
...     print('hello world')
...
>>> hello_world
'sumit'
```

- Также декораторы могут принимать в качестве аргументов не только функции.

  Здесь можно почитать об этом подробнее.
- Необходимость в декораторах может быть не очевидной до написания библиотеки. Поэтому, если декораторы кажутся вам бесполезными, посмотрите на них с точки зрения разработчика библиотеки. Хорошим примером является декоратор представления в Flask.
- Также стоит обратить внимание на `functools.wraps()` — функцию, которая помогает сделать декорируемую функцию похожей на исходную, делая такие вещи, как сохранение doctstring исходной функции.

?????????

1. Декораторы в Python: понять и полюбить

???????

> **Функция** — это группа связанных инструкций, выполняющих определенную задачу.

Функции помогают разбить нашу программу на более мелкие части. По мере того, как наша программа становится все больше и больше, функции делают ее более организованной и управляемой.

Кроме того, функцию можно вызвать из различных мест программы, что позволяет избежать повторения программного кода.

????????? ???????

```
def имя_функции(аргументы):
    """строка документации"""
    операторы
```

Давайте рассмотрим какие компоненты включает в себя определение функции:

1. Ключевое слово `def` — начало заголовка функции.
2. Имя функции — для однозначной идентификации функции. Оно соответствует правилам называния идентификаторов в Python.
3. С помощью параметров (аргументов) мы передаем значения в функцию. Аргументов может и не быть.
4. Двоеточие `:` обозначает конец заголовка функции.
5. Необязательная строка документации (docstring) нужна для описания того, что делает функция.
6. Один или несколько операторов Python составляют тело функции. Все инструкции должны иметь одинаковый отступ (4 пробела или 1 TAB).
7. Оператор `return` возвращает переданное значение из функции. Он необязателен.

?????? ???????

```
def greet(name):
    """
    Эта функция
    приветствует человека, имя которого
```

```
    хранится в параметре name.
    """
    print("Привет, " + name + ". Доброе утро!")
```

??? ??????? ???????

После того, как мы определили функцию, мы можем вызвать ее в программе или даже из командной строки Python. Чтобы вызвать функцию, мы просто вводим ее имя с соответствующими параметрами.

```
>>> greet('Джон')
Привет, Джон. Доброе утро!
```

> **Примечание**. Попробуйте сами запустить приведенный выше код с определением функции и посмотрите результат.

```
def greet(name):
    """
    Эта функция
    приветствует человека, имя которого
    хранится в параметре name.
    """
    print("Привет, " + name + ". Доброе утро!")

greet('Джон')
```

?????? ???????????

Первая строка после заголовка функции называется строкой документации, она описывает, что делает функция.

Документирование кода — не обязательная, но очень хорошая практика. Если вы не помните, что ели на ужин на прошлой неделе, всегда документируйте свой код.

В приведенном выше примере у нас есть строка документации сразу под заголовком функции. Обычно используют тройные кавычки, чтобы документация могла занимать несколько строк. Получить доступ к строке документации можно через атрибут `__doc__`.

?????? ?????? ???????????

Попробуйте запустить в оболочке Python следующую команду и посмотрите на результат.

```
>>> print(greet.__doc__)

    Эта функция
    приветствует человека, имя которого
    хранится в параметре name.
```

????????????? ????????

Оператор `return` используется для выхода из функции и возврата в то место, откуда она была вызвана.

????????? ??????????? ????????

```
return список_выражений
```

Оператор `return` может содержать выражение, которое возвращает значение. Если в операторе нет выражения или самого оператора возврата нет внутри функции, функция вернет объект `None`.

?????? ??????????? ????????

```
def absolute_value(num):
    """ Возвращает абсолютное значение
   введенного числа"""

    if num >= 0:
        return num
    else:
        return -num

print(absolute_value(2))

print(absolute_value(-4))
```

Вывод:

```
2
4
```

??? ???????? ???????

Рассмотрите схему. Так работают функции.

g-q5RCoTbnblJyXH8IJQrP2-0zTkQH8_

??????? ????????? ? ????? ????? ???????????

Область видимости переменной — это часть программы, в которой к данной переменной можно обращаться. Параметры и переменные, определенные в теле функции, доступны только внутри нее. Это значит, что они находятся в локальной области видимости.

Время жизни переменной — это период, в течение которого переменная находится в памяти. Время жизни переменной внутри функции длится до тех пор, пока функция выполняется. Переменные «уничтожаются», когда мы выходим из функции. Это значит, что функция не помнит значения переменных из предыдущих вызовов.

Вот пример, иллюстрирующий область видимости переменной внутри функции.

```
def my_func():
    x = 10
    print("Значение внутри функции:", x)

x = 20
my_func()
print("Значение вне функции:", x)
```

Вывод:

```
Значение внутри функции: 10
Значение вне функции: 20
```

Здесь мы видим, что значение `x` изначально равно 20. Хотя функция `my_func()` изменила значение `x` на 10, это не повлияло на ее значение вне функции.

Так происходит потому, что локальная переменная `x` внутри функции отличается от глобальной переменной `x`, которая находится вне функции. Хотя у них одинаковые имена, это две разные переменные с разной областью видимости.

А вот переменные, которые объявлены вне функции, будут видны внутри нее. У них глобальная область видимости.

Если переменная определена вне функции, то внутри функции мы можем прочитать ее значение. Однако мы не сможем его изменить. Для того, чтобы изменить ее значение, нужно объявить ее как глобальную с помощью ключевого слова `global`.

???? ???????

Функции в Python можно разделить на два типа:

1. **Встроенные функции** — функции, предоставляемые языком программирования.
2. **Пользовательские функции** — функции, описанные пользователем в программном коде.

????????? ?? ?????????? ???????

Python допускает значения по умолчанию для параметров функции. Если вызывающий абонент не передает параметр, используется значение по умолчанию.

```
def hello(year=2019):
  print(f'Hello World {year}')
hello(2020) # function parameter is passed
hello() # function parameter is not passed, so default value will be used
```

Вывод:

```
Hello World 2020
Hello World 2019
```

?????????? ??????????? ???????? ?????? ???????

Функция может иметь несколько операторов возврата. Однако при достижении одного из операторов возврата выполнение функции прекращается, и значение возвращается вызывающей стороне.

```
def odd_even_checker(i):
  if i % 2 == 0:
    return 'even'
  else:
    return 'odd'

print(odd_even_checker(20))
print(odd_even_checker(15))
```

??????? ??????????? ????????

Функция Python может возвращать несколько значений одно за другим. Это реализовано с использованием ключевого слова `yield`. Это полезно, когда вы хотите, чтобы функция возвращала большое количество значений и обрабатывала их. Мы можем разделить возвращаемые значения на несколько частей, используя оператор `yield`. Этот тип функции также называется функцией генератора.

```
def return_odd_ints(i):
  x = 1
  while x <= i:
    yield x
    x += 2

output = return_odd_ints(10)
for out in output:
  print(out)
```

Вывод:

```
1
3
5
7
9
```

?????????

Python допускает три типа параметров в определении функции:

- Формальные аргументы: те, которые мы видели в примерах до сих пор.
- Переменное количество аргументов без ключевых слов: например, def add(*args)
- Переменное количество аргументов ключевых слов или именованных аргументов: например, def add(**kwargs)

Некоторые важные моменты относительно переменных аргументов в Python:

- Порядок аргументов должен быть формальным, * args и ** kwargs.
- Не обязательно использовать имена параметров переменных как args и kwargs. Однако лучше всего использовать их для лучшей читаемости кода.
- Тип args — кортеж. Таким образом, мы можем передать кортеж для отображения с переменной * args.
- Тип kwargs — словарь. Таким образом, мы можем передать словарь для сопоставления с переменной ** kwargs.

```
def add(x, y, *args, **kwargs):
  sum = x + y
  for a in args:
    sum += a

  for k, v in kwargs.items():
    sum += v
  return sum

total = add(1, 2, *(3, 4), **{"k1": 5, "k2": 6})
print(total) # 21
```

??????????? ???????

Когда функция вызывает сама себя, она называется рекурсивной функцией. В программировании этот сценарий называется рекурсией.

Вы должны быть очень осторожны при использовании рекурсии, потому что есть вероятность, что функция никогда не завершится и перейдет в бесконечный цикл. Вот простой пример печати ряда Фибоначчи с использованием рекурсии.

```
def fibonacci_numbers_at_index(count):
  if count <= 1:
    return count
  else:
    return fibonacci_numbers_at_index(count - 1) + fibonacci_numbers_at_index(count - 2)

count = 5
i = 1
while i <= count:
  print(fibonacci_numbers_at_index(i))
  i += 1
```

Полезно знать о рекурсии, но в большинстве случаев при программировании это не нужно. То же самое можно сделать с помощью цикла `for` или `while`.

???? ?????? ???????

Функции Python являются экземплярами класса «функция». Мы можем проверить это с помощью функции `type()`.

```
def foo():
  pass

print(type(foo)) # <class 'function'>
```

????????? ??????? ? ???????

- Функция Python является частью файла сценария Python, в котором она определена, тогда как методы определены внутри определения класса.
- Мы можем вызвать функцию напрямую, если она находится в том же модуле. Если функция определена в другом модуле, мы можем импортировать модуль, а затем вызвать функцию напрямую. Нам нужен класс или объект класса для вызова методов.
- Функция Python может обращаться ко всем глобальным переменным, тогда как методы класса Python могут обращаться к глобальным переменным, а также к атрибутам и функциям класса.
- Тип данных функций Python — это «функция», а тип данных методов Python — «метод».

```
class Data:
  def foo(self):
    print('foo method')

  def foo():
    print('foo function')

# calling a function
foo()

# calling a method
d = Data()
d.foo()

# checking data types
print(type(foo)) # <class 'function'>
print(type(d.foo)) # <class 'method'>
```

Преимущества:

- Возможность повторного использования кода, потому что мы можем вызывать одну и ту же функцию несколько раз.
- Модульный код, поскольку мы можем определять разные функции для разных задач.
- Улучшает ремонтопригодность кода.
- Абстракция, поскольку вызывающему абоненту не нужно знать реализацию функции.

????????? ???????

Анонимные функции не имеют имени. Мы можем определить анонимную функцию в Python, используя ключевое слово `lambda`.

```
def square(x):
  return x * x

f_square = lambda x: x * x

print(square(10)) # 100
```

```
print(f_square(10)) # 100
```

?????????

1. Функции в Python
2. Функции Python
3. Функции
4. Возвращаемый тип функции

??????

?????????? ???????? ?? ?????????

Присвоение значение по умолчанию в python делается через оператор `or`.

```
other = s or "some default value"
```

Примеры:

```
42    or "something"    # returns 42
0     or "something"    # returns "something"
None  or "something"    # returns "something"
False or "something"    # returns "something"
""    or "something"    # returns "something"
```

???????? ????????????????
???????????????? ?? ?????
Python

Объектно ориентированное программирование на языке Python

???????? ? ????????????? ????????? ???????

Узнайте, как создавать и использовать вложенные классы в Python для повышения читаемости и организации кода в нашей увлекательной статье!

Nested classes in Python represented through cubes.

Вложенные классы, также известные как внутренние классы, являются классами, определенными внутри других классов. В Python, вложенные классы могут быть использованы для повышения читаемости и организации кода, а также для создания более сложных структур данных. В этой статье мы рассмотрим, как создавать и использовать вложенные классы в Python.

???????? ?????????? ??????

Чтобы создать вложенный класс, просто определите класс внутри другого класса. Вот пример:

```
class OuterClass:
    class InnerClass:
        pass
```

Здесь `InnerClass` является вложенным классом, определенным внутри `OuterClass`.

????????????? ?????????? ??????

Вложенные классы могут быть использованы как любые другие классы. Для того чтобы получить доступ к внутреннему классу из внешнего, необходимо использовать имя внешнего класса вместе с именем внутреннего класса. Вот пример:

```
# Создаем объект внутреннего класса
inner_object = OuterClass.InnerClass()

# Используем методы и свойства внутреннего класса
```

```
inner_object.some_method()
```

Также можно создать объект внутреннего класса внутри внешнего класса:

```
class OuterClass:
    class InnerClass:
        def say_hello(self):
            return "Привет из внутреннего класса!"

    def create_inner_object(self):
        inner_object = self.InnerClass()
        return inner_object

outer_object = OuterClass()
inner_object = outer_object.create_inner_object()
print(inner_object.say_hello())  # Вывод: Привет из внутреннего класса!
```

?????? ?????????? ??????

Давайте рассмотрим пример использования вложенных классов для создания иерархии объектов. Предположим, у нас есть класс `Department`, который содержит классы `Employee` и `Manager`.

```
class Department:
    class Employee:
        def __init__(self, name, position):
            self.name = name
            self.position = position

    class Manager:
        def __init__(self, name, position):
            self.name = name
            self.position = position

        def manage_employee(self, employee):
            print(f"{self.name} ({self.position}) управляет {employee.name}
({employee.position})")

# Создаем объекты Employee и Manager
```

```
employee = Department.Employee("Алексей", "разработчик")
manager = Department.Manager("Ольга", "руководитель проекта")

# Метод manage_employee используется для отображения информации о связи между менеджером и
сотрудником
manager.manage_employee(employee)
```

Этот пример показывает, как использовать вложенные классы для создания иерархии объектов и организации кода.

□□ В заключение, вложенные классы могут быть полезны для улучшения структуры и читаемости вашего кода на Python. Они позволяют создавать сложные структуры данных, сохраняя при этом чистоту и организацию кода.

?????? ? ???????

???????? ??????? ? ????????

Создание класса в *Python* начинается с инструкции *class*. Вот так будет выглядеть минимальный класс.

```
class C:
    pass
```

Класс состоит из объявления (инструкция *class*), имени класса (нашем случае это имя *C*) и тела класса, которое содержит атрибуты и методы (в нашем минимальном классе есть только одна инструкция *pass*).

Для того чтобы создать объект класса необходимо воспользоваться следующим синтаксисом:

имя_объекта = имя_класса()

??????????? ? ???????????? ???????? ??????

Как уже было сказано выше, 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. Пример:

```
class Rectangle:
    default_color = "green"

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

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* .

```
>>> Rectangle.default_color
'green'
```

width и height – это динамические атрибуты, при их создании было использовано ключевое слово self. Пока просто примите это как должное, более подробно про self будет рассказано ниже. Для доступа к width и height предварительно нужно создать объект класса Rectangle:

```
>>> rect = Rectangle(10, 20)
>>> rect.width
10
>>> rect.height
20
```

Если обратиться через класс, то получим ошибку:

```
>>> Rectangle.width
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Rectangle' has no attribute 'width'
```

При этом, если вы обратитесь к статическому атрибуту через экземпляр класса, то все будет OK, до тех пор, пока вы не попытаетесь его поменять.

Проверим ещё раз значение атрибута default_color:

```
>>> Rectangle.default_color
'green'
```

Присвоим ему новое значение:

```
>>> Rectangle.default_color = "red"
>>> Rectangle.default_color
'red'
```

Создадим два объекта класса Rectangle и проверим, что default_color у них совпадает:

```
>>> r1 = Rectangle(1,2)
>>> r2 = Rectangle(10, 20)
>>> r1.default_color
'red'
>>> r2.default_color
'red'
```

Если поменять значение default_color через имя класса Rectangle, то все будет ожидаемо: у объектов r1 и r2 это значение изменится, но если поменять его через экземпляр класса, то у экземпляра будет создан атрибут с таким же именем как статический, а доступ к последнему будет потерян:

Меняем default_color через r1:

```
>>> r1.default_color = "blue"
>>> r1.default_color
'blue'
```

При этом у r2 остается значение статического атрибута:

```
>>> r2.default_color
'red'
>>> Rectangle.default_color
'red'
```

Вообще напрямую работать с атрибутами – не очень хорошая идея, лучше для этого использовать свойства.

?????? ??????

Добавим к нашему классу метод. Метод – это функция, находящаяся внутри класса и выполняющая определенную работу.

Методы бывают статическими, классовыми (среднее между статическими и обычными) и уровня класса (будем их называть просто словом метод). Статический метод создается с декоратором @staticmethod, классовый – с декоратором @classmethod, первым аргументом в него передается cls, обычный метод создается без специального декоратора, ему первым аргументом передается self:

```
class MyClass:
    @staticmethod
    def ex_static_method():
        print("static method")
    @classmethod
    def ex_class_method(cls):
        print("class method")
    def ex_method(self):
        print("method")
```

Статический и классовый метод можно вызвать, не создавая экземпляр класса, для вызова ex_method() нужен объект:

```
>>> MyClass.ex_static_method()
static method

>>> MyClass.ex_class_method()
class method

>>> MyClass.ex_method()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ex_method() missing 1 required positional argument: 'self'

>>> m = MyClass()
>>> m.ex_method()
method
```

??????????? ?????? ? ????????????? ??????????? ??????

В Python разделяют конструктор класса и метод для инициализации экземпляра класса. Конструктор класса это метод *new*(cls, *args, **kwargs) для инициализации экземпляра класса используется метод *init*(self). При этом, как вы могли заметить *new* – это классовый метод, а *init* таким не является. Метод *new* редко переопределяется, чаще используется реализация от базового класса object (см. раздел Наследование), *init* же наоборот является очень удобным способом задать параметры объекта при его создании.

Создадим реализацию класса Rectangle с измененным конструктором и инициализатором, через который задается ширина и высота прямоугольника:

```
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

>>> rect = Rectangle(10, 20)
Hello from __new__
Hello from __init__

>>> rect.width
10

>>> rect.height
20
```

??? ????? self?

До этого момента вы уже успели познакомиться с ключевым словом self. self – это ссылка на текущий экземпляр класса, в таких языках как Java, C# аналогом является ключевое слово this. Через self вы получаете доступ к атрибутам и методам класса внутри него:

```
class Rectangle:

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

В приведенной реализации метод area получает доступ к атрибутам width и height для расчета площади. Если бы в качестве первого параметра не было указано self, то при попытке вызвать area программа была бы остановлена с ошибкой.

?????? ??????? ???????? ? ??????

Если вы знакомы с языками программирования Java, C#, C++ то, наверное, уже задались вопросом: “а как управлять уровнем доступа?”. В перечисленных языках вы можете явно указать для переменной, что доступ к ней снаружи класса запрещен, это делается с помощью ключевых слов (private, protected и т.д.). В Python таких возможностей нет, и любой

может обратиться к атрибутам и методам вашего класса, если возникнет такая необходимость. Это существенный недостаток этого языка, т.к. нарушается один из ключевых принципов ООП – инкапсуляция. Хорошим тоном считается, что для чтения/изменения/удаления какого-то атрибута должны использоваться специальные методы, которые называются **getter/setter/deleter**, их можно реализовать, но ничего не помешает изменить атрибут напрямую. При этом есть соглашение, что метод или атрибут, который начинается с нижнего подчеркивания, является скрытым, и снаружи класса трогать его не нужно (хотя сделать это можно).

Внесем соответствующие изменения в класс Rectangle:

```
class Rectangle:

    def __init__(self, width, height):
        self._width = width
        self._height = height

    def get_width(self):
        return self._width

    def set_width(self, w):
        self._width = w

    def get_height(self):
        return self._height

    def set_height(self, h):
        self._height = h

    def area(self):
        return self._width * self._height
```

В приведенном примере для доступа к _width и _height используются специальные методы, но ничего не мешает вам обратиться к ним (атрибутам) напрямую.

```
>>> rect = Rectangle(10, 20)
>>> rect.get_width()
10
>>> rect._width
10
```

Если же атрибут или метод начинается с двух подчеркиваний, то тут напрямую вы к нему уже не обратитесь (простым образом). Модифицируем наш класс Rectangle:

```
class Rectangle:

    def __init__(self, width, height):
        self.__width = width
        self.__height = height

    def get_width(self):
        return self.__width

    def set_width(self, w):
        self.__width = w

    def get_height(self):
        return self.__height

    def set_height(self, h):
        self.__height = h

    def area(self):
        return self.__width * self.__height
```

Попытка обратиться к __width напрямую вызовет ошибку, нужно работать только через get_width():

```
>>> rect = Rectangle(10, 20)

>>> rect.__width
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Rectangle' object has no attribute '__width'

>>> rect.get_width()
10
```

Попытка обратиться к __width напрямую вызовет ошибку, нужно работать только через get_width():

```
>>> rect = Rectangle(10, 20)

>>> rect.__width
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Rectangle' object has no attribute '__width'

>>> rect.get_width()
10
```

Но на самом деле это сделать можно, просто этот атрибут теперь для внешнего использования носит название: *Rectangle*_width:

```
>>> rect._Rectangle__width
10

>>> rect._Rectangle__width = 20

>>> rect.get_width()
20
```

????????

Свойством называется такой метод класса, работа с которым подобна работе с атрибутом. Для объявления метода свойством необходимо использовать декоратор @property.

Важным преимуществом работы через свойства является то, что вы можете осуществлять проверку входных значений, перед тем как присвоить их атрибутам.

Сделаем реализацию класса Rectangle с использованием свойств:

```
class Rectangle:

    def __init__(self, width, height):
        self.__width = width
        self.__height = height

    @property
    def width(self):
        return self.__width
```

```
    @width.setter
    def width(self, w):
        if w > 0:
            self.__width = w
        else:
            raise ValueError

    @property
    def height(self):
        return self.__height

    @height.setter
    def height(self, h):
        if h > 0:
            self.__height = h
        else:
            raise ValueError

    @height.deleter
    def height(self):
      del self.__height

    def area(self):
        return self.__width * self.__height
```

Теперь работать с **width** и **height** можно так, как будто они являются атрибутами:

```
>>> rect = Rectangle(10, 20)

>>> rect.width
10

>>> rect.height
20
```

Можно не только читать, но и задавать новые значения свойствам:

```
>>> rect.width = 50
```

```
>>> rect.width
50

>>> rect.height = 70

>>> rect.height
70
```

Если вы обратили внимание: в setter’ах этих свойств осуществляется проверка входных значений, если значение меньше нуля, то будет выброшено исключение `ValueError`:

```
>>> rect.width = -10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "test.py", line 28, in width
    raise ValueError
ValueError
```

??????????? ??????? ? ????????? ???????

Доступ к атрибутам объекта можно также осуществлять с помощью встроенных функции `getattr`, `setattr` и `delattr` извне так и внутри объекта с помощью встроенных методов `__getattr__`, `__setattr__` и `__delattr__`.

???????????

В организации наследования участвуют как минимум два класса: класс родитель и класс потомок. При этом возможно множественное наследование, в этом случае у класса потомка может быть несколько родителей. Не все языки программирования поддерживают множественное наследование, но в Python можно его использовать. По умолчанию все классы в Python являются наследниками от object, явно этот факт указывать не нужно.

Синтаксически создание класса с указанием его родителя выглядит так:

class имя_класса(имя_родителя1, [имя_родителя2,..., имя_родителя_n])

Переработаем наш пример так, чтобы в нем присутствовало наследование:

```
class Figure:

    def __init__(self, color):
        self.__color = color

    @property
    def color(self):
        return self.__color

    @color.setter
    def color(self, c):
        self.__color = c

class Rectangle(Figure):

    def __init__(self, width, height, color):
        super().__init__(color)
        self.__width = width
        self.__height = height

    @property
    def width(self):
        return self.__width

    @width.setter
    def width(self, w):
        if w > 0:
            self.__width = w
        else:
            raise ValueError

    @property
    def height(self):
        return self.__height

    @height.setter
    def height(self, h):
        if h > 0:
            self.__height = h
        else:
```

```
            raise ValueError

    def area(self):
        return self.__width * self.__height
```

Родительским классом является Figure, который при инициализации принимает цвет фигуры и предоставляет его через свойства. Rectangle – класс наследник от Figure. Обратите внимание на его метод *init*: в нем первым делом вызывается конструктор (хотя это не совсем верно, но будем говорить так) его родительского класса:

*super().*_init__(color)_

super – это ключевое слово, которое используется для обращения к родительскому классу.

Теперь у объекта класса Rectangle помимо уже знакомых свойств width и height появилось свойство color:

```
>>> rect = Rectangle(10, 20, "green")

>>> rect.width
10

>>> rect.height
20

>>> rect.color
'green'

>>> rect.color = "red"

>>> rect.color
'red'
```

??????????

Как уже было сказано во введении в рамках ООП полиморфизм, как правило, используется с позиции переопределения методов базового класса в классе наследнике. Проще всего это рассмотреть на примере. Добавим в наш базовый класс метод info(), который печатает сводную информацию по объекту класса Figure и переопределим этот метод в классе Rectangle, добавим в него дополнительные данные:

```
class Figure:

    def __init__(self, color):
        self.__color = color

    @property
    def color(self):
        return self.__color

    @color.setter
    def color(self, c):
        self.__color = c

    def info(self):
       print("Figure")
       print("Color: " + self.__color)

class Rectangle(Figure):

    def __init__(self, width, height, color):
        super().__init__(color)
        self.__width = width
        self.__height = height

    @property
    def width(self):
        return self.__width

    @width.setter
    def width(self, w):
        if w > 0:
            self.__width = w
        else:
            raise ValueError

    @property
    def height(self):
        return self.__height

    @height.setter
```

```
    def height(self, h):
        if h > 0:
            self.__height = h
        else:
            raise ValueError

    def info(self):
        print("Rectangle")
        print("Color: " + self.color)
        print("Width: " + str(self.width))
        print("Height: " + str(self.height))
        print("Area: " + str(self.area()))

    def area(self):
        return self.__width * self.__height
```

Посмотрим, как это работает

```
>>> fig = Figure("orange")

>>> fig.info()
Figure
Color: orange

>>> rect = Rectangle(10, 20, "green")

>>> rect.info()
Rectangle
Color: green
Width: 10
Height: 20
Area: 200
```

Таким образом, класс наследник может расширять функционал класса родителя.

?????????

1. Python. Урок 14. Классы и объекты

2. Python ООП
3. Built-in Functions
4. Customizing attribute access

?????????????? ??????????

1. Объектная модель Python. Классы, поля и методы
2. Волшебные методы, переопределение методов. Наследование
3. Настройка доступа к атрибутам класса Python
4. Python ООП

???????? ???????? ????????-
??????????????? ???????????????

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. Каждый объект является экземпляром некоторого класса. Классы образуют иерархии. Более подробно о понятии ООП можно прочитать на википедии.

Выделяют три основных “столпа” ООП- это инкапсуляция, наследование и полиморфизм.

??????????

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. Можно создавать переменные этого нового типа, такие переменные называются объекты.

??????????

Под наследованием понимается возможность создания нового класса на базе существующего. Наследование предполагает наличие отношения “является” между классом наследником и классом родителем. При этом класс потомок будет содержать те же атрибуты и методы, что и базовый класс, но при этом его можно (и нужно) расширять через добавление новых методов и атрибутов.

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..

??????????

Полиморфизм позволяет одинаково обращаться с объектами, имеющими однотипный интерфейс, независимо от внутренней реализации объекта. Например, с объектом класса “грузовой автомобиль” можно производить те же операции, что и с объектом класса “автомобиль”, т.к. первый является наследником второго, при этом обратное утверждение неверно (во всяком случае не всегда). Другими словами полиморфизм предполагает разную реализацию методов с одинаковыми именами. Это очень полезно при наследовании, когда в классе наследнике можно переопределить методы класса родителя.

?????? ? ??????

?????? ????????????? ???????????

- Модули и пакеты в Python. Импорт. Виртуальная среда venv.

Copy in Python (Deep Copy and Shallow Copy)

In Python, Assignment statements do not copy objects, they create bindings between a target and an object. When we use the = operator, It only creates a new variable that shares the reference of the original object. In order to create “real copies” or “clones” of these objects, we can use the copy module in Python.

Syntax of Python Deepcopy

> **Syntax:** copy.deepcopy(x)

Syntax of Python Shallowcopy

> **Syntax:** copy.copy(x)

Example:

In order to make these copies, we use the copy module. The copy() returns a shallow copy of the list, and deepcopy() returns a deep copy of the list. As you can see that both have the same value but have different IDs.

Example: This code showcases the usage of the `copy` module to create both shallow and deep copies of a nested list `li1`. A shallow copy, `li2`, is created using **`copy.copy()`**, preserving the top-level structure but sharing references to the inner lists. A deep copy, `li3`, is created using **`copy.deepcopy()`**, resulting in a completely independent copy of `li1`, including all nested elements. The code prints the IDs and values of `li2` and `li3`, highlighting the distinction between shallow and deep copies in terms of reference and independence.

```
import copy
li1 = [1, 2, [3, 5], 4]
li2 = copy.copy(li1)
print("li2 ID: ", id(li2), "Value: ", li2)
li3 = copy.deepcopy(li1)
```

```
print("li3 ID: ", id(li3), "Value: ", li3)
```

Output:

```
li2 ID:  2521878674624 Value:  [1, 2, [3, 5], 4]
li3 ID:  2521878676160 Value:  [1, 2, [3, 5], 4]
```

What is Deep copy in Python?

A deep copy creates a new compound object before inserting copies of the items found in the original into it in a recursive manner. It means first constructing a new collection object and then recursively populating it with copies of the child objects found in the original. In the case of deep copy, a copy of the object is copied into another object. It means that **any changes** made to a copy of the object **do not reflect in the original object.**

Deep copy in Python

Example:

In the above example, the change made in the list **did not affect** other lists, indicating the list is deeply copied.

This code illustrates deep copying of a list with nested elements using the `copy` module. It initially prints the original elements of `li1`, then deep copies them to create `li2`. A modification to an element in `li2` does not affect `li1`, as demonstrated by the separate printouts. This highlights how deep copying creates an independent copy, preserving the original list's contents even after changes to the copy.

```
import copy
li1 = [1, 2, [3,5], 4]
li2 = copy.deepcopy(li1)
print ("The original elements before deep copying")
for i in range(0,len(li1)):
    print (li1[i],end=" ")

print("\r")
li2[2][0] = 7
print ("The new list of elements after deep copying ")
for i in range(0,len( li1)):
    print (li2[i],end=" ")
```

```
print("\r")
print ("The original elements after deep copying")
for i in range(0,len( li1)):
    print (li1[i],end=" ")
```

Output:

```
The original elements before deep copying
1 2 [3, 5] 4
The new list of elements after deep copying
1 2 [7, 5] 4
The original elements after deep copying
1 2 [3, 5] 4
```

What is Shallow copy in Python?

A shallow copy creates a new compound object and then references the objects contained in the original within it, which means it constructs a new collection object and then populates it with references to the child objects found in the original. The copying process does not recurse and therefore won't create copies of the child objects themselves. In the case of shallow copy, a reference of an object is copied into another object. It means that **any changes** made to a copy of an object **do reflect** in the original object. In python, this is implemented using the "**copy()**" function.

Shallow copy in Python

Example:

In this example, the change made in the list **did affect** another list, indicating the list is shallowly copied. **Important Points:** The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Example: This code demonstrates shallow copying of a list with nested elements using the '`copy`' module. Initially, it prints the original elements of `li1`, then performs shallow copying into `li2`. Modifying an element in `li2` affects the corresponding element in `li1`, as both lists share references to the inner lists. This illustrates that shallow copying creates a new list, but it doesn't provide complete independence for nested elements.

```
import copy
li1 = [1, 2, [3,5], 4]
li2 = copy.copy(li1)
print ("The original elements before shallow copying")
for i in range(0,len(li1)):
    print (li1[i],end=" ")

print("\r")
li2[2][0] = 7
print ("The original elements after shallow copying")
for i in range(0,len( li1)):
    print (li1[i],end=" ")
```

Output:

```
The original elements before shallow copying
1 2 [3, 5] 4
The original elements after shallow copying
1 2 [7, 5] 4
```

?????????

- copy in Python (Deep Copy and Shallow Copy)
- Difference between Shallow and Deep copy of a class

????????

????????????

- Online JSON viewer and formatter
- Regular expressions(regexp)
- Online UUID Generator

?????????????? ? python

?????????? ? REST API ? ????????? ?? Python

Картинка к публикации: Разбираемся в REST API с примерами на Python

????????

REST, или Representational State Transfer, представляет собой архитектурный стиль для разработки сетевых приложений. Этот стиль был предложен Роем Филдингом в его докторской диссертации в 2000 году и стал основой для многих современных веб-сервисов. Давайте разберемся, что такое REST, как он работает и почему его использование столь важно.

REST - это набор принципов и ограничений, которые определяют, как клиенты и серверы должны взаимодействовать друг с другом. Основная идея REST заключается в том, что веб-ресурсы (например, данные) представлены в виде уникальных URL-адресов, и клиенты могут выполнять операции над этими ресурсами с использованием стандартных HTTP методов.

REST определяет несколько ключевых принципов:

Идентификация ресурсов: Все данные представлены в виде ресурсов, которые имеют уникальные URL-адреса. Например, ресурсом может быть информация о пользователях, заказах, продуктах и так далее.

Унификация интерфейса: Взаимодействие с ресурсами осуществляется с помощью стандартных HTTP методов:

- GET: Получение данных.
- POST: Создание новых данных.
- PUT: Обновление данных.
- DELETE: Удаление данных.

Представление данных: Ресурсы могут быть представлены в разных форматах, таких как JSON или XML, и клиенты могут выбирать предпочтительный формат.

Без состояния: Взаимодействие между клиентом и сервером должно быть без состояния, что означает, что каждый запрос должен содержать всю необходимую информацию для

выполнения действия. Сервер не хранит состояния клиента между запросами.

Следование ограничениям: REST API должны следовать ограничениям, определенным в архитектурном стиле.

REST API предоставляет множество преимуществ:

1. **Простота**: REST API основан на стандартных принципах HTTP, что делает его простым для понимания и использования.
2. **Масштабируемость**: RESTful приложения могут масштабироваться горизонтально, что позволяет им обслуживать большое количество клиентов.
3. **Независимость от языка**: REST API могут быть использованы на разных платформах и с разными языками программирования.
4. **Гибкость**: Клиенты могут выбирать формат данных, который им удобен.

????????? HTTP

В HTTP (Hypertext Transfer Protocol) заголовки представляют собой метаданные, которые передаются вместе с запросами и ответами между клиентом и сервером. Заголовки содержат информацию о том, как запрос или ответ должны быть обработаны, а также могут содержать дополнительные сведения о данных, передаваемых через протокол. Рассмотрим некоторые наиболее распространенные заголовки HTTP и их назначение.

1. Заголовки запроса (Request Headers)

- **Host:** Этот заголовок указывает на имя хоста сервера, к которому отправляется запрос. Например, "Host: example.com".
- **User-Agent:** Заголовок User-Agent содержит информацию о браузере или клиентском приложении, отправляющем запрос. Это позволяет серверу определить, с какого устройства и браузера пришел запрос.
- **Authorization:** Заголовок Authorization используется для передачи информации об аутентификации, например, токена доступа или логина и пароля.
- **Accept:** Этот заголовок указывает, какие типы контента клиент готов принять от сервера, например, "Accept: application/json".
- **Content-Type:** Заголовок Content-Type сообщает серверу о типе контента, который передается в теле запроса. Это важно для правильной обработки данных на сервере.

2. Заголовки ответа (Response Headers)

- **Content-Type:** Заголовок Content-Type в ответе сервера указывает на тип данных, возвращаемых клиенту, например, "Content-Type: application/json".

- **Cache-Control:** Этот заголовок управляет кэшированием ресурсов на стороне клиента. Он может указывать, насколько долго ресурс должен быть кэширован, или что его не следует кэшировать вообще.
- **Location:** Заголовок Location используется для указания нового местоположения (URL) ресурса в случае перенаправления (код состояния 3xx). Это позволяет клиенту перейти по новому URL.
- **Access-Control-Allow-Origin:** Этот заголовок используется для управления политикой Same-Origin Policy и указывает, какие источники могут получать доступ к ресурсам на сервере.
- **Server:** Заголовок Server содержит информацию о сервере, который обрабатывает запрос. Это может быть полезно для отладки и мониторинга.

Заголовки HTTP играют важную роль в коммуникации между клиентами и серверами, обеспечивая не только передачу данных, но и управление процессом обработки запросов и ответов. Разработчики могут использовать различные заголовки для настройки и оптимизации взаимодействия между клиентами и серверами в RESTful API.

HTTP ??????

HTTP (Hypertext Transfer Protocol) - это протокол, используемый для передачи данных в сети, и он играет важную роль в веб-разработке. В контексте REST API существует несколько основных HTTP методов, которые определяют, как клиенты взаимодействуют с ресурсами. Подробнее рассмотрим каждый из них:

GET

HTTP метод `GET` используется для запроса данных с сервера. Когда клиент отправляет GET-запрос, сервер должен вернуть запрошенные данные. Этот метод не должен влиять на состояние сервера или данных, и его использование считается безопасным и идемпотентным, что означает, что многократные запросы GET не должны изменять результат.

Пример GET-запроса на Python с использованием библиотеки `requests`:

```
import requests


response = requests.get('https://example.com/api/resource')
data = response.json()  # Получаем данные в формате JSON
```

POST

Метод `POST` используется для создания новых данных на сервере. Когда клиент отправляет POST-запрос, он передает данные серверу, который затем создает новый ресурс с этими данными. Этот метод может изменять состояние сервера и не идемпотентен.

Пример POST-запроса на Python:

```
import requests


data = {'key': 'value'}
response = requests.post('https://example.com/api/resource', json=data)
```

PUT

HTTP метод `PUT` применяется для обновления существующих данных на сервере. При использовании PUT-запроса, клиент отправляет данные, которые должны заменить существующие данные на сервере. Этот метод также не идемпотентен.

Пример PUT-запроса на Python:

```
import requests


data = {'key': 'new_value'}
response = requests.put('https://example.com/api/resource/1', json=data)
```

PATCH

HTTP метод `PATCH` также используется для обновления существующих данных на сервере, но с одной важной разницей по сравнению с PUT. Вместо того, чтобы заменять все данные ресурса, как это делает PUT, метод PATCH позволяет клиенту отправлять только те части данных, которые требуется обновить. Это особенно полезно, когда нужно внести небольшие изменения в ресурс, не перезаписывая его полностью. Этот метод также не идемпотентен.

Пример PATCH-запроса на Python:

```
import requests


data = {'key': 'new_value'}
response = requests.patch('https://example.com/api/resource/1', json=data)
```

При использовании метода PATCH, сервер должен обновить только те поля ресурса, которые указаны в запросе, и оставить остальные данные нетронутыми. Это позволяет более эффективно управлять изменениями на сервере, особенно в случаях, когда ресурсы могут

быть большими и изменения касаются только небольшой их части.

DELETE

Метод `DELETE` используется для удаления ресурса на сервере. Когда клиент отправляет DELETE-запрос, сервер должен удалить указанный ресурс. Этот метод, как и POST и PUT, изменяет состояние сервера и не является идемпотентным.

Пример DELETE-запроса на Python:

```
import requests


response = requests.delete('https://example.com/api/resource/1')
```

Эти HTTP методы обеспечивают базовый функционал для взаимодействия с RESTful API. С их помощью клиенты могут получать данные, создавать новые ресурсы, обновлять и удалять существующие данные, что делает REST API гибким и мощным инструментом для работы с данными в веб-приложениях.

??????? ? ??????????

В REST API ресурсы играют центральную роль. Они представляют собой сущности или данные, к которым клиенты могут обращаться с помощью HTTP методов. Рассмотрим, как определяются ресурсы и как устроена структура URL в REST API.

Определение ресурсов в REST API начинается с идентификации того, что вы хотите предоставить клиентам. Ресурс может быть чем угодно, от информации о пользователях и продуктах до комментариев и изображений. Важно выбрать набор ресурсов, которые логически разделяются и имеют смысл в контексте вашего приложения.

Каждый ресурс должен иметь уникальный идентификатор, который определяет его. Этот идентификатор часто представляется как часть URL-адреса. Например, если у вас есть ресурсы "пользователи" и "продукты", то их URL-адреса могли бы выглядеть следующим образом:

- Ресурс "пользователи": `https://example.com/api/users`
- Ресурс "продукты": `https://example.com/api/products`

Каждый ресурс также может иметь свои собственные подресурсы. Например, ресурс "пользователи" может иметь подресурсы, связанные с конкретным пользователем, такие как его заказы или профиль:

- Подресурс "заказы пользователя": `https://example.com/api/users/1/orders`
- Подресурс "профиль пользователя": `https://example.com/api/users/1/profile`

Структура URL в REST API обычно следует определенным соглашениям и паттернам, чтобы сделать ее понятной и легко читаемой. Обычно URL состоит из следующих компонентов:

- **Протокол**: Обычно `https://` для безопасной передачи данных.
- **Доменное имя**: Это имя вашего сервера, например, `example.com`.
- **Путь**: Путь к ресурсу или эндпойнту на сервере. Он определяет, какой ресурс или функцию вы хотите вызвать.
- **Параметры запроса**: Опциональные параметры, которые могут передаваться в запросе, например, фильтры или сортировка.
- **Фрагмент**: Опциональная часть URL, которая может использоваться на клиентской стороне.

Пример структуры URL для запроса к ресурсу "пользователи":

```
https://example.com/api/users
```

В этом URL `https://` - протокол, `example.com` - доменное имя, `/api/users` - путь к ресурсу "пользователи". Если бы вы хотели получить информацию о конкретном пользователе, вы могли бы добавить идентификатор пользователя к URL:

```
https://example.com/api/users/1
```

Этот URL указывает на ресурс "пользователь" с идентификатором 1. Клиент может использовать различные HTTP методы (GET, POST, PUT, PATCH, DELETE) для взаимодействия с этими URL-адресами и ресурсами.

Структура URL в REST API позволяет клиентам легко обращаться к различным ресурсам и подресурсам, делая взаимодействие с API более интуитивным и удобным.

??????? ? Python

Рассмотрим пример создания RESTful API с использованием Django. Для этого мы будем использовать Django REST framework, популярный инструмент для создания RESTful API на основе Django.

1. Создание RESTful сервера с использованием Django

Для начала, убедитесь, что у вас установлен Django и Django REST framework. Затем создайте новый проект Django и приложение:

```
$ django-admin startproject rest_api_project
$ cd rest_api_project
$ python manage.py startapp api
```

Затем добавьте `api` в `INSTALLED_APPS` в файле `settings.py` вашего проекта:

```
INSTALLED_APPS = [
    # ...
    'api',
    # ...
]
```

2. Определение ресурсов и их URL

Создайте модели Django, которые будут представлять ваши ресурсы. Например, давайте создадим модель для пользователей:

```
# api/models.py
from django.db import models

class User(models.Model):
    username = models.CharField(max_length=100)
    email = models.EmailField()
    # Добавьте другие поля по вашему усмотрению
```

Затем создайте сериализаторы Django REST framework, чтобы определить, как данные будут представлены в формате JSON:

```
# api/serializers.py
from rest_framework import serializers
from .models import User

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = '__all__'
```

3. Обработка HTTP методов для ресурсов

Создайте представления Django REST framework для обработки HTTP методов. Например, создайте представление для работы с пользователями:

```
# api/views.py
from rest_framework import viewsets
from .models import User
from .serializers import UserSerializer

class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

4. Отправка и прием данных через REST API

Настройте маршруты для ваших ресурсов в файле `urls.py`:

```
# api/urls.py
from rest_framework.routers import DefaultRouter
from .views import UserViewSet

router = DefaultRouter()
router.register(r'users', UserViewSet)

urlpatterns = [
    # Добавьте другие маршруты, если необходимо
]

urlpatterns += router.urls
```

Теперь ваш RESTful API готов к использованию. Вы можете создать, получать, обновлять и удалять пользователей с помощью HTTP методов. Для тестирования API, вы можете использовать инструменты, такие как `curl` или `Postman`, или написать Python скрипты, используя библиотеку `requests`.

Это всего лишь базовый пример создания RESTful API с использованием Django и Django REST framework. Вы можете добавить другие ресурсы, авторизацию, версионирование и многое другое, чтобы создать полноценное веб-приложение с RESTful API.

?????????????? ? ???????????

Аутентификация и авторизация играют важную роль в обеспечении безопасности RESTful API. Рассмотрим методы аутентификации и роли с разрешениями в контексте REST API.

Методы аутентификации используются для проверки подлинности клиентов, которые пытаются получить доступ к ресурсам API. В Django REST framework, существует несколько стандартных методов аутентификации:

- **Token Authentication**: Этот метод аутентификации использует токены для проверки подлинности клиента. Клиент должен предоставить действующий токен для каждого запроса. Это популярный метод для мобильных приложений и однопользовательских сценариев.
- **Session Authentication**: Этот метод аутентификации использует механизмы сессий браузера для проверки подлинности. Когда пользователь входит в систему, ему назначается сессия, и эта сессия сохраняется на стороне клиента. Он часто используется для веб-приложений.
- **Basic Authentication**: Basic Authentication требует от клиента предоставить имя пользователя и пароль при каждом запросе, закодированные в заголовке запроса. Этот метод не является безопасным, если не используется HTTPS.
- **OAuth**: OAuth - это протокол аутентификации и авторизации, который позволяет клиентам получать доступ к ресурсам от имени пользователя с его разрешения. Он часто используется в социальных сетях и сторонних приложениях.

Выбор метода аутентификации зависит от потребностей вашего приложения и уровня безопасности, который вам требуется.

В REST API роли и разрешения используются для определения, какие пользователи имеют доступ к каким ресурсам и какие действия они могут выполнять. Роли и разрешения часто настраиваются с использованием библиотеки Django REST framework.

Пример определения ролей и разрешений:

```
# api/models.py
from django.contrib.auth.models import User
from django.db import models

class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    is_admin = models.BooleanField(default=False)
    is_editor = models.BooleanField(default=False)
    # Другие поля и разрешения

# api/views.py
from rest_framework import viewsets, permissions
from .models import UserProfile
from .serializers import UserProfileSerializer
```

```
class UserProfileViewSet(viewsets.ModelViewSet):
    queryset = UserProfile.objects.all()
    serializer_class = UserProfileSerializer
    permission_classes = [permissions.IsAuthenticated, permissions.IsAdminUser]
```

В этом примере мы создаем модель `UserProfile`, которая связана с моделью `User`. Мы определяем две роли: администратор и редактор. Затем мы используем `permission_classes`, чтобы определить, что доступ к просмотру и изменению профилей `UserProfile` имеют только аутентифицированные пользователи и администраторы.

Вы можете определить свои собственные роли и разрешения в зависимости от требований вашего приложения. Роли и разрешения обеспечивают гибкость в управлении доступом к ресурсам в RESTful API.

????????? ??????

Обработка ошибок в REST API является важной частью разработки, так как она позволяет клиентам понимать, что пошло не так при выполнении запроса. В REST API ошибки обычно возвращаются клиентам в формате JSON, чтобы обеспечить четкое и информативное сообщение о произошедшей проблеме. Рассмотрим, как обрабатывать ошибки и какие стандартные коды состояния HTTP используются.

Определение исключений: Ваше REST API должно определять различные исключения для разных видов ошибок. Например, исключение для отсутствия ресурса, исключение для ошибки авторизации и так далее. Это помогает установить контроль над обработкой ошибок.

```
from rest_framework.exceptions import NotFound


def get_user(request, user_id):
    try:
        user = User.objects.get(id=user_id)
        # ...
    except User.DoesNotExist:
        raise NotFound("Пользователь не найден")
```

Генерация исключений: В вашем коде, при возникновении ошибки, сгенерируйте соответствующее исключение. Это делает код более ясным и обеспечивает структурированный способ обработки ошибок.

Обработка исключений: Далее, обработайте сгенерированные исключения в центральной части приложения или middleware, и верните JSON-ответ с описанием ошибки и соответствующим кодом состояния HTTP. Например:

```
{
    "error": "Пользователь не найден",
    "status_code": 404
}
```

В REST API стандартные коды состояния HTTP используются для указания результата выполнения запроса. Вот некоторые из наиболее распространенных кодов состояния HTTP и их значения:

Группа 1xx содержит информационные ответы. Эти коды состояния сообщают о том, что сервер получил запрос и продолжает обрабатывать его. Они чаще используются для информирования клиента о состоянии запроса.

- `100 Continue`: Запрос был понят и сервер ожидает продолжения.
- `101 Switching Protocols`: Сервер согласился на изменение протокола.
- Группа 2xx содержит успешные ответы. Эти коды состояния указывают на успешное выполнение запроса. Некоторые из кодов состояния 2xx включают:
- `200 OK`: Успешное выполнение запроса. Этот код состояния обычно используется при успешных операциях GET.
- `201 Created`: Ресурс успешно создан. Этот код состояния обычно используется при успешных операциях POST.
- `204 No Content`: Запрос выполнен успешно, но сервер не возвращает данные. Этот код состояния используется, когда не требуется возвращать данные в ответ на запрос DELETE.

Группа 3xx содержит коды состояния, которые указывают на необходимость перенаправления запроса. Эти коды часто используются, чтобы клиенты могли автоматически перейти по новому URL. Некоторые из кодов состояния 3xx включают:

- `300 Multiple Choices`: Есть несколько вариантов ответов, клиент может выбрать.
- `301 Moved Permanently`: Ресурс был перемещен постоянно.
- `302 Found`: Ресурс временно перемещен.

Группа 4xx содержит коды состояния, которые указывают на ошибки, связанные с запросом, сделанным клиентом. Эти коды состояния обычно свидетельствуют о проблемах с данными в запросе или с аутентификацией клиента. Некоторые из кодов состояния 4xx включают:

- `400 Bad Request`: Ошибка в запросе. Этот код состояния указывает на проблемы с данными в запросе.

- `401 Unauthorized`: Пользователь не аутентифицирован. Этот код состояния используется, когда требуется аутентификация для доступа к ресурсу.
- `403 Forbidden`: У пользователя нет разрешения на доступ к ресурсу. Этот код состояния используется для авторизационных ошибок.
- `404 Not Found`: Ресурс не найден. Этот код состояния указывает на отсутствие запрашиваемого ресурса.

Группа 5xx содержит коды состояния, которые указывают на ошибки, произошедшие на стороне сервера. Эти коды состояния обычно указывают на проблемы на сервере, которые могут быть вызваны внутренними ошибками сервера или перегрузкой. Некоторые из кодов состояния 5xx включают:

- `500 Internal Server Error`: Внутренняя ошибка сервера. Этот код состояния используется, когда на сервере произошла ошибка. Это может быть вызвано различными причинами, включая программные ошибки на сервере, нехватку ресурсов, сбои в работе сервера и другие технические проблемы.
- `501 Not Implemented`: Не реализовано. Этот код состояния указывает на то, что сервер не поддерживает или не реализовал функциональность, необходимую для выполнения запроса. По сути, сервер не знает, как обработать запрос, так как не имеет необходимой функциональности.
- `502 Bad Gateway`: Плохой, ошибочный шлюз. Код указывает на то, что сервер, выступая в роли шлюза или прокси-сервера, получил недействительный или ошибочный ответ от другого сервера, который он попытался использовать для выполнения запроса. Это может быть вызвано временной недоступностью другого сервера или ошибками в сетевой связи.
- `503 Service Unavailable`: Сервис недоступен. Код сообщает клиенту, что сервер временно недоступен для обработки запроса. Это может быть вызвано, например, перегрузкой сервера или его обслуживанием. Клиенту следует повторить запрос позднее.
- `504 Gateway Timeout`: Шлюз не отвечает. Код указывает на то, что сервер, выступая в роли шлюза или прокси-сервера, не получил ответ от другого сервера в разумный срок. Это может быть вызвано задержками или недоступностью другого сервера.

Важно правильно использовать стандартные коды состояния HTTP, чтобы клиенты могли легко интерпретировать результаты запросов и предпринимать необходимые действия в случае ошибок. Вместе с информативными сообщениями об ошибках это обеспечит более эффективное взаимодействие с вашим REST API.

??????????????

Версионирование в REST API - это процесс управления изменениями в API, чтобы обеспечить совместимость и поддержку существующих клиентов, даже когда API развивается и вносит изменения. Версионирование позволяет вам внедрять новые функции, ресурсы или

изменения в структуре данных без разрушения существующих клиентских приложений. Вот некоторые из подходов к версионированию REST API:

Версионирование в URL

В этом подходе версия API включается в URL. Обычно она указывается в виде числа или строки перед именем ресурса. Например:

```
https://api.example.com/v1/users
```

В случае внесения изменений, вы можете создать новую версию, например:

```
https://api.example.com/v2/users
```

Этот подход обеспечивает явную версионирование и позволяет клиентам явно указывать, какую версию они хотят использовать.

Версионирование в заголовке Accept

Другой подход - включить версию в заголовке `Accept` запроса. Например, клиент может отправить запрос с заголовком `Accept: application/vnd.example.v1+json`, чтобы запросить версию 1 API. Этот метод позволяет клиентам более гибко управлять версиями, но требует согласования между клиентом и сервером по правилам версионирования.

Версионирование в URL параметре

Вместо включения версии в URL или заголовке, ее можно передавать как параметр. Например:

```
https://api.example.com/users?version=1
```

Этот метод может быть удобен, если вам нужно поддерживать разные версии для разных параметров запроса.

Версионирование в заголовке Accept-Version

Похожий на предыдущий метод, версионирование может быть управляемо через заголовок `Accept-Version`. Например, `Accept-Version: 1.0` указывает на использование версии 1 API. Этот подход может предоставлять большую гибкость в управлении версиями.

Неявное версионирование

В некоторых случаях можно внедрить изменения в API, не меняя версию. Это может быть подходом, если изменения незначительны и не разрушают совместимость с клиентами. Однако следует быть осторожным, чтобы не нарушить существующие клиенты.

Выбор подхода к версионированию зависит от ваших потребностей и требований вашего проекта. Важно документировать версию API и оповещать клиентов о будущих изменениях, чтобы обеспечить плавное обновление и минимизацию проблем с совместимостью.

?????? ????????

Проектирование RESTful API - важный этап разработки, который влияет на его удобство использования и расширяемость. Вот несколько советов по проектированию REST API:

- **Используйте информативные URL**: URL должны быть понятными и описывать ресурсы. Например, используйте имена существительных во множественном числе для ресурсов, например, `/users`, `/products`.
- **Используйте правильные HTTP методы**: Применяйте стандартные HTTP методы (GET, POST, PUT, DELETE) в соответствии с их предназначением. Не используйте, например, GET для создания ресурсов.
- **Предоставляйте ясные ошибки и коды состояния**: Возвращайте правильные коды состояния HTTP и информативные сообщения об ошибках. Это помогает клиентам понять, что пошло не так.
- **Используйте аутентификацию и авторизацию**: Защитите свое API с помощью аутентификации и авторизации. Разрешите доступ только пользователям с необходимыми разрешениями.
- **Используйте версионирование**: Управляйте версиями вашего API, чтобы обеспечить совместимость с существующими клиентами при внесении изменений.
- **Используйте структурированные данные**: Предоставляйте данные в формате JSON или XML, чтобы упростить их обработку клиентами.
- **Предоставьте документацию**: Документируйте ваш API, включая описание ресурсов, доступных методов, примеры запросов и ответов.

Оптимизация производительности важна для обеспечения быстрого и отзывчивого API. Вот несколько советов по оптимизации производительности REST API:

- **Используйте кэширование**: Используйте HTTP-кэширование для сохранения ресурсов на клиентской стороне и уменьшения нагрузки на сервер.
- **Параллельная обработка**: Разрешите параллельную обработку запросов, чтобы обеспечить высокую производительность.
- **Оптимизируйте запросы к базе данных**: Используйте индексы, кэширование и другие методы для оптимизации запросов к базе данных.
- **Сжатие данных**: Используйте сжатие данных (например, gzip) для уменьшения объема передаваемых данных.
- **Ограничьте количество данных**: Предоставляйте параметры запроса, которые позволяют клиентам запросить только необходимые данные, чтобы уменьшить объем передаваемой информации.

- **Масштабируйте горизонтально**: Размещайте ваш API на нескольких серверах и масштабируйте его горизонтально для обработки больших нагрузок.
- **Мониторинг и профилирование**: Используйте инструменты мониторинга и профилирования, чтобы выявлять и устранять узкие места в производительности.
- **Оптимизация базы данных**: Настройте и оптимизируйте вашу базу данных для эффективного хранения и доступа к данным.

С учетом этих советов, вы можете создать производительное и эффективное REST API, которое будет удовлетворять потребности ваших клиентов и обеспечивать высокую производительность.

??????????????

Документирование вашего RESTful API является важной частью разработки, так как оно позволяет другим разработчикам или пользователям легко понимать, как использовать ваш API. В данном разделе мы рассмотрим два популярных инструмента для документирования REST API с использованием Django и Django REST framework: drf-spectacular и drf-yasg.

drf-spectacular

drf-spectacular - это инструмент для автоматической генерации документации REST API на основе вашего кода и аннотаций Django REST framework. Он позволяет создать информативную документацию, включая описание ресурсов, эндпойнтов, запросов, и ответов. Вот как его использовать:

- Установите drf-spectacular с помощью pip:

```
pip install drf-spectacular
```

- В файле settings.py вашего проекта добавьте 'drf_spectacular' в список установленных приложений:

```
INSTALLED_APPS = [
   # ...
   'drf_spectacular',
   # ...
]
```

- Настройте drf-spectacular в settings.py:

```
SPECTACULAR_SETTINGS = {
   'TITLE': 'My API',
   'DESCRIPTION': 'API documentation for My Project',
   'VERSION': '1.0.0',
}
```

- Создайте точку входа (URL) для генерации документации:

```
from drf_spectacular.views import SpectacularAPIView, SpectacularSwaggerView

urlpatterns = [
    # ...
    path('schema/', SpectacularAPIView.as_view(), name='schema'),
    path('swagger/', SpectacularSwaggerView.as_view(url_name='schema'), name='swagger-ui'),
    # ...
]
```

Теперь, перейдя по URL /swagger/, вы увидите сгенерированную документацию вашего API, которая позволит пользователям и разработчикам понять, как использовать ваши ресурсы и эндпойнты.

Пример описания ресурса и эндпойнта с использованием аннотаций Django REST framework:

```
from drf_spectacular.utils import extend_schema

@extend_schema(
   summary="Get a list of items",
   description="This endpoint returns a list of items.",
   responses={200: ItemSerializer(many=True)},
)
class ItemList(APIView):
   def get(self, request):
       items = Item.objects.all()
       serializer = ItemSerializer(items, many=True)
       return Response(serializer.data)
```

drf-yasg

drf-yasg (Yet Another Swagger Generator) - это еще один инструмент для генерации документации REST API с использованием Django REST framework. Он предоставляет возможность создавать документацию в формате Swagger (OpenAPI) для вашего API. Вот как

его использовать:

- Установите drf-yasg с помощью pip:

```
pip install drf-yasg
```

- В файле settings.py вашего проекта добавьте 'drf_yasg' в список установленных приложений:

```
INSTALLED_APPS = [
   # ...
   'drf_yasg',
   # ...
]
```

- Настройте drf-yasg в settings.py:

```
SWAGGER_SETTINGS = {
   'SECURITY_DEFINITIONS': {
       'api_key': {
           'type': 'apiKey',
           'name': 'Authorization',
           'in': 'header',
       },
   },
   'USE_SESSION_AUTH': False,
   'PERSIST_AUTH': True,
}
```

- Создайте точку входа (URL) для генерации документации:

```
from rest_framework import permissions
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

schema_view = get_schema_view(
    openapi.Info(
        title="My API",
        default_version='v1',
        description="API documentation for My Project",
```

```
        terms_of_service="https://www.myproject.com/terms/",
        contact=openapi.Contact(email="contact@myproject.com"),
        license=openapi.License(name="My License"),
    ),
    public=True,
    permission_classes=(permissions.AllowAny,),
)

urlpatterns = [
    # ...
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-
ui'),
    path('redoc/', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
    # ...
]
```

После этого, перейдя по URL /swagger/ или /redoc/, вы увидите сгенерированную документацию вашего API в формате Swagger или ReDoc.

drf-spectacular и drf-yasg предоставляют мощные инструменты для документирования RESTful API в Django и Django REST framework, и выбор между ними зависит от ваших предпочтений и требований проекта. С их помощью вы можете обеспечить четкую и информативную документацию для вашего API, что сделает его использование более удобным для разработчиков и пользователей.

??????????

REST API (Representational State Transfer Application Programming Interface) играет важную роль в разработке веб-приложений и обмене данными между клиентами и серверами. Важность REST API заключается в нескольких ключевых аспектах:

Удобство использования: REST API предоставляет простой и интуитивно понятный способ взаимодействия с серверами. Он основан на стандартных HTTP методах, что делает его доступным для разработчиков на разных платформах.

Расширяемость и гибкость: REST API позволяет легко добавлять новые ресурсы и методы без нарушения существующей функциональности. Это обеспечивает гибкость в развитии приложений.

Независимость клиентов и серверов: RESTful API позволяет клиентам и серверам быть независимыми друг от друга. Это означает, что клиенты могут развиваться и обновляться

независимо от сервера и наоборот.

Прозрачность: REST API часто использует структурированные форматы данных, такие как JSON, что обеспечивает прозрачность в передаче информации между клиентами и серверами.

Безопасность: REST API может использовать методы аутентификации и авторизации для защиты данных и ресурсов. Это важно для обеспечения безопасности приложений.

??????????? ????????

RESTful API остается актуальным и популярным в мире веб-разработки. Однако, с развитием технологий, появляются новые тенденции и подходы. Некоторые из перспектив развития REST API включают:

GraphQL: GraphQL - это язык запросов, который позволяет клиентам запрашивать только необходимые данные, что устраняет избыточную передачу данных. GraphQL становится популярным альтернативным подходом к REST API.

gRPC: gRPC - это высокопроизводительный протокол обмена данными, который может использоваться для создания API. Он использует Protocol Buffers для определения данных и API, и поддерживает двунаправленное взаимодействие.

Serverless и микросервисы: Развитие архитектуры микросервисов и серверлесс-технологий создает новые способы организации и развертывания API, что может повлиять на будущее развитие RESTful API.

Секьюрити: С увеличением интереса к безопасности, будущее развитие REST API включает в себя более жесткие меры безопасности, такие как OAuth 2.0 и OpenID Connect.

В целом, REST API остается важной частью веб-разработки и будет продолжать развиваться и адаптироваться к потребностям разработчиков и клиентов в будущем. Это мощный инструмент для создания веб-приложений и обеспечения обмена данными в интернете.

?????? ????????????? ???????????

1. Разбираемся в REST API с примерами на Python
2. Проектирование RESTful API с помощью Python и Flask
3. Пишем свой REST API на Python с Flask: подробный guide

??? ????????? ??????? ? Python

Словари являются одной из наиболее важных и полезных структур данных в Python. Они могут помочь вам решить широкий спектр задач программирования. Из этой статьи вы узнаете, как итерировать словарь в Python.

К концу этой статьи вы узнаете:

- Что такое словари, а также некоторые их основные функции и детали реализации
- Как итерировать словарь в Python, используя основные инструменты, предлагаемые языком
- Какие реальные задачи вы можете выполнить, просматривая словарь в Python
- Как использовать некоторые более продвинутые методы и стратегии для перебора словаря в Python

Для получения дополнительной информации о словарях, вы можете обратиться к следующим ресурсам:

- Dictionaries in Python
- Itertools in Python 3, By Example
- The documentation for `map()` and `filter()`

Готовы? Поехали!

????????? ???? ? ????????

Словари являются краеугольным камнем Python. Сам язык построен вокруг словарей. Модули, классы, объекты, globals(), locals(): все это словари. Словари были центральным элементом для Python с самого начала.

Официальная документация Python определяет словарь следующим образом:

> Ассоциативный массив, где произвольные ключи отображаются на значения. Ключами могут быть любые объекты с методами __hash__() и __eq__(). (Источник)

Есть несколько моментов, о которых следует помнить:

1. Словари сопоставляют ключи со значениями и сохраняют их в массиве или коллекции.
2. Ключи должны быть хэшируемого типа (hashable), что означает, что в качестве ключа используют хэш значения ключа, который никогда не меняется в течение срока жизни ключа.

Словари часто используются для решения всевозможных задач программирования, поэтому они являются фундаментальной частью инструментария разработчика Python.

В отличие от последовательностей (**sequences**), которые так же являются итерационными поддерживающими доступ к элементам с использованием целочисленных индексов, словари индексируются по ключам.

Ключи в словаре очень похожи на множества `set`, представляющие собой коллекцию уникальных объектов, которые можно хэшировать. Поскольку объекты должны быть хешируемыми, изменяемые (mutable) объекты нельзя использовать в качестве ключей словаря.

С другой стороны, значения могут быть любого типа Python, независимо от того, являются они хэшируемыми или нет. Там нет буквально никаких ограничений.

В Python 3.6 и более поздних версиях ключи и значения словаря перебираются в том же порядке, в котором они были созданы. Однако это поведение может отличаться в разных версиях Python и зависит от истории вставок и удалений словаря.

В Python 2.7 словари являются неупорядоченными структурами. Порядок элементом словарей неизменяемый. Это означает, что порядок элементов является детерминированным и повторяемым. Давайте посмотрим на пример:

```
>>> # Python 2.7
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
>>> a_dict
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
```

Если вы покинете интерпритатор и позже откроете новый интерактивный сеанс, вы получите тот же порядок элементов:

```
>>> # Python 2.7. New interactive session
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
```

```
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
>>> a_dict
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
```

Более внимательное рассмотрение этих двух выходных данных показывает, что результирующий порядок в обоих случаях одинаков. Вот почему вы можете сказать, что порядок является детерминированным.

В Python 3.5 словари все еще неупорядочены, но на этот раз **рандомизированные** структуры данных. Это означает, что каждый раз, когда вы снова запускаете словарь, вы получаете другой порядок элементов. Давайте посмотрим:

```
>>> # Python 3.5
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
>>> a_dict
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
```

Если вы войдете в новый интерактивный сеанс, вы получите следующее:

```
>>> # Python 3.5. New interactive session
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
{'fruit': 'apple', 'pet': 'dog', 'color': 'blue'}
>>> a_dict
{'fruit': 'apple', 'pet': 'dog', 'color': 'blue'}
```

На этот раз вы можете видеть, что порядок элементов отличается на обоих выходах. Вот почему вы можете сказать, что это рандомизированные структуры данных.

В Python 3.6 и более **поздних версиях словари являются упорядоченными структурами данных**, что означает, что они хранят свои элементы в том же порядке, в котором они были созданы, как вы можете видеть здесь:

```
>>> # Python 3.6 and beyond
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
{'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
{'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
```

Это относительно новая функция словарей Python, и она очень полезна. Но если вы пишете код, который может быть запущен в разных версиях Python, вы не должны полагаться на этот функционал, поскольку он может генерировать некорректное поведение.

Другой важной особенностью словарей является то, что они являются изменчивыми структурами данных, что означает, что вы можете добавлять, удалять и обновлять их элементы. Стоит отметить, что это также означает, что их нельзя использовать в качестве ключей для других словарей, так как они не являются объектами, которые можно хэшировать.

Примечание. Все, что описано в этом разделе, относится к базовой реализации Python, CPython.

Другие реализации Python, такие как PyPy, IronPython или Jython, могут демонстрировать другое поведение словаря, которые выходят за рамки данной статьи.

??? ??????????? ???????? ? Python: ??????

Словари являются полезной и широко используемой структурой данных в Python. Как программист Python, вы часто будете в ситуациях, когда вам придется перебирать словарь, в то время как вы выполняете некоторые действия над его парами ключ-значение.

Когда дело доходит до перебора словаря в Python, язык предоставляет вам несколько отличных инструментов, которые мы рассмотрим в этой статье.

?????? ???????? ?? ??????

Словари Python являются отображающими объектами. Это означает, что они наследуют некоторые **специальные методы**, которые Python внутренне использует для выполнения некоторых операций. Эти методы называются с использованием соглашения об именах, заключающегося в добавлении двойного подчеркивания в начале и в конце имени метода.

Чтобы визуализировать методы и атрибуты любого объекта Python, вы можете использовать **dir()**, которая является встроенной функцией. Если вы запустите **dir()** с пустым словарем в качестве аргумента, вы сможете увидеть все методы и атрибуты, которые реализуют словари:

```
>>> dir({})
['__class__', '__contains__', '__delattr__', ... , '__iter__', ...]
```

Если вы внимательно посмотрите на предыдущий вывод, вы увидите **__iter__**. Это метод, который вызывается, когда для контейнера требуется итератор, и он должен возвращать новый объект итератора, который может выполнять итерацию по всем объектам в контейнере.

Примечание: вывод предыдущего кода был сокращен (...) для экономии места.

Для мапинга (например, словарей) **.__iter__()** должен перебирать ключи. Это означает, что если вы поместите словарь непосредственно в цикл `for`, Python автоматически вызовет **.__iter__()** для этого словаря, и вы получите итератор по его ключам:

```
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> for key in a_dict:
... print(key)
...
color
fruit
pet
```

Python достаточно умен, чтобы знать, что **a_dict** – это словарь и что он реализует **.__iter__()**. В этом примере Python автоматически вызывает **.__iter__(),** и это позволяет вам перебирать ключи **a_dict**.
Это самый простой способ перебора словаря в Python. Просто поместите его прямо в цикл **for**, и все готово!

Если вы используете этот подход вместе с небольшой уловкой, то вы можете обрабатывать ключи и значения любого словаря. Хитрость заключается в использовании оператора индексации [] со словарем и его ключами для получения доступа к значениям:

```
>>> for key in a_dict:
... print(key, '->', a_dict[key])
...
color -> blue
fruit -> apple
pet -> dog
```

Предыдущий код позволил вам получить доступ к ключам (**key**) и значениям (**a_dict[key]**) **a_dict** одновременно. Таким образом, вы можете выполнить любую операцию как с ключами, так и со значениями.

???????? ?? .items()

Когда вы работаете со словарями, вероятно, вы захотите работать как с ключами, так и со значениями. Одним из наиболее полезных способов перебора словаря в Python является использование метода **.items()**, который возвращает новый вид элементов словаря:

```
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> d_items = a_dict.items()
>>> d_items # Here d_items is a view of items
dict_items([('color', 'blue'), ('fruit', 'apple'), ('pet', 'dog')])
```

Представления словаря, такое как **d_items**, обеспечивают динамическое представление записей словаря, что означает, что при изменении словаря представления отражают эти изменения.

Представления могут быть перебраны для получения соответствующих данных, поэтому вы можете итерировать словарь в Python, используя объект представления, возвращаемый **.items()**:

```
>>> for item in a_dict.items():
... print(item)
...
('color', 'blue')
('fruit', 'apple')
('pet', 'dog')
```

Объект представления, возвращаемый функцией **.items()**, выдает пары ключ-значение по одной и позволяет перебирать словарь, и таким образом, вы получаете доступ к ключам и значениям одновременно.

Если вы присмотритесь к отдельным элементам, полученным с помощью **.items()**, вы заметите, что они действительно являются кортежами объектов. Давайте посмотрим:

```
>>> for item in a_dict.items():
... print(item)
... print(type(item))
...
('color', 'blue')
<class 'tuple'>
('fruit', 'apple')
<class 'tuple'>
('pet', 'dog')
<class 'tuple'>
```

Вы можете использовать распаковку кортежей для перебора ключей и значений словаря. Для этого вам просто нужно распаковать элементы каждого элемента в две разные переменные, представляющие ключ и значение:

```
>>> for key, value in a_dict.items():
... print(key, '->', value)
```

```
...
color -> blue
fruit -> apple
pet -> dog
```

Здесь, переменные **key** и **value** в заголовке вашего цикла **for** распаковываются. Каждый раз, когда цикл запускается, **key** будет хранить ключ, а **value** будет хранить значение элемента, который был обработан. Таким образом, у вас будет больше контроля над элементами словаря, и вы сможете обрабатывать ключи и значения отдельно.

Примечание: обратите внимание, что **.values()** и **.keys()** возвращают объекты представления так же, как **.items()**, как вы увидите в следующих двух разделах.

???????? ????? .keys()

Если вам просто нужно работать с ключами словаря, то вы можете использовать метод **.keys()**, который возвращает новый объект представления, содержащий ключи словаря:

```
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> keys = a_dict.keys()
>>> keys
dict_keys(['color', 'fruit', 'pet'])
```

Чтобы перебрать словарь в Python с помощью **.keys()**, вам просто нужно вызвать **.keys()** в заголовке цикла for:

```
>>> for key in a_dict.keys():
... print(key)
...
color
fruit
pet
```

Используя тот же трюк, который вы видели ранее (оператор индексации []), вы можете получить доступ к значениям словаря:

```
>>> for key in a_dict.keys():
... print(key, '->', a_dict[key])
...
color -> blue
fruit -> apple
pet -> dog
```

Таким образом, вы получите доступ к ключам (**key**) и значениям (**a_dict[key]**) **a_dict** одновременно, и вы сможете выполнять с ними любые действия.

???????? ?? .values()

Также можно использовать значения для перебора словаря. Один из способов сделать это – использовать **.values()**, который возвращает представление со значениями словаря:

```
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> values = a_dict.values()
>>> values
dict_values(['blue', 'apple', 'dog'])
```

В предыдущем коде **values** содержит ссылку на объект представления, содержащий значения **a_dict**.

Как и любой объект представления, объект, возвращаемый функцией **.values()**, также может быть итерирован. В этом случае **.values()** возвращает значения **a_dict**:

```
>>> for value in a_dict.values():
... print(value)
...
blue
apple
dog
```

Стоит отметить, что методы **keys()** и **values()** также поддерживают тесты членства (in) (membership tests (`in`)), что является важной функцией, если вы пытаетесь узнать, есть ли определенный элемент в словаре или нет:

```
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> 'pet' in a_dict.keys()
True
>>> 'apple' in a_dict.values()
True
>>> 'onion' in a_dict.values()
False
```

Проверка членства с помощью **in** возвращает **True**, если ключ (или значение или элемент) присутствует в тестируемом словаре, и возвращает **False** в противном случае. Тест на членство позволяет вам не выполнять итерацию по словарю в Python, если вы просто хотите узнать, присутствует ли определенный словарь (или значение, или элемент) в словаре или нет.

????????? ???????? ? ??????

Часто бывает необходимо изменить значения и ключа, когда вы перебираете словарь в Python. Есть некоторые моменты, которые вы должны принять во внимание, чтобы выполнить эту задачу.

Например, значения можно изменять всякий раз, когда вам нужно, но вам нужно будет использовать исходный словарь и ключ, который отображает значение, которое вы хотите изменить:

```
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> for k, v in prices.items():
... prices[k] = round(v * 0.9, 2) # Apply a 10% discount
...
>>> prices
{'apple': 0.36, 'orange': 0.32, 'banana': 0.23}
```

В этом примере кода, чтобы изменить значения **prices** и применить скидку 10%, мы использовали выражение **prices[k] = round(v * 0.9, 2)**.

Так зачем вам использовать оригинальный словарь, если у вас есть доступ к его ключу (**k)** и его значениям (**v**)? Если мы можем изменить их напрямую?

Реальная проблема заключается в том, что изменения **k** и **v** не отражаются в исходном словаре. То есть, если вы измените какой-либо из них (**k** или **v**) непосредственно внутри цикла, то, что действительно происходит, так это то, что вы потеряете ссылку на соответствующий компонент словаря, не изменяя ничего в словаре.

С другой стороны, ключи могут быть добавлены или удалены из словаря путем преобразования представления, возвращаемого функцией **.keys()**, в объект **list**:

```
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> for key in list(prices.keys()): # Use a list instead of a view
... if key == 'orange':
... del prices[key] # Delete a key from prices
...
>>> prices
{'apple': 0.4, 'banana': 0.25}
```

Этот подход может иметь некоторые последствия для производительности, в основном связанные с потреблением памяти. Например, вместо объекта просмотра, который выдает элементы по требованию, у вас будет полный новый **list** в памяти вашей системы. Тем не менее, это может быть безопасным способом изменения ключей при переборе словаря в Python.

Наконец, если вы попытаетесь удалить ключ из **prices**, используя напрямую **.keys()**, тогда Python вызовет **RuntimeError**, сообщающую, что размер словаря изменился во время итерации:

```
>>> # Python 3. dict.keys() returns a view object, not a list
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> for key in prices.keys():
... if key == 'orange':
... del prices[key]
...
Traceback (most recent call last):
File "<input>", line 1, in <module>
for key in prices.keys():
RuntimeError: dictionary changed size during iteration
```

Это потому, что **.keys()** возвращает объект словаря-представления, который выдает ключи по запросу по одному, и если вы удаляете элемент (**del values [key]**), то Python вызывает **RuntimeError**, потому что вы изменили словарь во время итерации.

Примечание. В Python 2 объекты **.items()**, **.keys()** и **.values()** возвращают список объектов. Но **.iteritems()**, **iterkeys()** и **.itervalues()** возвращают итераторы. Итак, если вы используете Python 2, то вы можете изменить ключи словаря, используя **.keys()** напрямую.

С другой стороны, если вы используете **iterkeys()** в своем коде Python 2 и пытаетесь изменить ключи словаря, вы получите **RuntimeError**.

??????? ?? ?????????? ????

До сих пор вы видели более простые способы перебора словаря в Python. Теперь пришло время посмотреть, как вы можете выполнять некоторые действия с элементами словаря во время итерации. Давайте посмотрим на некоторые примеры из реальной жизни.

Примечание. Позже в этой статье вы увидите другой способ решения тех же самых проблем с помощью других инструментов Python.

??????????? ?????? ? ???????? ? ????????

Предположим, у вас есть словарь и по какой-то причине необходимо превратить ключи в значения и наоборот. В этой ситуации вы можете использовать цикл **for** для перебора словаря и создания нового словаря, используя ключи в качестве значений и наоборот:

```
>>> a_dict = {'one': 1, 'two': 2, 'thee': 3, 'four': 4}
>>> new_dict = {}
>>> for key, value in a_dict.items():
... new_dict[value] = key
...
>>> new_dict
{1: 'one', 2: 'two', 3: 'thee', 4: 'four'}
```

Выражение **new_dict[value] = key** сделает всю работу за вас, превратив ключи в значения и используя значения в качестве ключей. Чтобы этот код работал, данные, хранящиеся в исходных значениях, должны иметь тип данных, который можно хэшировать.

??????????

Иногда вы будете в ситуациях, когда у вас есть словарь, и вы захотите создать новый, чтобы хранить только данные, которые удовлетворяют заданному условию. Вы можете сделать это с помощью **if** внутри цикла **for** следующим образом:

```
>>> a_dict = {'one': 1, 'two': 2, 'thee': 3, 'four': 4}
>>> new_dict = {} # Create a new empty dictionary
>>> for key, value in a_dict.items():
... # If value satisfies the condition, then store it in new_dict
... if value <= 2:
... new_dict[key] = value
...
>>> new_dict
{'one': 1, 'two': 2}
```

В этом примере вы отфильтровали элементы со значением больше 2. Теперь **new_dict** содержит только элементы, которые удовлетворяют условному значению <= 2. Это одно из возможных решений для такого рода проблем. Позже вы увидите более понятный и понятный способ получить тот же результат.

????????? ????????? ???????

Также часто требуется выполнять некоторые вычисления, пока вы перебираете словарь в Python. Предположим, вы сохранили данные о продажах вашей компании в словаре, и теперь вы хотите узнать общий доход за год.

Чтобы решить эту проблему, вы можете определить переменную с начальным значением ноль. Затем вы можете накапливать каждое значение вашего словаря в этой переменной:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> total_income = 0.00
>>> for value in incomes.values():
... total_income += value # Accumulate the values in total_income
...
>>> total_income
14100.0
```

Здесь вы просматривали **incomes** и последовательно накапливали их значения в **total_income**, как и хотели. Выражение **total_income += value** делает все работу, и в конце цикла вы получите общий доход за год. Обратите внимание, что **total_income += value** эквивалентно **total_income = total_income + value**.

?????????????? ??????????? (comprehensions)

Генераторы словарей (**Dictionary comprehension**) – это компактный способ обработки всех или части элементов в коллекции и возврата словаря в качестве результата. В отличие от списочных представлений, им нужны два выражения, разделенные двоеточием, за которым следуют предложения **for** и **if** (необязательно). Когда выполняется генератор словаря, полученные пары ключ-значение вставляются в новый словарь в том же порядке, в котором они были созданы.

Предположим, например, что у вас есть два списка данных, и вам нужно создать новый словарь из них. В этом случае вы можете использовать zip (* iterables) в Python для циклического обхода элементов обоих списков:

```
>>> objects = ['blue', 'apple', 'dog']
>>> categories = ['color', 'fruit', 'pet']
>>> a_dict = {key: value for key, value in zip(categories, objects)}
>>> a_dict
{'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
```

Здесь **zip()** получает два итерируемых объекта categories и objects в качестве аргументов и создает итератор, который объединяет элементы из каждого объекта. Объекты кортежа, сгенерированные zip(), затем распаковываются в ключ и значение, которые в итоге используются для создания нового словаря.

Генератор словарей открывает широкий спектр новых возможностей и предоставляет вам отличный инструмент для перебора словаря в Python.

??? ??? ? ??????????? ?????? ? ???????? ? ????????

Если вы по-другому взгляните на проблему превращения ключей в значения и наоборот, вы увидите, что вы могли бы написать более эффективное решение с использованием генератора словарей:

```
>>> a_dict = {'one': 1, 'two': 2, 'thee': 3, 'four': 4}
>>> new_dict = {value: key for key, value in a_dict.items()}
>>> new_dict
{1: 'one', 2: 'two', 3: 'thee', 4: 'four'}
```

С генератором словаря вы создали совершенно новый словарь, в котором ключи заняли место значений, и наоборот. Этот новый подход дал вам возможность писать более читабельный, лаконичный, эффективный и Pythonic код.

Условие для работы этого кода такое же, как вы видели ранее: значения должны быть объектами, которые можно хэшировать. В противном случае вы не сможете использовать их в качестве ключей для **a_dict**.

????????? ??????????

Чтобы отфильтровать элементы в словаре с генератором, вам просто нужно добавить условие **if**, которое определяет условие, которое вы хотите выполнить. В предыдущем примере, когда вы фильтровали словарь, это условие было, если v <= 2. С этим условием **if**, добавленным в конец генератора словаря, вы отфильтруете элементы, значения которых больше 2. Давайте посмотрим:

```
>>> a_dict = {'one': 1, 'two': 2, 'thee': 3, 'four': 4}
>>> new_dict = {k: v for k, v in a_dict.items() if v <= 2}
>>> new_dict
{'one': 1, 'two': 2}
```

Теперь **new_dict** содержит только элементы, которые удовлетворяют вашему условию. По сравнению с предыдущими решениями, это код более Pythonic и эффективный.

????????? ?????????? ????????

Помните пример с продажами компании? Если вы используете генератор списков (**list comprehension**) для перебора значений словаря, вы получите более компактный, быстрый и Pythonic-код:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> total_income = sum([value for value in incomes.values()])
>>> total_income
14100.0
```

Генератор списков создал объект **list**, содержащий значения **incomes**, а затем мы суммировали их все с помощью функции **sum()** и сохранили результат в **total_income**.

Если вы работаете с действительно большим словарем, и использование памяти является проблемой для вас, тогда вы можете использовать **выражение-генератор** (**generator expression**) вместо генератора списков. **выражение-генератор** – это выражение, которое возвращает итератор. Это похоже на генератор списков, но вместо квадратных скобок для его определения необходимо использовать круглые скобки:

```
>>> total_income = sum(value for value in incomes.values())
>>> total_income
14100.0
```

То есть если вы поменяете квадратные скобки для пары круглых скобок (здесь круглые скобки sum()), вы превратите генератор списков в выражение-генератор, и ваш код будет более эффективным в памяти, потому что выражение-генератор использует элементы по запросу. Вместо того, чтобы создавать и хранить весь список в памяти, он будет хранить только один элемент за раз.

Примечание. Если вы новичок в выражение-генератор (generator expressions), вы можете взглянуть на Introduction to Python Generators, чтобы лучше изучить тему.

Наконец, есть более простой способ решить эту проблему, просто используя **incomes.values()** непосредственно в качестве аргумента для **sum()**:

```
>>> total_income = sum(incomes.values())
>>> total_income
14100.0
```

sum() получает в качестве аргумента итерацию и возвращает общую сумму его элементов. Здесь **incomes.values()** играет роль итерируемого значения, переданного в sum(). Результат – общий доход, который вы искали.

???????? ????????? ?????????

Теперь предположим, что у вас есть словарь, и вам нужно создать новый с удаленными выбранными ключами. Помните, как объекты словаря похожи на sets? Эти сходства выходят за рамки просто коллекции хэшируемых и уникальных объектов. Эти объекты также поддерживают общие операции над множествами. Давайте посмотрим, как вы можете

воспользоваться этим, чтобы удалить определенные элементы из словаря:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> non_citric = {k: incomes[k] for k in incomes.keys() - {'orange'}}
>>> non_citric
{'apple': 5600.0, 'banana': 5000.0}
```

Этот код работает, потому что элементы словаря поддерживают операции над множествами, такие как объединения, пересечения и различия. Когда вы написали **incomes.keys () – {‘orange’}**, вы выполняли операцию с заданным различием. Если вам нужно выполнить какие-либо операции над множествами с ключами словаря, то вы можете просто использовать элементы напрямую, без предварительного преобразования его в set.

?????????? ???????

Часто необходимо сортировать элементы коллекции. Начиная с Python 3.6, словари являются упорядоченными структурами данных, поэтому, если вы используете Python 3.6 (и более поздние версии), вы сможете сортировать элементы любого словаря с помощью **sorted()** и с помощью генератора словаря:

```
>>> # Python 3.6, and beyond
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> sorted_income = {k: incomes[k] for k in sorted(incomes)}
>>> sorted_income
{'apple': 5600.0, 'banana': 5000.0, 'orange': 3500.0}
```

Этот код позволяет создать новый словарь с ключами в отсортированном порядке. Это возможно, потому что **sorted(incomes)** возвращает список отсортированных ключей, которые можно использовать для создания нового словаря **sorted_dict**.

???????? ? ??????????????? ???????

Иногда вам может понадобиться перебрать словарь в Python, но вы хотите сделать это в отсортированном порядке. Это может быть достигнуто с помощью **sorted()**. Когда вы вызываете **sorted(iterable)**, вы получаете list с элементами в отсортированном порядке.

Давайте посмотрим, как вы можете использовать **sorted()** для перебора словаря, когда вам нужно сделать это в отсортированном порядке.

?????????? ?? ??????

Если вам нужно перебрать словарь в Python и отсортировать его по ключам, вы можете использовать свой словарь в качестве аргумента для **sorted()**. Это вернет list, содержащий ключи в отсортированном порядке, и вы сможете их перебирать:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> for key in sorted(incomes):
... print(key, '->', incomes[key])
...
apple -> 5600.0
banana -> 5000.0
orange -> 3500.0
```

В этом примере вы отсортировали словарь (в алфавитном порядке) по ключам, используя **sorted(incomes)** в заголовке цикла **for**. Обратите внимание, что вы также можете использовать **sorted(incomes.keys())**, чтобы получить тот же результат. В обоих случаях вы получите список, содержащий ключи вашего словаря в отсортированном порядке.

Примечание. Порядок сортировки будет зависеть от типа данных, который вы используете для ключей или значений, и от внутренних правил, которые Python использует для сортировки этих типов данных.

?????????? ?? ?????????

Вам также может понадобиться перебрать словарь в Python с его элементами, отсортированными по значениям. Вы также можете использовать **sorted()**, но со вторым аргументом **key**.

Аргумент **key** определяет функцию одного аргумента, которая используется для извлечения ключа сравнения для каждого элемента, который вы обрабатываете.

Чтобы отсортировать элементы словаря по значениям, вы можете написать функцию, которая возвращает значение каждого элемента, и использовать эту функцию в **key** аргумента для **sorted()**:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> def by_value(item):
... return item[1]
...
>>> for k, v in sorted(incomes.items(), key=by_value):
... print(k, '->', v)
...
```

```
('orange', '->', 3500.0)
('banana', '->', 5000.0)
('apple', '->', 5600.0)
```

В этом примере мы определили **by_value()** и использовали его для сортировки **incomes** по значению. Затем мы перебираем словарь в порядке сортировки, используя **sorted()**. Ключевая функция (**by_value ()**) сообщает **sorted()** отсортировать **incomes.items()** по второму элементу каждого элемента, то есть по значению (**item [1]**).

Вы также можете просто перебирать значения словаря в отсортированном порядке, не беспокоясь о ключах. В этом случае вы можете использовать **.values()** следующим образом:

```
>>> for value in sorted(incomes.values()):
... print(value)
...
3500.0
5000.0
5600.0
```

sorted(comes.values ()) возвращает значения словаря в отсортированном порядке по вашему желанию. Ключи не будут доступны, если вы используете **incomes.values()**, но иногда вам не нужны ключи, только значения, и это быстрый способ получить к ним доступ.

????????

Если вам нужно отсортировать словари в обратном порядке, вы можете добавить **reverse = True** в качестве аргумента для **sorted()**. Ключевое слово аргумент **reverse** должно принимать логическое значение. Если установлено значение **True**, то элементы сортируются в обратном порядке:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> for key in sorted(incomes, reverse=True):
... print(key, '->', incomes[key])
...
orange -> 3500.0
banana -> 5000.0
apple -> 5600.0
```

Здесь вы перебирали ключи доходов в обратном порядке, используя **sorted(incomes, reverse=True)** в заголовке цикла **for**.

Наконец, важно отметить, что **sorted()** не меняет порядок основного словаря. Что на самом деле происходит, так это то, что **sorted()** создает независимый список со своими элементами в отсортированном порядке, поэтому **incomes** остаются прежними:

>>> incomes
{'apple': 5600.0, 'orange': 3500.0, 'banana': 5000.0}

?????????????? ???????? ? ??????? .popitem()

Иногда нужно перебрать словарь в Python и последовательно удалить его элементы. Для выполнения этой задачи можно использовать **.popitem()**, которая удалит и возвратит произвольную пару ключ-значение из словаря. С другой стороны, когда вы вызываете **.popitem()** в пустом словаре, он вызывает ошибку **KeyError**.

Пример:

```
# File: dict_popitem.py

a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}

while True:
try:
print(f'Dictionary length: {len(a_dict)}')
item = a_dict.popitem()
# Do something with item here...
print(f'{item} removed')
except KeyError:
print('The dictionary has no item now...')
break
```

Внутри цикла **while** мы определили блок **try…except** для того, чтобы поймать **KeyError**, сгенерированный функцией **.popitems()**, когда **a_dict** станет пустым. В блоке **try…except** мы обрабатываем словарь, удаляя элемент в каждой итерации. Переменная **item** хранит ссылку на последующие элементы и позволяет нам выполнять с ними некоторые действия.

Если мы запустим этот скрипт из командной строки, то мы получим следующие результаты:

```
$ python3 dict_popitem.py
Dictionary length: 3
('pet', 'dog') removed
```

```
Dictionary length: 2
('fruit', 'apple') removed
Dictionary length: 1
('color', 'blue') removed
Dictionary length: 0
The dictionary has no item now...
```

Здесь **.popitem()** последовательно удаляет элементы из **a_dict**. Цикл прервался, когда словарь стал пустым, и **.popitem()** вызвал исключение **KeyError**.

?????????????? ?????????? ?? ?????????? ???????? Python

Python предоставляет некоторые встроенные функции, которые могут быть полезны при работе со словарями. Эти функции являются своего рода инструментом итерации, который предоставляет вам еще один способ перебора словаря. Давайте посмотрим на некоторые из них.

map()

map() определяется как **map(function, iterable, ...)** и возвращает итератор, который применяет функцию к каждому элементу итерируемого. Таким образом, **map()** можно рассматривать как инструмент итерации, который можно использовать для перебора словаря.

Предположим, у вас есть словарь, содержащий цены на несколько продуктов, и вам нужно применить скидку к ним. В этом случае вы можете определить функцию, которая управляет скидкой, а затем использует ее в качестве первого аргумента для **map()**. Вторым аргументом может быть **price.items()**:

```
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> def discount(current_price):
... return (current_price[0], round(current_price[1] * 0.95, 2))
...
>>> new_prices = dict(map(discount, prices.items()))
>>> new_prices
{'apple': 0.38, 'orange': 0.33, 'banana': 0.24}
```

Здесь **map()** перебирает элементы словаря (**prices.items()**), чтобы применить скидку 5% к каждому фрукту с помощью **discount()**. В этом случае вам нужно использовать **dict()** для создания словаря **new_prices** из итератора, возвращаемого **map()**.

Обратите внимание, что **discount()** возвращает кортеж в форме (**key, value**), где **current_price[0]** представляет ключ, а **round(current_price [1] * 0.95, 2)** представляет новое значение.

filter()

filter () – это еще одна встроенная функция, которую вы можете использовать для перебора словаря и фильтрации некоторых его элементов. Эта функция определяется как **filter(function, iterable)** и возвращает итератор из тех элементов итерируемого, для которых функция возвращает значение **True**.

Предположим, вы хотите знать продукты с ценой ниже 0,40. Вам нужно определить функцию, чтобы определить, удовлетворяет ли цена этому условию, и передать ее в качестве первого аргумента для **filter()**. Вторым аргументом может быть **prices.keys()**:

```
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> def has_low_price(price):
... return prices[price] < 0.4
...
>>> low_price = list(filter(has_low_price, prices.keys()))
>>> low_price
['orange', 'banana']
```

Здесь вы перебирали ключи **prices** с помощью **filter()**. Тогда **filter()** применяет **has_low_price()** к каждому ключу **prices**. Наконец нужно использовать **list()** для генерации списка продуктов с низкой ценой, потому что **filter()** возвращает итератор, и нам нужен объект list.

?????????????? collections.ChainMap

`collections` – полезный модуль из стандартной библиотеки Python, предоставляющий специализированные типы данных контейнеров. Одним из таких типов данных является **ChainMap**, который является словарным классом для создания единого представления нескольких сопоставлений (например, словарей). С **ChainMap** вы можете сгруппировать несколько словарей вместе, чтобы создать одно обновляемое представление.

Теперь предположим, что у вас есть два (или более) словаря, и вам нужно перебирать их вместе как один. Для этого вы можете создать объект **ChainMap** и инициализировать его своими словарями:

```
>>> from collections import ChainMap
>>> fruit_prices = {'apple': 0.40, 'orange': 0.35}
```

```
>>> vegetable_prices = {'pepper': 0.20, 'onion': 0.55}
>>> chained_dict = ChainMap(fruit_prices, vegetable_prices)
>>> chained_dict # A ChainMap object
ChainMap({'apple': 0.4, 'orange': 0.35}, {'pepper': 0.2, 'onion': 0.55})
>>> for key in chained_dict:
... print(key, '->', chained_dict[key])
...
pepper -> 0.2
orange -> 0.35
onion -> 0.55
apple -> 0.4
```

После импорта **ChainMap** из **collections** вам необходимо создать объект **ChainMap** со словарями, которые вы хотите объединить в цепочку, а затем вы можете свободно перебирать полученный объект, как если бы вы делали это с обычным словарем.

Объекты **ChainMap** также реализуют **.keys()**, **values()** и **.items()**, как это делает стандартный словарь, поэтому вы можете использовать эти методы для итерации по словарному объекту, сгенерированному **ChainMap**, точно так же, как вы это делаете с обычным словарь:

```
>>> for key, value in chained_dict.items():
... print(key, '->', value)
...
apple -> 0.4
pepper -> 0.2
orange -> 0.35
onion -> 0.55
```

В этом случае мы вызвали **.items()** для объекта **ChainMap**. Объект **ChainMap** вел себя так, как будто это был обычный словарь, а **.items()** возвращает объект представления словаря, который можно перебирать как обычно.

????????????? itertools

Iterotools – модуль, который предоставляет некоторые полезные инструменты для выполнения итерационных задач. Давайте посмотрим, как можно использовать некоторые из них для перебора словаря в Python.

??????????? ???????? ? ??????? cycle()

Предположим, вы хотите перебрать словарь в Python, но вам нужно перебирать его несколько раз в одном цикле. Чтобы выполнить эту задачу, вы можете использовать **itertools.cycle(iterable)**, который заставляет итератор возвращать элементы из **iterable** и сохранять копию каждого из них. Когда итерация исчерпана, **cycle()** возвращает элементы из сохраненной копии. Это выполняется циклически, поэтому вы можете остановить цикл.

В следующем примере вы будете перебирать элементы словаря три раза подряд:

```
>>> from itertools import cycle
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> times = 3 # Define how many times you need to iterate through prices
>>> total_items = times * len(prices)
>>> for item in cycle(prices.items()):
... if not total_items:
... break
... total_items -= 1
... print(item)
...
('apple', 0.4)
('orange', 0.35)
('banana', 0.25)
('apple', 0.4)
('orange', 0.35)
('banana', 0.25)
('apple', 0.4)
('orange', 0.35)
('banana', 0.25)
```

Этот код перебирает **prices** определенное количество раз (в данном случае 3). Этот цикл может длиться столько, сколько нужно, но вы должны позаботиться о его остановке. Условие **if** прерывает цикл, когда **total_items** ведет обратный отсчет до нуля.

???????? ? chain()

itertools также предоставляет функцию **chain(*iterables)**, которая получает некоторые итерируемые аргументы в качестве аргументов и создает итератор, который возвращает элементы из итерируемого объекта до тех пор, пока он не будет исчерпан, а затем итерирует по следующему итерируемуго объекту и т. д., пока все они не будут исчерпаны.

Это позволяет вам перебирать несколько словарей в цепочке, как в случае с **collections.ChainMap**:

```
>>> from itertools import chain
>>> fruit_prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> vegetable_prices = {'pepper': 0.20, 'onion': 0.55, 'tomato': 0.42}
>>> for item in chain(fruit_prices.items(), vegetable_prices.items()):
... print(item)
...
('apple', 0.4)
('orange', 0.35)
('banana', 0.25)
('pepper', 0.2)
('onion', 0.55)
('tomato', 0.42)
```

В приведенном выше коде **chain()** вернула итерацию, которая объединила элементы из **fruit_prices** и **vegetable_prices**.

Также возможно использовать **.keys()** или **.values()**, в зависимости от ваших потребностей, с условием быть однородным: если вы используете **.keys()** в качестве аргумента для **chain()** , то вам нужно использовать **.keys( )** для остальных из них.

?????????????? ?????????? ?????????? ??????? (**)

Python 3.5 приносит новую и интересную функцию. PEP 448 – Additional Unpacking Generalizations могут упростить вашу жизнь, когда дело доходит до перебора нескольких словарей в Python. Давайте посмотрим, как это работает, на коротком примере.

Предположим, у вас есть два (или более) словаря, и вам нужно выполнять их итерацию вместе, без использования **collection.ChainMap** или **itertools.chain()**. В этом случае можно использовать оператор распаковки словаря **(**)**, чтобы объединить два словаря в новый и затем выполнить итерацию по нему:

```
>>> fruit_prices = {'apple': 0.40, 'orange': 0.35}
>>> vegetable_prices = {'pepper': 0.20, 'onion': 0.55}
>>> # How to use the unpacking operator **
>>> {**vegetable_prices, **fruit_prices}
{'pepper': 0.2, 'onion': 0.55, 'apple': 0.4, 'orange': 0.35}
```

```
>>> # You can use this feature to iterate through multiple dictionaries
>>> for k, v in {**vegetable_prices, **fruit_prices}.items():
... print(k, '->', v)
...
pepper -> 0.2
onion -> 0.55
apple -> 0.4
orange -> 0.35
```

Оператор распаковки словаря **(**)** действительно замечательная функция в Python. Она позволяет объединить несколько словарей в один новый, как мы это делали в примере с **vegetable_prices** и **fruit_prices**. После объединения словарей с оператором распаковки вы можете перебирать новый словарь как обычно.

Важно отметить, что если словари, которые вы пытаетесь объединить, имеют повторяющиеся или общие ключи, то значения самого послденего словаря будут преобладать:

```
>>> vegetable_prices = {'pepper': 0.20, 'onion': 0.55}
>>> fruit_prices = {'apple': 0.40, 'orange': 0.35, 'pepper': .25}
>>> {**vegetable_prices, **fruit_prices}
{'pepper': 0.25, 'onion': 0.55, 'apple': 0.4, 'orange': 0.35}
```

Ключ **pepper** присутствует в обоих словарях. После объединения их значение **fruit_prices** для **pepper** (0.25) превалирует, потому что **fruit_prices** – самый последний словарь.

??????????

В этой статье мы основы того, как перебирать словарь в Python, а также некоторые более продвинутые методы и стратегии!

Вы узнали:

- Что такое словари, а также некоторые их основные функции и детали реализации
- Каковы основные способы перебора словаря в Python:
- Какие задачи вы можете выполнить, итерируя словарь
- Как использовать некоторые более сложные методы и стратегии для перебора словаря

??????????

Инструменты

IPython

Интерактивная оболочка для языка программирования Python, которая предоставляет расширенную интроспекцию, дополнительный командный синтаксис, подсветку кода и автоматическое дополнение.

Инструменты

Poetry

??? ????? Poetry?

Poetry - это инструмент для управления проектами на Python, который предоставляет следующие возможности:

- управление зависимостями с воспроизводимыми установками и резолвером конфликтов
- автоматическое управление виртуальными окружениями
- сборка и публикация.

??????????

Установка Poetry выполняется очень просто как на Unix-системах:

```
curl -sSL https://install.python-poetry.org | python3 -
```

Так и на Windows:

```
(Invoke-WebRequest -Uri https://install.python-poetry.org -UseBasicParsing).Content | py -
```

Далее, в зависимости от вашей системы, необходимо добавить соответствующую директорию в PATH:

```
# Linux, MacOS, WSL
$HOME/.local/bin

# Windows
%APPDATA%\Python\Scripts
```

Перезагружаем оболочку и проверяем корректность установки:

```
poetry --version
```

```
# Вывод должен быть примерно таким
# Poetry (version 1.4.2)
```

??????????????

???????? ???????

??? ??????? ?????? ????? ????????

Для создания проекта с нуля воспользуемся командой **new**:

```
poetry new my-project
```

При выполнении этой команды Poetry создаст папку со следующей структурой. Наиболее интересен здесь файл **pyproject.toml**, который мы рассмотрим в следующей секции:

```
my-project/
├── README.md
├── my_project
│   └── __init__.py
├── pyproject.toml
└── tests
    └── __init__.py

2 directories, 4 files
```

Если же мы хотим начать использовать Poetry в уже существующем проекте, то нам поможет команда **init**:

```
cd my-project2
poetry init
```

Далее, Poetry задаст нам несколько вопросов о нашем проекте (имя пакета, версия, описание, лицензия и поддерживаемые версии Python), а также предложит в интерактивном режиме указать зависимости (что, как по мне, не очень удобно):

```
This command will guide you through creating your pyproject.toml config.

Package name [my-project2]:
Version [0.1.0]:
```

```
Description []:  My Project, but second
Author [None, n to skip]:  Timur Kasimov
License []:  MIT
Compatible Python versions [^3.8]:

Would you like to define your main dependencies interactively? (yes/no) [yes] no
Would you like to define your development dependencies interactively? (yes/no) [yes] no
Generated file

[tool.poetry]
name = "my-project2"
version = "0.1.0"
description = "My Project, but second"
authors = ["Timur Kasimov"]
license = "MIT"
readme = "README.md"
packages = [{include = "my_project2"}]

[tool.poetry.dependencies]
python = "^3.8"



[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"



Do you confirm generation? (yes/no) [yes] yes
```

?????????? ???????????? ?????????

Команды **new** и **init** не создают виртуальных окружений. При первом выполнении команд, связанных с установкой зависимостей, Poetry создает виртуальное окружение, выбрав базовый интерпретатор по следующей логике:

1. Poetry проверяет, активировано ли уже какое-то виртуальное окружение. Если да, то оно будет использовано
2. Если никакое виртуальное окружение не активировано, то Poetry попытается использовать Python, который был использован при установке Poetry
3. Если версия Python с предыдущего шага несовместима с версией, указанной в **pyproject.toml**, то Poetry попросит явно активировать нужную версию

Советую сразу указать корректный базовый интерпретатор, выполнив в папке проекта следующую команду:

```
poetry env use python3.8  # Если python3.8 есть в PATH
poetry env use /path/to/python  # Можно указать и полный путь
```

Если вы используете pyenv, можно использовать экспериментальную фичу Poetry:

```
poetry config virtualenvs.prefer-active-python true
pyenv install 3.9.8
pyenv local 3.9.8
```

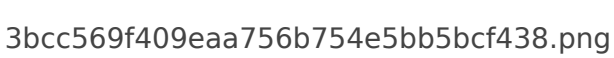
По умолчанию, Poetry создает виртуальные окружения в папке **{cache_dir}/virtualenvs**. Если вы хотите, чтобы виртуальное окружение находилось в папке проекта, можно выполнить следующую команду:

```
poetry config virtualenvs.in-project true
```

Теперь создаваемое виртуальное окружение будет находиться в папке **.venv** в корне проекта.

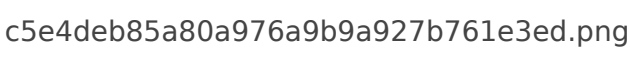
??? ??????? ?????? ????? PyCharm

PyCharm поддерживает интеграцию с Poetry. Можно выбрать Poetry как при создании нового проекта:



3bcc569f409eaa756b754e5bb5bcf438.png

Так и в существующем проекте. Для этого необходимо в правом нижнем углу нажать "No Interpreter" (если у вас до этого не был настроен интерпретатор в проекте) или на имя интерпретатора, далее "Add New Interpreter" → "Add Local Interpreter", и в открывшемся окне выбрать "Poetry environment":



c5e4deb85a80a976a9b9a927b761e3ed.png

pyproject.toml ? poetry.lock

pyproject.toml - это файл, который содержит в себе всю информацию о проекте: метаданные (имя, версия и т.п.) и зависимости, а также в нем могут присутствовать настройки других инструментов.

Файл **poetry.lock** же содержит в себе все зависимости проекта с зафиксированными версиями и формируется автоматически (пожалуйста, не редактируйте его вручную).

[tool.poetry] содержит в себе метаданные:

- name - имя проекта. Должно быть валидным по PEP 508;
- version - версия проекта. Должна быть валидной по PEP 440;
- description - короткое описание проекта;
- license - лицензия;
- authors - авторы проекта в формате "name <email>". Должен присутствовать как минимум один автор.

Остальные спецификаторы можно найти в документации, они не являются обязательными.

[tool.poetry.dependencies] содержит в себе версию Python и основные зависимости проекта (так называемую main-группу).

В PEP-517 был представлен стандартный способ определять альтернативные системы сборки для Python-проектов. Poetry совместим с PEP-517 и использует poetry-core для сборки, что и обозначено в секции **build-system**:

```
[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

Многие питоновские инструменты поддерживают конфигурацию через **pyproject.toml**. Например, в моих проектах в нем уютно расположились настройки isort и mypy:

```
[tool.isort]
line_length = 120
...
force_grid_wrap = 2

[tool.mypy]
python_version = 3.9
...
follow_imports = "skip"
```

????????? ? ???????? ???????

??????????? ????????? + ???????????????

Команда add добавляет зависимость в **pyproject.toml**, выполняет разрешение зависимостей и устанавливает зависимость:

```
poetry add fastapi
```

В секции **[tool.poetry.dependencies]** появилась наша зависимость:

```
[tool.poetry.dependencies]
python = "^3.8"
fastapi = "^0.95.2"
```

Рассмотрим, как указывать версии при установке. Первый вариант - это Caret Requirements:

Требование	Допустимые версии
^1.2.3	>=1.2.3, <2.0.0
^1.2	>=1.2.0, <2.0.0
^1	>=1.0.0, <2.0.0
^0.2.3	>=0.2.3, <0.3.0
^0.0.3	>=0.0.3, <0.0.4
^0.0	>=0.0.0, <0.1.0
^0	>=0.0.0, <1.0.0

Второй вариант - это Tilde requirements, что позволяет указывать минимальную допустимую версию с некоторой возможностью обновления:

Требование	Допустимые версии
~1.2.3	>=1.2.3,<1.3.0
~1.2	>=1.2.0,<1.3.0
~1	>=1.0.0,<2.0.0

Wildcard requirements позволяют обновление до последней версии в позиции, где расположен символ "*":

Требование	Допустимые версии
*	>=0.0.0
1.*	>=1.0.0,<2.0.0
1.2.*	>=1.2.0,<1.3.0

Inequality requirements позволяют вручную указать диапазон допустимых версий, или же точную версию:

```
>=1.2.0
>1
```

```
<2
!=1.2.3
==1.5.2

# Можно комбинировать
>=1.2,<1.5
```

Стоит обратить внимание на следующий момент: если вы указываете точную версию, и другим зависимостям требуется другая версия, то резолвер не сможет разрешить все конфликты и процедура установки (или обновления) не будет выполнена.

Extras ? groups

Прежде чем идти дальше, хочется ввести 2 понятия - extras и groups. Они довольно схожи, но на деле служат для разных целей.

Dependency groups (далее просто группы) содержат в себе опциональные зависимости, используемые **только при разработке.** Установить зависимости из групп можно только через Poetry. Каждый проект содержит в себе одну неявную обязательную группу - main, которая находится в секции **[tool.poetry.dependencies]**.

Установка в группу выполняется следующим образом:

```
poetry add --group test pytest
```

Группа вместе с зависимостью появилась в pyproject.toml:

```
[tool.poetry.group.test.dependencies]
pytest = "^7.3.1"
```

Extras же предназначены для введения дополнительных зависимостей, которые включают какую-либо функциональность в вашем проекте. Установить зависимость как extra можно вот так:

```
poetry add --extras postgres psycopg2-binary
poetry add --extras mysql --extras database mysql-connector-python  # Можно перечислять несколько extras
```

Зависимости появились в **[tool.poetry.dependencies]** с пометкой extras:

```
[tool.poetry.dependencies]
python = "^3.8"
psycopg2-binary = {version = "^2.9.6", extras = ["postgres"]}
```

```
mysql-connector-python = {version = "^8.0.33", extras = ["mysql", "database"]}
```

После сборки и публикации можно будет устанавливать ваш пакет как обычно:

```
poetry add "my-project[postgres,database]"
poetry add "my-project[mysql]
```

????????? ? Git ? ????????? package registries

Установка из публичных репозиториев выполняется следующим образом:

```
poetry add "git+https://github.com/psf/requests"  # будет использован последний коммит с основной ветки
poetry add "git+https://github.com/psf/requests#update-3.0"  # будет использован последний коммит с ветки update 3.0

# Все то же самое, но через SSH
poetry add "git+ssh://git@github.com:requests/requests.git"
poetry add "git+ssh://git@github.com:requests/requests.git#update-3.0"
```

Для установки из приватных git-репозиториев можно воспользоваться SSH, но в таком случае будет выполняться сборка.

Если же у вас есть приватный package registry, последовательность действия следующая (рассматривать будем на примере GitLab). Добавляем источник:

```
poetry source add my-repo "https://my.gitlab.com/projects/1/packages/pypi/simple"
```

Добавленный источник появился в файле **pyproject.toml**:

```
[[tool.poetry.source]]
name = "my-repo"
url = "https://my.gitlab.com/projects/1/packages/pypi/simple"
default = false
secondary = false
```

Настраиваем аутентификацию (для GitLab Package Registry рекомендую использовать Personal Access Token):

```
poetry config http-basic.my-repo <token-name> <secret-token>
```

Выполняем установку, указав источник:

```
poetry add --source my-repo my-package
```

Не забудьте изменить my.gitlab.com на адрес вашего GitLab, а также указать корректный ID проекта. Имя источника в примере используется my-repo, но можно выбрать любое другое на ваше усмотрение.

????????? ?? ??????

Poetry позволяет устанавливать зависимости как из локальных файлов и папок:

```
poetry add package-1.0.0.tar.gz
poetry add package-1.0.0.whl
poetry add ~/my/local/package
```

Так и с удаленных серверов:

```
poetry add https://download.pytorch.org/whl/cpu/torch-2.0.0%2Bcpu-cp39-cp39-linux_x86_64.whl
```

????? ??? ?????????

В этом разделе кратко пробежимся по основным опциям, которые можно использовать при установке зависимостей:

Опция	Пояснение
--group (-G)	Группа, в которую необходимо добавить зависимость. Если такой группы не существует, она будет создана
--editable	Установить зависимость в editable режиме
--extras	Extras для активации зависимости
--optional	Добавить зависимость как опциональную
--python	Версия Python, для которой зависимость должна быть установлена
--platform	Платформа, для которой зависимость должна быть установлена (linux, darwin или win32)
--source	Имя источника, из которого будет установлена зависимость. По умолчанию используется PyPI, о настройке собственных источников ниже
--allow-prereleases	Разрешить установку пререлизов
--dry-run	Вывести последовательность действий, но не выполнять никаких операций
--lock	Не выполнять установку, только обновить lock-файл

poetry install

Команда install при запуске выполняет следующую последовательность действий:

- читает файл **pyproject.toml**
- если существует файл **poetry.lock**, то версии зависимостей берутся из него. Если его не существует, Poetry выполнит разрешение зависимостей и создаст его
- устанавливает зависимости

Рассмотрим основные опции:

Опция	Пояснение
--without	Группы, которые будут проигнорированы при установке
--with	Группы, которые будут установлены
--only	Установить только определенные группы (в этом случае --without и --with будут проигнорированы)
--only-root	Установить только проект, проигнорировав все зависимости
--sync	Синхронизировать виртуальное окружение с lock-файлом
--no-root	Не устанавливать сам проект
--dry-run	Вывести последовательность действий, но не выполнять никаких операций
--extras (-E)	Extras, которые необходимо установить
--all-extras	Включить все extras в установку
--compile	Транслировать исходники в байт-код

???????? ???????

Для удаления какой-либо зависимости можно воспользоваться командой remove:

```
poetry remove requests
```

Если зависимость находится в какой-то группе, используйте опцию --group:

```
poetry remove --group my-group requests
```

???????? ????????????

Команда lock позволяет зафиксировать зависимости, обновив файл **poetry.lock**:

```
poetry lock
```

Будьте внимательны! По умолчанию, **poetry lock** попытается выполнить обновление всех зависимостей до последних допустимых версий. Чтобы этого избежать, используйте опцию --no-update.

?????? ?????? ????? Poetry

poetry shell

С помощью **poetry shell** можно запустить оболочку с активированным виртуальным окружением. Если его не существует, то оно будет создано.

Т.к. **poetry shell** не просто активирует виртуальное окружение, а именно создает оболочку, то стоит использовать для выхода **exit**, а не **deactivate**.

??????? ? pyproject.toml

В файл **pyproject.toml** можно включить секцию **[tool.poetry.scripts]**, которая содержит в себе описание скриптов, которые будут доступны к использованию при установке проекта:

```
[tool.poetry.scripts]
my-script = "my_package.console:run"
```

Здесь мы описываем скрипт my-script, при запуске которого выполнится функция run из модуля console из пакета my_package. При обновлении или добавлении скриптов не забывайте выполнять команду **poetry install**, чтобы сделать их доступными в виртуальном окружении проекта.

poetry run

Команда **run** позволяет запускать команды в виртуальном окружении проекта. Например:

```
poetry run python --version
```

Что более интересно, с помощью **run** можно запускать скрипты, определенные в **pyproject.toml**:

```
poetry run my-script
```

?????? ? ?????????? ???????

Представим, что вы разрабатываете какую-то библиотеку, и настал торжественный момент сборки и публикации. Используя Poetry, сделать это можно вот так.

Собираем наш пакет:

```
poetry build  # собираем как sdist, так и wheel
poetry build --format sdist  # собираем только sdist
poetry build --format wheel  # собираем только wheel
```

Для публикации на PyPI предварительно получаем API token и устанавливаем его:

```
poetry config pypi-token.pypi <my-token>
```

И, наконец, публикуем:

```
poetry publish
```

Если же необходимо опубликовать пакет в приватный registry (например, в GitLab), то настройка репозитория немного усложняется. Предварительно не забываем сгенерировать Personal Access Token:

```
poetry config repositories.gitlab "my.gitlab.com/projects/1/packages/pypi"
poetry config http-basic.gitlab <token-name> <secret-token>  # здесь token-name и secret-token
- ваш Personal Access Token
```

Не забудьте изменить my.gitlab.com на адрес вашего GitLab, а также указать корректный ID проекта. Здесь имя репозитория в Poetry выбрано gitlab, можно использовать другое на ваше усмотрение. Пора публиковать:

```
poetry publish --repository gitlab
```

В качестве бонуса - пример публикации при использовании GitLab CI:

```
build_and_publish:
  stage: build_and_publish
  script:
    - poetry install --without dev
    - poetry build
    - poetry config repositories.gitlab
"${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/pypi"
    - poetry config http-basic.gitlab gitlab-ci-token "$CI_JOB_TOKEN"
```

```
    - poetry publish --repository gitlab
```

????????? poetry

В этом разделе кратко пробежимся по основным командам для настройки и управления непосредственно Poetry.

?????????? Poetry

Обновиться можно как на более новую, так и на более старую версию. **Будьте аккуратны** - если обновиться на версию ниже 1.2, то обновление на более высокую версию будет невозможно и придется переустанавливать Poetry:

```
poetry self update 1.3.2
```

?????????? ?????????

Для установки, удаления и обновления плагинов можно пользоваться командами **self add**, **self remove** и **self update** соответственно. Механизм работы этих команд аналогичен механизму работы команд **add**, **remove**, **update** за исключением того, что команды в пространстве имен **self** выполняется в виртуальном окружении самого Poetry.

При управлении плагинами нам также доступны команды **self lock** и **self install**. Работают они аналогично вышеупомянутым командам.

????????? Poetry

Команда **config**, помимо управления репозиториями, позволяет редактировать настройки Poetry:

```
# [setting-key] - имя настройки
# [setting-value] - значение
poetry config [options] [setting-key] [setting-value1] ... [setting-valueN]
```

Чаще всего используются следующие опции:

- **cache-dir** (строка) - директория для кэша Poetry
- **virtualenvs.create** (true / false) - создавать ли виртуальные окружения для проектов (если не существуют). Будьте внимательны: если вы установите данную настройку в false и Poetry не обнаружит виртуальное окружение в папках {cache-dir}/virtualenvs или {project-dir}/.venv, то установка зависимостей будет выполняться в системный Python.
- **virtualenvs.in-project** (true / false) - создавать виртуальные окружения в корне проекта. По умолчанию, Poetry создает виртуальные окружения в папке {cache-dir}/virtualenvs.

Остальные опции можно найти в документации.

????? - Docker-????? ? ???????? ? ????????????? Poetry

Давайте рассмотрим, как собрать максимально компактный Docker-образ, если разработку нашего проекта мы вели с помощью Poetry. Имеем следующий **pyproject.toml**:

```
[tool.poetry]
name = "my-project"
version = "0.1.0"
description = ""
authors = ["Your Name <you@example.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = ">=3.8.1,<4.0"
fastapi = "^0.95.2"
sqlalchemy = "^2.0.15"
uvicorn = "^0.22.0"

[tool.poetry.group.test.dependencies]
pytest = "^7.3.1"



[tool.poetry.group.dev.dependencies]
flake8 = "^6.0.0"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Наше приложение - это вот такое замечательное REST API:

```
from fastapi import FastAPI


app = FastAPI()
```

```
@app.get('/hello')
def hello():
    return 'hello, world!'
```

Первый способ, который сразу приходит в голову - установим Poetry в образ, через него поставим зависимости и запустим наш сервис. Dockerfile в таком случае выглядит примерно вот так:

```
FROM python:3.8.16-slim-bullseye

WORKDIR /app

COPY poetry.lock pyproject.toml ./

RUN python -m pip install --no-cache-dir poetry==1.4.2 \
    && poetry config virtualenvs.create false \
    && poetry install --without dev,test --no-interaction --no-ansi \
    && rm -rf $(poetry config cache-dir)/{cache,artifacts}

COPY app.py ./

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "80"]
```

Собираем образ:

```
docker image build -t poetry-tutorial-naive:latest -f Dockerfile.naive .
```

Получаем образ размером 225MB. Неплохо, но можно и меньше. Давайте попробуем применить multi-stage сборку. На первом этапе с помощью Poetry подготовим виртуальное окружение, а потом скопируем его в наш итоговый образ. Итак, Dockerfile:

```
FROM python:3.8.16-slim-bullseye AS builder

WORKDIR /app
COPY poetry.lock pyproject.toml ./

RUN python -m pip install --no-cache-dir poetry==1.4.2 \
    && poetry config virtualenvs.in-project true \
    && poetry install --without dev,test --no-interaction --no-ansi

FROM python:3.8.16-slim-bullseye
```

```
COPY --from=builder /app /app
COPY app.py ./

CMD ["/app/.venv/bin/uvicorn", "app:app", "--host", "0.0.0.0", "--port", "80"]
```

Собираем:

```
docker image build -t poetry-tutorial-copy:latest -f Dockerfile.copy .
```

Получаем образ размером 159MB. Уже гораздо лучше!

Может быть, если экспортировать все зависимости в привычный requirements.txt и установить через pip, то получится еще компактнее? Попробуем:

```
FROM python:3.8.16-slim-bullseye AS builder

COPY poetry.lock pyproject.toml ./
RUN python -m pip install --no-cache-dir poetry==1.4.2 \
    && poetry export --without-hashes --without dev,test -f requirements.txt -o
requirements.txt

FROM python:3.8.16-slim-bullseye

WORKDIR /app

COPY --from=builder requirements.txt ./
RUN python -m pip install --no-cache-dir -r requirements.txt

COPY app.py ./

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "80"]
```

Собираем образ:

```
docker image build --no-cache -t poetry-tutorial-requirements:latest -f
Dockerfile.requirements
```

Получилось 163MB, что немного больше, чем у предыдущего способа, который и вышел победителем.

Итоговая таблица:

Способ	Размер образа
Нативный	225MB (-0.0%)
Экспорт requirements.txt и установка через pip	163MB (-27.5%)
Копирование venv	159MB (-29.3%)

Обратите внимание на следующие вещи:

- жестко фиксируйте версию Poetry в ваших Dockerfile'ах. Разработчики Poetry очень любят что-то ломать от релиза к релизу, или объявлять ставшие привычными вещи deprecated.
- не тащите лишние зависимости в ваши образы. Используйте флаг --without.