Переопределение функций и декораторы (дополнительный материал)

План урока

1. Переопределение функций
2. Инструкция pass. Согласованность аргументов
3. Инструкция def
4. Предосторожности при переопределении функций
5. Функция внутри функции
6. Декораторы

Аннотация

Этот урок содержит дополнительный материал для самостоятельного изучения.

Переопределение функций

Как мы видели в самом начале занятия, функцию можно записать в переменную. Но оказывается, что помимо этого с именем функции можно работать как с самой обыкновенную переменной. В том числе значение, на которое **указывает** переменная с именем функции, можно изменить.

Рассмотрим привер. У нас есть функция `input()`, которая читает данные из стандартного потока ввода. Давайте сделаем так, чтобы она не беспокоила пользователя и всегда давала один и тот же ввод. Для начала создадим свою функцию, которая будет работать вместо `input()`, а затем подменим значение `input` этой функцией.

```
def mainAnswerInTheUniverse():
    return 42
input = mainAnswerInTheUniverse
x = input()
print(x) # 42
```

Инструкция pass. Согласованность аргументов

Давайте теперь, чтобы совсем не отвлекать пользователя работой программы, подменим еще и функцию `print()` так, чтобы она ничего не печатала на экран. На одном из прошлых занятий мы писали функцию `nop()`, которая не делала ничего, но принимала в себя любой набор аргументов. Ей и воспользуемся.

```
def nop(*rest, **kwargs):
    pass
print = nop
print("(шепотом) Потише, пожалуйста!", end='')
```

Теперь функция `print()` связана с объектом функции, которая при вызове не делает ничего. Привычная команда `print()` изменила поведение.

Снова обращаем ваше внимание на то, что в аргументах функции `nop()` указано произвольное число аргументов (и произвольное число именованных аргументов). Благодаря этому, мы можем передавать ей разное число аргументов, как и в старую функцию `print()` и разные наборы опций, описываемые именованными аргументами (sep, end и т.п.). На самом деле, теперь не вызывают ошибки даже те наборы аргументов, которые не работают со встроенной функцией `print()`: функция `print()` принимает не любые именованные параметры, а только небольшой список, а функция `nop()` (а значит, и переопределенный `print()`) — абсолютно любые.

Подменяя функцию приходится заботиться о том, чтобы её аргументы соответствовали аргументам исходной функции. Иначе придется переписывать не только функцию, но и её вызовы, а в таком случае гораздо лучше просто завести новую функцию.

Инструкция def

Есть и более простой (но немного менее гибкий) способ заменить существующую функцию на другую. Для этого достаточно определить функцию с тем же именем — и она заменит существующую:

```
def input():
    return 42
x = input()
print(x) # 42
```

Интересно, что инструкцию `def` можно использовать в разных сложных конструкциях, словно это абсолютно обычная команда вроде присваивания или вызова функции.

Например, можно определять функцию по-разному в зависимости от некоторого условия:

```
language = 'fr'
if language == 'ru':
    def hello(name):
        print('Привет, ', name)
else:
    def hello(name):
        print('Hi, ', name)

hello('Joe') # => Hi, Joe
```

Дальше мы будем использовать более многословный способ.

Предосторожности при переопределении функций

Теперь, когда вы научились использовать такой сильный инструмент, как переопределение функций, давайте обсудим предосторожности, которые надо соблюдать при подобном занятии.

Стоит понимать, что изменять существующие функции **очень опасно**.

Почти всегда есть способ обойтись без этого, и этот способ стоит предпочесть. Переопределение функций делает программу плохо предсказуемой. Когда вы пишете функцию, часто бывает, что ее поведение зависит от поведения других функций. И если в дальнейшем вы поменяете поведение какой-то из этих функций, то сломаются многие из функций, которые опирались на измененную.

Еще хуже — почти недопустимо — изменять встроенные функции, как мы недавно делали.

Во-первых, в программе они используются настолько часто, что очень тяжело отследить все места, на которые они влияют. Например, представьте, что вы изменили функцию вычисления квадратного корня так, чтобы она не только считала его значение, но еще и печатала результат. Теперь, если вы попробуете вызвать функцию, решающую квадратные уравнения, вам не избежать вывода на экран корня из дискриминанта, даже если вы этого не хотели.

Во-вторых, встроенные функции и функции стандартной библиотеки вполне могут зависеть друг от друга, а вы об этом даже не знаете. Вам будем казаться, что изменилось поведение одной функции, когда на самом деле это далеко не так.

Теперь, после предупреждения о том, что подменять функции опасно, мы поговорим о том, когда и как это можно делать с пользой (и как при этом ничего не испортить).

Функция внутри функции

Аналогия функций с переменными имеет продолжение. Бывают функции, которые определены глобально, а бывают функции, определенные локально. Если внутри функции переопределить `print()`, то это будет локальная функция `print()`, а глобальная не изменится. Когда мы выйдем из области видимости, в которой был объявлен локальный `print()`, слово **print** вновь будет ссылаться на функцию в глобальной области видимости и будет работать как прежде.

Модификатор видимости `global`, кстати, с именами функций тоже работает (и им точно так же не рекомендуется пользоваться).

Подменять существующую функцию локально, в пределах другой функции обычно безопасно.

Давайте сделаем функцию, которая разыгрывает короткий диалог с пользователем, но чтобы программа казалась умнее, пока настоящий искусственный интеллект только разрабатывается, мы подменим функцию `answer()`, которая дает ответы на вопросы.

```
def answer(question):
    return 'В разработке'


def dialog():
    def answer(question):
        if question.lower().startswith('когда') :
            return 'Никогда!'
        else:
            return 'Разоблачили.'
    question = input()
    while question != '':
        print(answer(question))
        question = input()


dialog()

# <= Когда станет тепло?
# => Никогда!
# <= Когда смогу найти богатство?
# => Никогда!
# <= Какие в Чили существуют города?
# => Разоблачили
```

Этот пример служит исключительно иллюстрацией, ни в коем случае не рекомендацией. Ведь мы могли не переименовывать функцию, а просто определить новую функцию.

Задача: Подмените функцию `print()` так, чтобы она ПЕЧАТАЛА ВЕСЬ ТЕКСТ В ВЕРХНЕМ РЕГИСТРЕ. Реализовывать работу с именованными аргументами (sep, end, ...) не нужно.

Декораторы

Вы научились переопределять функцию так, чтобы новая функция использовала старую. Такой процесс называется **декорированием** функции. Но мы это сделали не слишком удачно: у нас появилась лишняя переменная и функция `oldPrint()`. Чтобы не вводить лишнюю переменную в глобальной области видимости, мы определим функцию в локальной области видимости, т.е. внутри другой функции. Внутреннюю функцию мы вернем как результат выполнения внешней функции:

```
def useUppercasedArguments(oldFunc):
    def newFunc(*args, **kwargs):
        argsUpcased = [str(arg).upper() for arg in args]
        oldFunc(*argsUpcased, **kwargs)
    return newFunc
print = useUppercasedArguments(print)
```

В коде выше произошло следующее: мы определили функцию `useUppercasedArguments()`, которая принимает одну функцию как аргумент и возвращает новую функцию, сделанную на основе старой. Функция, которая занимается этими превращениями, называется **декоратором**. Имя `newFunc`, которое мы определили внутри декоратора — локальное. Как только мы выполним `return newFunc`, это будет просто функция без имени.

Этой новой функции мы даем имя, которое было у старой функции. Декораторами такие функции как `useUppercasedArguments()` называются потому, что обычно они добавляют какие-то штрихи (декор) к уже существующему поведению.

Бывает так, что функцию (или, скорее, несколько функций) сразу пишут в расчёте на то, что она будет декорирована определенным образом. Обычно так поступают, когда функции хотят добавить некоторое типичное поведение. Например:

- Когда каждый вызов функции нужно залоггировать, т.е. вывести при вызове сообщение о том, как и когда функция была вызвана.
- Когда результат функции должен быть закеширован (т.е. после вычисления сохранен на будущее, чтобы не считать его повторно).
- Чтобы функция использовалась для ответа на запросы к веб-серверу.
- Чтобы перед запуском проверялось какое-то условие (например, что пользователь имеет право доступа к выполнению функции).

Мы напишем функцию, которая будет выводить журнал запусков и результатов декорированой функции. При каждом вызове функции мы перед вызовом будем писать номер вызова, аргументы и значение, которое функция вернула. Мы хотим вести такой отчет для функции приготовления бургеров из прошлого занятия, а также для функции наливания возвращающей тип напитка.

```
def logged(func):
    count = 0
    def decoratedFunc(*args, **kwargs):
        nonlocal count
        count +=1
        print(count, '>>', 'Arguments:', args, 'Named arguments:', kwargs)
        result = func(*args, **kwargs)
        print('--', 'Result:', result)
        return result
    return decoratedFunc


@logged
def make_burger(typeOfMeat, withOnion=False, withTomato=True):
    print('Булочка')
    if withOnion:
        print('Луковые колечки')
    if withTomato:
        print('Ломтик помидора')
    print('Котлета из', typeOfMeat)
    print('Булочка')


@logged
def drinking_type(type):
    return 'У нас есть только чай'
```

Сначала посмотрите на определения функций `make_burger()` и `drinking_type()`. Тело функций представляет мало интереса, это — очень простые функции.

А вот `@logged` перед определением функции — это новый для вас объект. Эта строчка говорит, что функция, которая идет дальше, будет задекорирована с помощью декоратора **logged**. Обратите внимание, что задекорированы обе функции одним декоратором. И никаких промежуточных функций и вспомогательных переменных мы не использовали.

Нелокальные переменные

Теперь обратимся к декоратору. В нем есть моменты, которых вы раньше не встречали. По условию, мы должны посчитать каждый вызов функции. На одном из прошлых уроков мы делали функцию `askAgain()`, которая тоже считала вызовы. В тот раз мы использовали глобальную переменную-счетчик, чтобы хранить, сколько раз мы уже вызывали функцию. В этот раз мы используем похожую технику и определяем переменную `count`, но не глобальную, а локальную для функции-декоратора. Затем мы создаем новую локальную функцию `decoratedFunc()`, которую позднее вернем в качестве результата. Эта функция должна иметь доступ к счетчику, который снаружи, но все-таки не в глобальной области видимости. Эта промежуточная область видимости называется **нелокальной**. Внутренняя функция видит переменные в объемлющей функции, но если она хочет такую переменную измененить, то должна объявить её нелокальной.

При поиске переменной или функции с указанными именем приоритет (или как говорят, правило разрешения имен) следующий:

1. сначала ищем локальную переменную (функцию),
2. если не нашли локальную - ищем нелокальную,
3. затем глобальную
4. и в самом конце — встроенную в язык.

Если вложенность функций больше двух уровней, то нелокальная переменная ищет в "ближайшей" области видимости, т.е. в функции вложенностью на один меньше. Если не находит, то поиск переходит в самую ближнюю из внешних областей видимости, затем в чуть более далекую — и так пока не найдется нужное имя.

Фактически, интерпретатор ищет "где поближе".

Такая техника позволяет сделать внешнюю для функции переменную, но при этом спрятанную от посторонних глаз (в отличие от глобальной). Такие переменные нужны в первую очередь для того, чтобы хранить какие-то данные, относящиеся к функции, между вызовами функции. Локальные переменные стираются при выходе из функции, глобальные - сохраняются, но видны всему свету, а нелокальные — это идеальное сочетание закрытости и "сохраняемости".

В некоторых языках программирования принято называть такие переменные статическими.

Важный момент: если вы попробуете запустить две получившиеся функции, то увидите, что их счётчики независимы. Это совершенно логично, поскольку эти переменные локальны для запусков самого декоратора `logged`. Когда мы оборачивали одну функцию, был создан один счётчик; когда оборачивали вторую - другой.

Небольшие, но типичные детали реализации декоратора.

Также стоит обратить внимание на список аргументов функции. Наша декорированая функция в отличие от исходной принимает любой набор аргументов, а когда вызывает исходную функцию, передает их в неизменном виде с помощью звездочек, раскрывающих список позиционных параметров и словарь именованных параметров. Это стандартный паттерн (то есть шаблон) для вызова функции, которую мы решили обернуть. Но не всегда список переданных аргументов бывает корректен, в этом случае обернутая функция просто завершится с ошибкой. Наконец, обратите внимание на ещё одну стандартную конструкцию: мы сначала сохранили значение, которое вернула обернутая функция, а затем его же и вернули.

Теперь мы пронумеруем и напечатанную строку с аргументами, и строку с результатом функции одним и тем же номером. Благодаря этому мы можем пронаблюдать дерево вызовов рекурсивной функции, т.е. такой функции, которая вызывает сама себя. Вы, возможно помните, что мы делали однажды функцию такую рекурсивную функцию для вычисления n-го числа Фиббоначчи.

Теперь обернем с помощью нашего декоратора её и вызовем:

```
def logged(func):
    count = 0
    def decoratedFunc(*args, **kwargs):
        nonlocal count
        count +=1
        current_index = count
        print(current_index, '>>', 'Arguments:', args, 'Named arguments:', kwarg
s)
        result = func(*args, **kwargs)
        print(current_index, '--', 'Result:', result)
        return result
    return decoratedFunc


@logged
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

fib(5)
```

Почему мы сделали current_index = count, а не использовали сам count, ведь count нигде в этой функции не изменяется? Но это иллюзия. `count` — нелокальная переменная, поэтому её может изменить вызов другой функции, имеющей доступ к этой переменной. У нас как раз такая ситуация: функция вызывает саму себя и в процессе этого вызова переменная `count` изменяется. Чтобы такого не было, мы заранее сохраняем значение счетчика в отдельную переменную, которую не меняют вызовы других функций. А так как числа неизменяемы, то "изменение" (присваивание нового значения) значения переменной count никак не повлияет на `current_index`.

Задача: Напишите декоратор **checkPassword**, который запрашивает пароль, прежде чем вызвать функцию и, если он неверный, - возвращает None и печатает "В доступе отказано". Запарольте этим декоратором функцию вычисления числа Фиббоначчи.

Захват значения из аргумента

Иногда бывает удобно сделать функцию, которая создает другие функции на основе аргументов. Например, мы хотим создать функции квадрата и куба числа.

```
def power(degree):
    def func(x):
        return x ** degree
    return func
square = power(2)
cube = power(3)

print(square(5)) # --> 25
```

Внутренняя функция взяла параметр degree из аргумента функции power(). Этот прием часто используется для того, чтобы зафиксировать какой-то набор параметров и сделать функцию с более коротким списком аргументов. Рассмотрим еще один пример. Пусть у нас есть функция send_invitation(email, name, text, date, city) и мы хотим использовать её, не перечисляя всякий раз весь этот внушительный список параметров. Например, вы знаете, что меняется только текст и адресат (имя и адрес), а город и дата мероприятия зафиксированы. Часть параметров будет браться из аргументов новой функции, часть — из аргументов старой.

```
def invitation_sender(city, date, text):
    def sender(email, name):
        return send_invitation(email, name, text, date, city)
    return sender
sendmail = invitation_sender('Москва', '1 апреля 2017 г', 'Приглашаем вас на вст
речу')
sendmail('vasiliy-petrov@yandex.ru', 'Василий Петров')
sendmail('petr-alekseev@yandex.ru', 'Петр Алексеев')
sendmail('vasiliy-vasilyev@yandex.ru', 'Василий Васильев')
```

Функция, которая захватила объект из внешнего контекста, этот объект удерживает вечно. Хотя degree в функции power является локальной переменной, эта переменная не исчезнет бесследно после завершения работы функции power. Функция, которая будет возвращена несет значение этой переменной degree с собой.

Немного о терминологии

Программисты, как и любые другие специалисты, любят давать понятиям названия. Давайте дадим названия тем понятиям, которые вы уже узнали.

Функция, которая использует внешние переменные, не являющиеся её аргументами, называется **замыканием**. Если использует нелокальные переменные — она является замыканием (независимо от того, были ли они определены во внешней функции или пришли из аргументов внешней функции). Если функция использует глобальные переменные — это тоже замыкание. Но чаще всего замыканием называют все-таки функцию, которая использует нелокальные переменные. Такая функция как бы "таскает за собой" свои внешние переменные, но никому их не показывает.

Функция, которая принимает функцию в качестве аргумента или же возвращает функцию, называется **функцией высшего порядка**.

Функция, которая принимает функцию и возвращает функцию называется **декоратором**. Но при этом предполагается, что возвращенная функция должна быть оберткой над переданной функцией.

Задача *: Напишите генератор декораторов **checkPassword**, т.е. функцию, которая возвращает декоратор. Генератор декораторов принимает в качестве параметра пароль, и получившийся декоратор должен запароливать функцию этим паролем. Этот декоратор будет применяться следующим образом:

```
@checkPassword('password')
make_burger(typeOfMeat, withOnion = False, withTomato = True):
    # ...
```

Т.е при определении функции сначала вызывается функция `checkPassword()` с аргументом 'password', получается декоратор, затем уже этот получившийся декоратор применяется к функции.

Задача *: Напишите декоратор `cached`, который будет кэшировать результат вызова функции.

Когда задекорированная функция впервые вызывается с некоторым аргументом, необходимо выполнить исходную функцию и сохранить посчитанный результат. При последующих вызовах не надо вызывать исходную функцию, а следует найти сохраненное значение и вернуть его. Этот декоратор будет очень полезен для чистых вычислений (т.е. вычислений, зависящих только от переданных аргументов), которые занимают продолжительное время.

Вы уже видели подобное использование функций в уроке про области видимости переменных. В задаче оттуда "Долго только поначалу" рекурсивное вычисление числа Фиббоначчи при помощи функции `fib()` занимает очень много времени, но функция `longOnlyOnce()` позволяет "закэшировать" результат вычисления этой функции.

В этот раз задача сложнее: во-первых, задекорированная функция должна самостоятельно осуществлять кэширование. во-вторых, кэширование должно зависеть от переданных параметров, т.е. вам следует сохранять не один результат, а список посчитанных уже пар значений вида (набор параметров, результат).

Мы предлагаем искать результат, перебирая список посчитанных значений, пока не найдёте переданный набор параметров. В будущем, когда вы узнаете про такую структуру данных как **словарь**, вы сможете переписать декоратор значительно более эффективно, не используя долгий перебор.

Стоит отметить, что из-за перебора скорость задекорированной функции может значительно снизиться, но для очень долго выполняемых функций вы всё равно получите прирост в скорости.

Пример того, как можно будет использовать ваш декоратор:

```
@cached
def fib(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

print(fib(60))
```

Если рекурсивное вычисление числа Фиббоначчи не закэшировать, вы едва ли дождётесь того, что шестидесятое число последовательности посчитается. Версия же с кэшированием сделает это очень быстро.

Есть множество менее искусственных примеров, где такое кэширование может быть полезно: хотя в примере с числами Фиббоначчи есть эффективный алгоритм их вычисления, вы не всегда знаете быстрый способ. А даже если знаете, иногда его слишком тяжело реализовать, тогда как сделать кэширование при помощи вашего декоратора займёт у вас всего лишь одну строчку.

Кроме математических вычислений бывают и другие долгие операции. Например, так можно сохранять результаты интернет-запросов (интернет-запрос — это очень долгая операция по меркам компьютера).