Основы Python

Урок 8. Регулярные выражения и декораторы в Python

## На этом уроке

1. Поработаем с регулярными выражениями.
2. Узнаем, что такое декоратор в Python.
3. Научимся писать собственные декораторы.

## Оглавление

[Регулярные выражения: примеры](#_fjuessrof5va)

[Валидация](#_nvtanxc9802o)

[Парсинг](#_je2u2etaix6n)

[Декораторы: зачем](#_20itwrs53eel)

[Декоратор без аргументов](#_a4ci9aaicn2l)

[\*Декоратор с аргументами](#_w6rfvkogmtk1)

[Подведение итогов](#_a68uvyn2ozze)

[Практическое задание](#_4i7ojhp)

[Дополнительные материалы](#_2xcytpi)

[Используемая литература](#_1ci93xb)

# 

# Регулярные выражения: примеры

## Валидация

Наступил момент, когда вы готовы прикоснуться к чему-то серьёзному. Следует осознавать, что работа с [регулярными выражениями](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D1%8F) требует больших усилий. Зато она развивает абстрактное мышление и существенно повышает вашу ценность как потенциального сотрудника.

Решим простую задачу валидации данных: пользователь должен ввести своё имя с большой буквы, допустимы только русские буквы. Попробуем вариант с множеством:

| valid\_letters = {chr(sym\_code) for sym\_code in range(ord('а'), ord('я') + 1)} valid\_letters.add('ё')   def name\_is\_valid(name):  if not name or set(name.lower()) - set(valid\_letters):  return False  return name.istitle()   if \_\_name\_\_ == '\_\_main\_\_':  while True:  name = input('Введите имя:\n')  if name\_is\_valid(name):  break  print(f'пользователь: {name}') |
| --- |

Можно было решить и по-другому — проверить коды букв имени, но код стал бы сложнее, ведь кроме промежутка букв от «а» до «я» пришлось бы отдельно проверять букву «ё».

Теперь решим задачу при помощи регулярного выражения и модуля [re](https://docs.python.org/3.8/library/re.html):

| import re  RE\_NAME = re.compile(r'^[А-ЯЁ][а-яё]+$')   def name\_is\_valid(name):  return RE\_NAME.match(name) ... |
| --- |

Фактически решение в одну строку. Мы только добавили небольшую оптимизацию — скомпилировали регулярное выражение заранее, при большом количестве вызовов функции будет серьёзный рост скорости.

**Замечание:** обычно паттерн регулярного выражения пишем с префиксом “r”, чтобы интерпретатор воспринимал его как сырую строку.

Попробуем написать «регулярку» для валидации даты в формате «ДД.ММ.ГГГГ»:

| RE\_DATE = re.compile(r'^(\d{2}\.){2}\d{4}$') |
| --- |

Обратите внимание, что мы не проверяем корректность самой даты (например, 32 число или 30 февраля) — только соответствие формату. Как вы думаете, что изменится, если убрать экранирование перед “.” в паттерне? Тут начинается коварство: валидные даты так и останутся валидными, но теперь валидацию будут проходить даты с любым разделителем: и запятая, и дефис, любой символ будет валиден. Напишем простой тест:

| import re  RE\_DATE = re.compile(r'^(\d{2}.){2}\d{4}$')  for date in ['23.01.2021', '23,01,2021', '23~01~2021']:  assert RE\_DATE.match(date), f'wrong date {date}' |
| --- |

Он завершается без ошибок — значит, все даты прошли валидацию.

**Замечание:** здесь мы использовали ключевое слово [assert](https://docs.python.org/3/reference/simple_stmts.html?highlight=assert#the-assert-statement) для проверки верности выражения. Если выражение после этого слова не True, то поднимается исключение [AssertionError](https://docs.python.org/3/library/exceptions.html#AssertionError) и выполнение кода прерывается. Если через запятую написать сообщение, оно появится в консоли. Если для вас это кажется сложным, просто напишите в цикле “print(RE\_DATE.match(date))” — увидите, что регулярка сработала на все даты.

Теперь с экранированием:

| import re  RE\_DATE = re.compile(r'^(\d{2}\.){2}\d{4}$')  for date in ['23.01.2021', '23,01,2021', '23~01~2021']:  assert RE\_DATE.match(date), f'wrong date {date}' *# ...AssertionError: wrong date 23,01,2021* |
| --- |

**Вывод:** любое регулярное выражение надо стараться тестировать. Очень часто ошибка бывает не очевидна.

**Примечание:** в этой регулярке мы использовали группировку: так как в дате два раза повторяется паттерн “<число><число><разделитель>”, поймали его как группу “(\d{2}\.)” и взяли её дважды “{2}”.

А если понадобится искать даты с другими разделителями: вида “23-01-2021” или “23/01/2021”? Если мы используем регулярное выражение, решение очень простое:

| RE\_DATE = re.compile(r'^(\d{2}[./-]){2}\d{4}$') |
| --- |

**Важно:** дефис должен быть последним внутри квадратных скобок или его нужно экранировать! Обязательно проверьте этот нюанс.

## Парсинг

Попробуем при помощи регулярного выражения найти все даты в тексте:

| import re  RE\_DATE = re.compile(r'(?:\d{2}[./-]){2}\d{4}')  txt = 'Погода 23.01.2021 была отличная! Зато за день до этого (22/01/2021) - очень холодно. ' \  'Надеемся, что 24-01-2021 будет без ветра.'  print(RE\_DATE.findall(txt)) *# ['23.01.2021', '22/01/2021', '24-01-2021']* |
| --- |

Регулярку пришлось переписать: убрали спецсимволы начала строки ^ и конца строки $ — выражение стало менее строгим. Кроме этого использовали спецсимвол ?: в группе, чтобы она не захватывалась (обязательно посмотрите, какой результат будет без этого спецсимвола).

\*Теперь решим задачу посложнее: есть текст с данными о товарах пользователя. Известно, что название товара обёрнуто в кавычки, а его цена идет после названия в круглых скобках (между ними может быть пробел, табуляция или даже несколько пробелов). Нужно получить список кортежей вида (<название товара>, <цена>):

| import re  RE\_PRODUCTS = re.compile(r'"([^"]+)"\s\*\((\d+(?:[,.]\d+)\*).\*\)')  txt = ''' Иван сегодня сделал заказ: "iPhone 12" (158900,6 руб), "Galaxy S21"(98653.7 р). Позже он добавил в корзину "iPad"\t(32451) '''  print(RE\_PRODUCTS.findall(txt)) *# [('iPhone 12', '158900,6'), ('Galaxy S21', '98653.7'), ('iPad', '32451')]* |
| --- |

В этом регулярном выражении важна каждая деталь. Даже небольшие изменения приведут к «поломке». Что мы ищем между кавычек? Всё, кроме кавычки. Разумеется, обособляем в группу. Дальше допускаем ноль или несколько пробельных символов и ищем что-то между круглыми скобками — их экранируем. Так как после цены товара может быть какой-то текст (например, валюта), дописали “.\*”. Особое внимание обращаем на то, как мы перехватываем цену: это может быть целое число, а может быть вещественное: “(?:[,.]\d+)\*” — поймает дробную часть, если она будет. Тут снова видим спецсимвол ?: — попробуйте без него, результат изменится.

Следующая задача: найти в тексте слова, которые начинаются и заканчиваются на одну и ту же букву:

| import re  RE\_EQ\_LETTERS = re.compile(r'\b(\w)\w\*\1\b')  txt = 'Однако, хорошо у вас получилось. А как еще могло быть?'  print(RE\_EQ\_LETTERS.findall(txt)) print(RE\_EQ\_LETTERS.search(txt)) print(RE\_EQ\_LETTERS.match(txt)) print(\*RE\_EQ\_LETTERS.finditer(txt)) *# ['к', 'е'] # <re.Match object; span=(35, 38), match='как'> # None # <re.Match object; span=(35, 38), match='как'> <re.Match object; span=(39,  # 42), match='еще'>* |
| --- |

Обособляем первую букву слова в первую группу и проверяем ее наличие при помощи \1 в конце слова. Границы слова ловим при помощи спецсимвола \b.

В этом примере хорошо видна разница между разными методами объекта [Regular Expression Object](https://docs.python.org/3.8/library/re.html#regular-expression-objects):

* [.findall()](https://docs.python.org/3.8/library/re.html#re.Pattern.findall) — возвращает список совпадений с паттерном, если есть группы — возвращает их (как в нашем случае, вернул первые буквы слов);
* [.search()](https://docs.python.org/3.8/library/re.html#re.Pattern.search) — находит первое совпадение с паттерном и возвращает [Match Object](https://docs.python.org/3.8/library/re.html#match-objects);
* [.match()](https://docs.python.org/3.8/library/re.html#re.Pattern.match) — работает аналогично .search(), но ищет с начала строки, а не по всей строке (очень часто их [путают](https://docs.python.org/3.8/library/re.html#search-vs-match), в нашем случае ничего не нашёл);
* [.finditer()](https://docs.python.org/3.8/library/re.html#re.Pattern.finditer) — по сути, работает как .search(), но в режиме итератора возвращает Match Object для всех совпадений (именно он нашёл все слова).

А если по заданию нужно будет искать без учёта регистра? Тут начинаются особенности реализации: в Python мы можем передавать флаги функции re.compile():

| import re  RE\_EQ\_LETTERS = re.compile(r'\b(\w)\w\*\1\b', re.IGNORECASE)  txt = 'Однако, хорошо у вас получилось. А как еще могло быть?'  print(\*map(lambda x: x.group(0), RE\_EQ\_LETTERS.finditer(txt)), sep=', ') *# Однако, как, еще* |
| --- |

У флага [re.IGNORECASE](https://docs.python.org/3.8/library/re.html#re.IGNORECASE) говорящее имя. При обработке многострочных текстов будет полезен флаг [re.MULTILINE](https://docs.python.org/3.8/library/re.html#re.MULTILINE).

\*Последний, самый сложный пример: распарсить GET-данные в URL адресе в именованные группы:

| import re  RE\_GET\_PARSER = re.compile(r'(?<=[&?])(?P<key>[^&]+)=(?P<val>[^&]+)(?=&\*)')  url = 'https://translate.google.com/?hl=ru&sl=en&tl=ru&text=go&op=translate'  print(\*map(lambda x: x.groupdict(), RE\_GET\_PARSER.finditer(url)), sep=', ') *# {'key': 'hl', 'val': 'ru'}, {'key': 'sl', 'val': 'en'},*  *# {'key': 'tl', 'val': 'ru'}, {'key': 'text', 'val': 'go'},*  *# {'key': 'op', 'val': 'translate'}* |
| --- |

**Первый важный** момент: использовали здесь lookahead (взгляд вперёд) и lookbehind (взгляд назад). То есть группа обособляется, если после неё или до неё есть определенный паттерн (в нашем случае до группы [&?], а после — &\*). Попробуйте разные варианты этой регулярки — лучший способ понять, «как это работает».

**Второй** момент: обращаемся к методу [.groupdict()](https://docs.python.org/3.8/library/re.html#re.Match.groupdict) объекта Match Object вместо метода [.group()](https://docs.python.org/3.8/library/re.html#re.Match.group).

На этом знакомство с регулярными выражениями заканчиваем: на самом деле нужно много практики, чтобы чувствовать себя комфортно в этой области.

**Примечание:** для проверки шаблонов можно использовать дополнительные плагины к среде разработки или онлайн-инструменты, например, <https://regex101.com/>.

# Декораторы: зачем

Если посмотреть [определение](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), то задача декоратора — реализация дополнительного поведения объекта.

Например, у нас была функция обработки запроса к серверу и в какой-то момент понадобилось ограничить доступ только авторизованным пользователям. Можно, конечно, внутри самой функции добавить проверки. А если это библиотека или фреймворк и ограничение доступа опционально? Можно сделать две функции: одну для всех, другую с проверкой авторизации. Но это разрастание кодовой базы и проблемы с её поддержанием в будущем. Плюс нарушаем самый главный принцип: DRY. Можно сделать одну функцию, но с дополнительным флагом и проверять факт авторизованности пользователя, если он установлен: тоже решение «так себе» — нагружаем функцию избыточным кодом, который не всегда будет работать. Использование декоратора тут будет идеальным вариантом — добавим за счёт него проверку авторизации (именно так сделано во фреймворке Django).

Ещё примеры: кеширование и логирование. То есть если мы не хотим менять готовую реализацию какого-то функционала в коде, которая скорее всего хорошо протестирована и исправно работает, а хотим добавить что-то (возможно, временное) к этой реализации, «нам нужен декоратор». Иногда ещё говорят «обернуть» декоратором. То есть если что-то и сломается, круг поиска будет существенно уже, чем если бы мы меняли основной код.

Попробуем начать с простого примера. Пусть есть функция, генерирующая HTML-разметку для тега <input> (без тега <label> для упрощения):

| def render\_input(field):  return f'<input id="id\_{field}" type="text" name="{field}">'   username\_f = render\_input('username') print(username\_f) *# <input id="id\_username" type="text" name="username">* |
| --- |

Предположим, что нам понадобилось обернуть этот тег в другой, например <p> или <li>. Попробуем написать функцию-обёртку:

| def p\_wrapper(func):  print(func)   def tag\_wrapper(\*args, \*\*kwargs):  print('args', args)  print('kwargs', kwargs)  markup = func(\*args, \*\*kwargs)  print(markup)  return markup   return tag\_wrapper   @p\_wrapper def render\_input(field):  return f'<input id="id\_{field}" type="text" name="{field}">'   username\_f = render\_input('username') print(render\_input) *# <function render\_input at 0x00000000025FB6A8> # args ('username',) # kwargs {} # <input id="id\_username" type="text" name="username"> # <function p\_wrapper.<locals>.tag\_wrapper at 0x00000000027062F0>* |
| --- |

**Факт:** синтаксически любая функция, написанная с префиксом @ над другой функцией, становится декоратором.

Разумеется, функция должна быть особым образом написана, чтобы реально выполнять роль декоратора.

**Первое:** её аргументом является оборачиваемая функция (интересующимся рекомендуем почитать про [функции первого класса](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0)).

**Второе:** внутри неё должна быть ещё одна функция, принимающая аргументы оборачиваемой функции. Это и есть самый сложный для понимания момент.

Внимательно изучите результат работы этого скрипта. Подумайте, как аргументы оборачиваемой функции попадают внутрь? Подсказка: что по факту мы вызываем, когда пишем render\_input('username')?

Правильный ответ вы уже видите в консоли: p\_wrapper.<locals>.tag\_wrapper. Ведь декоратор возвращает не результат вызова, а callback — внутреннюю функцию tag\_wrapper. То есть вызов render\_input('username') превращается в вызов tag\_wrapper('username'). Теперь вопрос с аргументами должен быть снят. Если это не так, думайте ещё. Дальше нельзя.

Внутри tag\_wrapper можем делать всё что угодно с аргументами оборачиваемой функции — можем их валидировать, преобразовывать, просто логировать. То же самое, и даже больше, можем делать с результатом вызова оборачиваемой функции — например, можем его закешировать. Но давайте для начала закончим этот простой пример:

| def p\_wrapper(func):  def tag\_wrapper(\*args, \*\*kwargs):  markup = func(\*args, \*\*kwargs)  return f'<p>{markup}</p>'   return tag\_wrapper   @p\_wrapper def render\_input(field):  return f'<input id="id\_{field}" type="text" name="{field}">'   username\_f = render\_input('username') print(username\_f) *# <p><input id="id\_username" type="text" name="username"></p>* |
| --- |

**Замечание:** \*\*kwargs здесь для того, чтобы избежать проблем в будущем, если у оборачиваемой функции появятся именованные аргументы.

Всё работает: без декоратора получаем оригинальное значение функции, с декоратором — обёрнутое в тег <p>.

# Декоратор без аргументов

Давайте теперь реализуем декоратор поинтереснее - сделаем кеш на основе словаря:

| def simple\_cache(func):  cache = {}   def wrapper(\*args):  nonlocal cache  key = str(\*args)  if key not in cache:  cache[key] = func(\*args)  return cache[key]   return wrapper   @simple\_cache def render\_input(field):  print(f"call render\_input('{field}')")  return f'<input id="id\_{field}" type="text" name="{field}">'   username\_f = render\_input('username') password\_f = render\_input('password') username\_f\_2 = render\_input('username') print(username\_f) print(password\_f) print(username\_f\_2) *# call render\_input('username')* *# call render\_input('password')* *# <input id="id\_username" type="text" name="username">* *# <input id="id\_password" type="text" name="password">* *# <input id="id\_username" type="text" name="username">* |
| --- |

Кеш работает — при повторном вызове render\_input('username') реально функция render\_input() не вызывалась: данные взяли из кеша.

Без нюансов в этом коде не обошлось.

**Первый нюанс:** ключ словаря должен быть immutable (неизменяемым), поэтому обернули аргументы в строку. Если вы точно уверены, что аргументы будут [хешируемыми](https://stackoverflow.com/questions/14535730/what-does-hashable-mean-in-python), можно этого не делать. Также можно было бы написать свою функцию для хеширования аргументов.

**Второй нюанс:** использовали ключевое слово [nonlocal](https://docs.python.org/3.8/reference/simple_stmts.html?highlight=nonlocal#nonlocal). Таким образом мы обращаемся к «замороженной» во внешней по отношению к обертке wrapper() области видимости переменной cache. Это очень важно! Ведь если бы мы хранили cache в wrapper(), она создавалась бы каждый раз заново и ничего не получилось бы запомнить. Функция simple\_cache() фактически вызывается один раз для каждой декорируемой функции — это происходит в момент инициализации модуля. Именно в этот момент создается переменная cache и продолжает существовать в памяти всё время выполнения скрипта (\*такой прием используют в [замыканиях](https://realpython.com/inner-functions-what-are-they-good-for/#whats-a-closure)). Давайте проверим это при помощи скрипта:

| from time import perf\_counter  print(f'{perf\_counter()}: script started')   def simple\_cache(func):  cache = {}  print(f'{perf\_counter()}: {func.\_\_name\_\_} cache created ({id(cache)})')   def wrapper(\*args):  nonlocal cache  key = tuple(args)  if key not in cache:  cache[key] = func(\*args)  return cache[key]   return wrapper   @simple\_cache def calc\_sum(x, y):  return x + y   @simple\_cache def calc\_mul(x, y):  return x \* y  *# 0.436655291: script started* *# 0.436687107: calc\_sum cache created (40674936)* *# 0.436700962: calc\_mul cache created (61166504)* |
| --- |

Видим, что сразу после старта скрипта создалось два объекта словаря: отдельно для кеширования каждой из функций.

**Третий нюанс:** важно понимать, что со временем в кеше может накопиться много объектов, поэтому для реального проекта нужно предусмотреть ограничение его размера через механизм [периодической очистки](https://en.wikipedia.org/wiki/Cache_replacement_policies#Examples).

**\*Примечание:** очень часто на собеседованиях дают задание реализовать [lru\_cache](https://docs.python.org/3.8/library/functools.html#functools.lru_cache) — обязательно изучите эту тему в будущем.

Важной особенностью Python является возможность применения нескольких декораторов к функции. При этом важен их порядок. Попробуем параллельно с кешированием реализовать простое логирование в консоли — вынесем его из функции render\_input():

| def simple\_cache(func):  cache = {}   def wrapper(\*args):  nonlocal cache  key = str(\*args)  if key not in cache:  cache[key] = func(\*args)  return cache[key]   return wrapper   def logger(func):  def wrapper(\*args):  result = func(\*args)  print(f'\tcall {func.\_\_name\_\_}({", ".join(map(str, args))})')  return result   return wrapper   @simple\_cache @logger def render\_input(field):  return f'<input id="id\_{field}" type="text" name="{field}">'   username\_f = render\_input('username') password\_f = render\_input('password') username\_f\_2 = render\_input('username') print(username\_f) print(password\_f) print(username\_f\_2) *# call render\_input(username) # call render\_input(password) # <input id="id\_username" type="text" name="username"> # <input id="id\_password" type="text" name="password"> # <input id="id\_username" type="text" name="username">* |
| --- |

Теперь всё намного лучше: мы разгрузили функцию render\_input() от решения «лишней» задачи — логирования. Причем можем в любой момент отключить и логирование, и кеш — всё будет работать. Осталось ответить на один **важный вопрос**: имеет ли значение порядок декораторов? Сейчас видим, что первым сработал логгер. Чтобы это понять, попробуем поменять декораторы местами:

| ... @logger @simple\_cache def render\_input(field):  return f'<input id="id\_{field}" type="text" name="{field}">' ... *# call wrapper(username) # call wrapper(password) # call wrapper(username) ...* |
| --- |

Теперь логгер показывает вызов внутренней функции декоратора кеширования, причём трижды — так и должно быть! Тут всплывает ещё один **важный нюанс**: декоратор всё-таки оставляет следы. В некоторых ситуациях это может иметь значение, например, если мы написали [docstring](https://www.python.org/dev/peps/pep-0257/) для функции. Есть решение — использовать специальный декоратор [wraps](https://docs.python.org/3.8/library/functools.html#functools.wraps) из модуля functools:

| from functools import wraps   def simple\_cache(func):  cache = {}   @wraps(func)  def wrapper(\*args):  nonlocal cache  key = str(\*args)  if key not in cache:  cache[key] = func(\*args)  return cache[key]   return wrapper   def logger(func):  def wrapper(\*args):  result = func(\*args)  print(f'\tcall {func.\_\_name\_\_}({", ".join(map(str, args))})')  return result   return wrapper   @logger @simple\_cache def render\_input(field):  return f'<input id="id\_{field}" type="text" name="{field}">'   username\_f = render\_input('username') password\_f = render\_input('password') username\_f\_2 = render\_input('username') print(username\_f) print(password\_f) print(username\_f\_2) *# call render\_input(username) # call render\_input(password) # call render\_input(username)* |
| --- |

Теперь всё прозрачно — декоратор simple\_cache() работает, но мы видим именно декорируемую функцию на выходе, а не внутреннюю обёртку. Это очень важный нюанс!

**Примечание:** декорируемую функцию func передаём как аргумент декоратору wraps.

### \*Декоратор с аргументами

Предположим, что нам понадобилось контролировать детальность логирования: без аргументов, с аргументами, с результатом. Можно сделать три декоратора, но у вас при такой мысли сразу должен срабатывать сигнал «стоп» — это будет некрасиво и много копипаста. Более правильный вариант — передать декоратору некоторый аргумент, например, verbosity. Это приведет к появлению ещё одного уровня вложенности. Такие декораторы встречаются в фреймворке Django и не только.

В нашем случае решение может быть таким:

| def logger(verbosity=0):  def \_logger(func):  def wrapper(\*args):  result = func(\*args)  msg = f'\tcall {func.\_\_name\_\_}'  if verbosity > 0:  msg = f'{msg}({", ".join(map(str, args))})'  if verbosity > 1:  msg = f'{msg} -> {result}'  return msg   return wrapper   return \_logger   @logger() def render\_input(field):  return f'<input id="id\_{field}" type="text" name="{field}">'   username\_f = render\_input('username') password\_f = render\_input('password') print(username\_f) print(password\_f) *# call render\_input # call render\_input* |
| --- |

Применили декоратор без аргументов — получили минимальный уровень детализации.

**Важно:** теперь декоратор пишем как вызов функции (с круглыми скобками)!

Попробуем чуть детальнее — @logger(verbosity=1):

| call render\_input(username)  call render\_input(password) |
| --- |

Максимальная детализация — @logger(verbosity=2):

| call render\_input(username) -> <input id="id\_username" type="text" name="username">  call render\_input(password) -> <input id="id\_password" type="text" name="password"> |
| --- |

Можно таким же образом в декоратор передать функцию. Именно так и происходит в декораторе [user\_passes\_test](https://docs.djangoproject.com/en/3.1/topics/auth/default/#django.contrib.auth.decorators.user_passes_test) в фреймворке Django — его аргументом является функция, проверяющая пользователя по некоторым признакам: например, является ли он суперпользователем. В этом случае декоратор — своего рода посредник, который пробрасывает дополнительную проверку внутрь некоторого недоступного для редактирования кода.

**\*Примечание:** если вернуться к популярному декоратору кеширования [lru\_cache](https://docs.python.org/3.8/library/functools.html#functools.lru_cache), то он доступен в обоих вариантах: и без аргументов, и с аргументами. Рекомендуем изучить аналогичный [пример](https://realpython.com/primer-on-python-decorators/#both-please-but-never-mind-the-bread).

Разумеется, этими примерами тема декораторов не исчерпывается — есть отдельные случаи классов-декораторов и декораторов для методов классов. Но это все уже вопрос будущих курсов и спе

# Подведение итогов

Подошёл к концу курс по основам языка Python. Мы не пытались охватить все особенности этого замечательного языка или изучить все функции его стандартной библиотеки — основная задача была начать формировать определённое мышление, определённый подход.

Материал давался на разных уровнях, поэтому рекомендуем вернуться к нему через некоторое время: скорее всего, откроете много новых нюансов, недосягаемых при первом прохождении курса. Это нормально.

Не забывайте важное правило: продолжать кодить. Только так вы будете расти. Именно в процессе решения реальных задач приходит опыт. Особенно, когда что-то необъяснимо ломается или код ведёт себя совсем не так, как вы думали, — отладка требует большого объёма времени, и именно во время неё вы растёте как разработчик.

# Практическое задание

1. Написать функцию email\_parse(<email\_address>), которая при помощи регулярного выражения извлекает имя пользователя и почтовый домен из email адреса и возвращает их в виде словаря. Если адрес не валиден, выбросить исключение ValueError. Пример:

| >>> email\_parse('someone@geekbrains.ru') {'username': 'someone', 'domain': 'geekbrains.ru'} >>> email\_parse('someone@geekbrainsru') Traceback (most recent call last):  File "<stdin>", line 1, in <module>  ...  raise ValueError(msg) ValueError: wrong email: someone@geekbrainsru |
| --- |

**Примечание:** подумайте о возможных ошибках в адресе и постарайтесь учесть их в регулярном выражении; имеет ли смысл в данном случае использовать функцию re.compile()?

1. \*(вместо 1) Написать регулярное выражение для парсинга файла логов web-сервера из ДЗ 6 урока nginx\_logs.txt  
   (<https://github.com/elastic/examples/raw/master/Common%20Data%20Formats/nginx_logs/nginx_logs>) для получения информации вида: (<remote\_addr>, <request\_datetime>, <request\_type>, <requested\_resource>, <response\_code>, <response\_size>), например:

| raw = '188.138.60.101 - - [17/May/2015:08:05:49 +0000] "GET /downloads/product\_2 HTTP/1.1" 304 0 "-" "Debian APT-HTTP/1.3 (0.9.7.9)"' parsed\_raw = ('188.138.60.101', '17/May/2015:08:05:49 +0000', 'GET', '/downloads/product\_2', '304', '0') |
| --- |

**Примечание:** вы ограничились одной строкой или проверили на всех записях лога в файле? Были ли особенные строки? Можно ли для них уточнить регулярное выражение?

1. Написать декоратор для логирования типов позиционных аргументов функции, например:

| def type\_logger...  ...   @type\_logger def calc\_cube(x):  return x \*\* 3   >>> a = calc\_cube(5) 5: <class 'int'> |
| --- |

**Примечание:** если аргументов несколько - выводить данные о каждом через запятую; можете ли вы вывести тип значения функции? Сможете ли решить задачу для именованных аргументов? Сможете ли вы замаскировать работу декоратора? Сможете ли вывести имя функции, например, в виде:

| >>> a = calc\_cube(5) calc\_cube(5: <class 'int'>) |
| --- |

1. Написать декоратор с аргументом-функцией (callback), позволяющий валидировать входные значения функции и выбрасывать исключение ValueError, если что-то не так, например:

| def val\_checker...  ...   @val\_checker(lambda x: x > 0) def calc\_cube(x):  return x \*\* 3   >>> a = calc\_cube(5) 125 >>> a = calc\_cube(-5) Traceback (most recent call last):  ...  raise ValueError(msg) ValueError: wrong val -5 |
| --- |

**Примечание:** сможете ли вы замаскировать работу декоратора?

Задачи со \* предназначены для продвинутых учеников, которым мало сделать обычное задание.

# Дополнительные материалы

1. [Лутц Марк. Изучаем Python](https://www.ozon.ru/context/detail/id/5730448/).
2. <https://tproger.ru/translations/regular-expression-python/>.
3. [Замыкания](https://realpython.com/inner-functions-what-are-they-good-for/#whats-a-closure).

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://docs.python.org/3/>.