функциональное программирование на

PYTHON

в примерах сообщества Stack Overflow

Издательство АСТ
г. Москва

УДК 004.43
ББК 32.973
М52

Мерц, Дэвид.

М52 Функциональное программирование на Python / Д. Мерц, Л. Клаймен. — Москва : Издательство АСТ, 2026. — 128 с. : ил. — (Программирование. Оффер на миллион).

ISBN 978-5-17-178630-4.

Перед вами практическое руководство, посвященное функциональному программированию на Python — подходу, который помогает писать более предсказуемый, модульный и устойчивый к ошибкам код. Книга шаг за шагом знакомит читателя с основными концепциями функционального стиля, от базовых принципов и встроенных средств языка до продвинутых приемов, применяемых в современных проектах.

Вы узнаете, как использовать функции высшего порядка, замыкания, декораторы и ленивые вычисления для построения гибких программных решений. Подробно рассматриваются механизмы итераторов и генераторов, принципы композиции функций, организация пайплайнов обработки данных и применение асинхронного функционального программирования. Особое внимание уделено темам мемоизации, конкурентности и интеграции с популярными инструментами, такими как itertools, functools, operator, asyncio и Pydantic.

В книге подробно рассказывается, как сочетать лаконичность и выразительность функционального стиля с возможностями Python, избегая излишней императивности и улучшая читаемость кода. Приведенные примеры и практические рекомендации помогут не только глубже понять теорию функционального программирования, но и сразу применить ее на практике — при проектировании, оптимизации и тестировании приложений.

Издание предназначено для разработчиков, стремящихся выйти за рамки традиционного объектно-ориентированного подхода и освоить функциональную парадигму, чтобы писать более элегантный, надежный и современный Python-код.

УДК 004.43
ББК 32.973

ISBN 978-5-17-178630-4

Содержание

Предисловие

Что такое функциональное программирование (functional programming, FP)?

Начнем, пожалуй, с самого трудного вопроса: что же такое функциональное программирование (functional programming, FP)?

Один из ответов — это то, что вы делаете, когда программируете на таких языках, как Lisp, Scheme, Clojure, Scala, Haskell, ML, OCaml, Erlang и некоторых других. Это верный ответ, но он мало что проясняет. К сожалению, трудно получить единое мнение о том, что именно представляет собой функциональное программирование, даже от самих функциональных программистов. Также можно противопоставить функциональное программирование императивному программированию (imperative programming) — тому, чем вы занимаетесь на языках вроде C, Pascal, C++, Java, Perl, Awk, TCL и большинстве других. Функциональное программирование — это также не объектно-ориентированное программирование (object-oriented programming, OOP), хотя некоторые языки поддерживают оба подхода. И это не логическое программирование (logic programming), например Prolog, но и здесь существуют мультипарадигмальные (multiparadigm) языки.

Лично я в общих чертах охарактеризовал бы функциональное программирование как обладающее по меньшей мере несколькими из следующих признаков. Языки, которые называют функциональными, упрощают реализацию этих идей, делая другие вещи сложными или вовсе недоступными:

- Функции (functions) — объекты первого класса (first-class objects). То есть все, что вы можете делать с данными (data), можно делать и с самими функциями (например, передавать функцию в другую функцию).
- Рекурсия (recursion) используется в качестве основного управляющего механизма. В некоторых языках не существует никаких других конструкций «цикла».
- Существует фокус на обработку списков (list processing) — например, отсюда происходит название Lisp. Списки (lists) часто используются вместе с рекурсией по подспискам как замена циклам.
- «Чистые» (pure) функциональные языки избегают побочных эффектов (side effects). Это исключает почти повсеместный в императивных языках шаблон присваивания сначала одного, затем другого значения одной и той же переменной (variable) для отслеживания состояния программы.
- Функциональное программирование либо не поощряет, либо полностью запрещает инструкции (statements), вместо этого опираясь на вычисление выражений (expressions) — другими словами, на функции плюс аргументы. В чистом случае одна программа — это одно выражение (плюс поддерживающие определения).

- Функциональное программирование (functional programming) заботится о том, что именно следует вычислить, а не о том, как это следует вычислить.
- Большая часть функционального программирования использует функции высшего порядка (higher-order functions) — то есть функции, которые оперируют функциями, оперирующими функциями.

Сторонники функционального программирования утверждают, что все перечисленные свойства обеспечивают более быструю разработку, меньший объем и меньшую склонность к ошибкам кода. Кроме того, теоретики высокого уровня в области информатики, логики и математики считают гораздо более простым доказать формальные свойства функциональных языков и программ, чем императивных языков и программ. Одним из ключевых понятий в функциональном программировании является «чистая функция» — функция, которая всегда возвращает один и тот же результат при одних и тех же аргументах, — что гораздо ближе к значению функции в математике, чем в императивном программировании.

Python определенно не является чистым функциональным языком программирования (pure functional programming language) — побочные эффекты широко распространены в большинстве программ на Python. То есть переменные часто переназначаются, изменяемые коллекции данных (mutable data collections) нередко меняют свое содержимое, а ввод-вывод (I/O) свободно перемежается с вычислениями. Он также не является языком функционального программирования (functional programming language) в более общем смысле. Однако Python — мультипарадигмальный язык (multiparadigm language), который делает функциональное программирование простым, когда это требуется, и позволяет легко смешивать его с другими стилями программирования.

За пределами стандартной библиотеки (standard library)

Хотя в рамках данного краткого отчета они не будут обсуждаться, существует большое число полезных сторонних библиотек Python для функционального программирования. Единственным исключением здесь будет обсуждение библиотеки Matthew Rocklin под названием `multipledispatch` как лучшей на текущий момент реализации своей концепции.

Большинство сторонних библиотек вокруг функционального программирования представляют собой коллекции функций высшего порядка (higher-order functions) и иногда расширения инструментов для «ленивой» работы с итераторами (iterators), содержащихся в `itertools`. Некоторые заметные примеры включают следующее, но этот список не следует считать исчерпывающим:

- `pyrsistent` содержит ряд неизменяемых коллекций (immutable collections). Все методы структуры данных, которые обычно изменяют ее (mutate), вместо этого возвращают новую копию структуры с требуемыми обновлениями. Исходная структура остается нетронутой.
- `toolz` предоставляет набор утилитарных функций (utility functions) для итераторов (iterators), функций (functions) и словарей (dictionaries). Эти функции тесно взаимосвязаны и образуют строительные блоки типичных операций анализа данных. Они расширяют стандартные библиотеки

`itertools` и `functools` и во многом опираются на стандартные библиотеки современных функциональных языков.

- `hypothesis` — это библиотека для создания модульных тестов (unit tests) с целью поиска ошибок в вашем коде, о которых вы бы никогда не подумали. Она генерирует случайные данные, соответствующие вашей спецификации, и проверяет, что ваша гарантия все еще выполняется в этих случаях. Это часто называют тестированием на основе свойств (property-based testing), популяризированным библиотекой Haskell QuickCheck.
- `more_itertools` пытается собрать полезные композиции итераторов, которые не покрываются ни `itertools`, ни рецептами, включенными в его документацию. Эти композиции сложно реализовать корректно, и данная хорошо продуманная библиотека помогает пользователям избежать подводных камней при самостоятельной реализации.

Стилевое примечание

Как и в большинстве книг по программированию, для встроенных и блочных примеров кода будет использоваться `моноширинный шрифт`, включая простые имена команд или функций. Внутри блоков кода условный фрагмент псевдокода (pseudo code) обозначается словом в угловых скобках, то есть это невалидный для Python фрагмент, например `<code-block>` . В иных случаях синтаксически валидные, но не определенные функции используются с описательными именами, такими как `get_the_data()`.

(Избегая) управление потоком выполнения (Flow Control)

В типичных императивных программах на Python, включая те, что используют классы (classes) и методы (methods) для размещения императивного кода, блок кода обычно состоит из внешних циклов (loops) `for` или `while`, присваивания переменных состояния (state variables) внутри этих циклов, модификации структур данных (data structures), таких как `dict`, `list` и `set` (или различных иных структур из стандартной библиотеки или сторонних пакетов), а также операторов ветвления `if/elif/else` или `try/except/finally`. Все это понятно и поначалу кажется простым для применения. Однако проблемы часто возникают именно из-за тех побочных эффектов, которые связаны с переменными состояния и изменяемыми структурами данных (mutable data structures) — они нередко хорошо моделируют наши представления из физического мира контейнеров, но при этом сложно рассуждать о том, в каком состоянии находятся данные в конкретный момент выполнения программы.

Одно из решений — сосредоточиться не на конструировании коллекции данных, а на описании того, что именно составляет эту коллекцию данных. Когда вы просто думаете: «Вот есть некоторые данные, что мне нужно с ними сделать?» — а не о механизме построения данных, рассуждать часто оказывается проще. Императивное управление потоком, описанное в предыдущем абзаце, гораздо больше про «как», чем про «что», и часто мы можем сместить фокус своего внимания.

Инкапсуляция (Encapsulation)

Очевидный способ сфокусироваться на «что», а не на «как», — просто отрефакторить код и поместить конструирование данных в более изолированное место, то есть в функцию или метод. Например, рассмотрим существующий фрагмент императивного кода, который выглядит так:

```
# настроить исходные данные
collection = get_initial_state()
state_var = None
for datum in data_set:
    if condition(state_var):
        state_var = calculate_from(datum)
        new = modify(datum, state_var)
        collection.add_to(new)
    else:
        new = modify_differently(datum)
        collection.add_to(new)
# теперь действительно работаем с данными
for thing in collection:
    process(thing)
```

Мы можем просто убрать «как» построения данных из текущей области видимости и спрятать его в функцию, о которой можно размышлять изолированно (или вовсе не думать, когда абстракция достаточно хороша). Например:

```
# спрятать построение данных
def make_collection(data_set):
    collection = get_initial_state()
    state_var = None
    for datum in data_set:
        if condition(state_var):
            state_var = calculate_from(datum, state_var)
            new = modify(datum, state_var)
            collection.add_to(new)
        else:
            new = modify_differently(datum)
            collection.add_to(new)
    return collection
# теперь действительно работаем с данными
for thing in make_collection(data_set):
    process(thing)
```

Мы не изменили логику программирования и даже не затронули строки кода, но все же сместили фокус с «Как мы конструируем `collection`?» на «Что создает `make_collection()`?».

Вложения (comprehensions)

Использование вложений списков и прочих comprehension-форм одновременно делает код более компактным и смещает фокус с «как» на «что». Включаемое выражение (comprehension) — это выражение, которое использует те же ключевые слова, что и циклы и условные блоки (conditionals), но инвертирует их порядок, чтобы сосредоточиться на данных, а не на процедуре. Простое изменение формы выражения нередко существенно влияет на то, как мы рассуждаем о коде и насколько легко его понимать. Тернарный оператор (ternary operator) также выполняет похожую перестройку фокуса, используя те же ключевые слова в ином порядке. Например, если исходный код был таким:

```
collection = list()
for datum in data_set:
    if condition(datum):
        collection.append(datum)
    else:
        new = modify(datum)
        collection.append(new)
```

... то более компактно можно написать так:

```
collection = [d if condition(d) else modify(d)
    for d in data_set]
```

Гораздо важнее, чем просто сэкономить несколько символов и строк — это ментальный сдвиг, достигаемый размышлением о том, что такое `collection`,

и избеганием необходимости отвечать на вопрос «Каково состояние `collection` в этой точке цикла?».

Включения списков дольше всего присутствуют в Python и в некотором смысле являются самыми простыми. Сейчас в синтаксисе Python также доступны включения выражений, включения множеств (set comprehensions) и включения словарей (dict comprehensions). Однако оговоримся: хотя вы можете вкладывать выражения друг в друга на произвольную глубину, после довольно простого уровня они перестают прояснять и начинают «затуманивать» текущую ситуацию. Для действительно сложного построения коллекции данных рефакторинг в функции остается более читаемым вариантом.

Генераторы (Generators)

Генераторные выражения (generator comprehensions) имеют тот же синтаксис, что и генераторы списков (list comprehensions) — за исключением того, что вокруг них нет квадратных скобок (но круглые скобки синтаксически требуются в некоторых контекстах вместо скобок списка). При этом они «ленивые». То есть это лишь описание того, как получить данные, которое не материализуется, пока его явно не попросят — либо вызовом метода `.next()` у объекта, либо проходом по нему в цикле. Это часто экономит память для больших последовательностей и откладывает вычисления до момента, когда они действительно понадобятся. Например:

```
log_lines = (line for line in read_line(huge_log_file)
    if complex_condition(line))
```

В типичных случаях поведение будет таким же, как если бы вы построили список, но поведение во время выполнения будет более «приятным». Очевидно, у этого генераторного выражения есть и императивные версии (imperative versions), например:

```
def get_log_lines(log_file):
    line = read_line(log_file)
    while True:
        try:
            if complex_condition(line):
                yield line
            line = read_line(log_file)
        except StopIteration:
            raise
log_lines = get_log_lines(huge_log_file)
```

Да, императивную версию можно было бы тоже упростить, но показанный вариант предназначен для иллюстрации скрытого «как» цикла `for` по итерируемому объекту (iterable) — дополнительных подробностей, от которых мы также хотим абстрагироваться в своем мышлении. По сути, даже использование `yield` — это своего рода абстракция поверх базового протокола итератора (iterator protocol). Мы могли бы сделать то же самое с классом, у которого есть методы `.next()` и `.iter()`. Например:

```
class GetLogLines(object):
    def __init__(self, log_file):
```

```
        self.log_file = log_file
        self.line = None
    def __iter__(self):
        return self
    def __next__(self):
        if self.line is None:
            self.line = read_line(log_file)
        while not complex_condition(self.line):
            self.line = read_line(self.log_file)
        return self.line
log_lines = GetLogLines(huge_log_file)
```

Отвлекаясь от протокола итератора и «ленивости» в целом, читатель должен увидеть, что генераторные выражения лучше фокусируют внимание на «что», тогда как императивная версия — хотя, возможно, и удачная как рефакторинг — сохраняет фокус на «как».

Словари и множества (Dicts and Sets)

Аналогично тому, как списки (lists) могут создаваться с помощью вложенных списков, а не путем создания пустого списка, обхода в цикле и многократных вызовов `.append()`, словари (dictionaries) и множества (sets) можно создавать «сразу целиком», а не путем многократных вызовов `.update()` или `.add()` в цикле. Например:

```
>>> {i:chr(65+i) for i in range(6)}
{0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F'}
>>> {chr(65+i) for i in range(6)}
{'A', 'B', 'C', 'D', 'E', 'F'}
```

Императивные версии этих генераторов выглядят очень похоже на примеры, показанные ранее для других встроенных типов данных (built-in datatypes).

Рекурсия (Recursion)

Функциональные программисты часто делают упор на выражении управления потоком выполнения (flow control) через рекурсию (recursion+), а не через циклы (loops). В таком подходе можно избежать изменения состояния каких-либо переменных или структур данных в алгоритме и, что важнее, приблизиться к «что», а не к «как» вычисления. Однако, рассматривая возможность использования рекурсивных стилей, следует различать случаи, когда рекурсия — это просто «итерация другим способом», и случаи, когда задачу легко разделить на более мелкие подзадачи и подойти к каждой из них сходным образом.

Есть две причины делать упомянутое различие. С одной стороны, использование рекурсии по сути как способа прохода по последовательности элементов хотя и возможно, на самом деле не является «питоничным». Это вполне соответствует стилю других языков, таких как Lisp, но в Python часто выглядит неестественно. С другой стороны, Python сравнительно медленный на рекурсии и имеет ограниченную глубину стека. Да, вы можете изменить ее с помощью `sys.setrecursionlimit()` на значение больше стандартного 1000; но если вы ловите себя на том, что делаете это, вероятно, это ошибка. В Python отсутствует

внутренняя особенность под названием «оптимизация хвостовых вызовов (tail call elimination)», делающая глубокую рекурсию вычислительно эффективной в некоторых языках. Рассмотрим тривиальный пример, где рекурсия — по сути разновидность итерации:

```
def running_sum(numbers, start=0):
    if len(numbers) == 0:
        print()
        return
    total = numbers[0] + start
    print(total, end=" ")
    running_sum(numbers[1:], total)
```

Однако здесь мало что можно рекомендовать: итерация, которая просто многократно изменяет переменную состояния `total`, была бы более читаемой, и к тому же эту функцию вполне разумно вызывать для последовательностей намного большей длины, чем 1000.

В других случаях рекурсивный стиль, даже поверх последовательных операций, все же выражает алгоритмы более интуитивно и в форме, о которой легче рассуждать. Чуть менее тривиальный пример — факториал в рекурсивном и итеративном стилях:

```
def factorialR(N):
    "Recursive factorial function"
    assert isinstance(N, int) and N >= 1
    return 1 if N <= 1 else N * factorialR(N-1)
def factorialI(N):
    "Iterative factorial function"
    assert isinstance(N, int) and N >= 1
    product = 1
    while N >= 1:
        product *= N
        N -= 1
    return product
```

Хотя этот алгоритм можно легко выразить с помощью переменной «текущего произведения», рекурсивное выражение все же ближе к «что», чем к «как» алгоритма. Подробности многократного изменения значений `product` и `N` в итеративной версии ощущаются как «бухгалтерия», а не сущность самого вычисления (хотя итеративная версия, вероятно, быстрее, и лимит рекурсии легко достигается, если его не корректировать).

В качестве примечания: самая быстрая версия `factorial()`, которую я знаю в Python, выполнена в стиле функционального программирования и также хорошо выражает «что» алгоритма, если вам знакомы некоторые функции высшего порядка (higher-order functions):

```
from functools import reduce
from operator import mul
def factorialHOF(n):
    return reduce(mul, range(1, n+1), 1)
```

Там, где рекурсия действительно убедительна, а иногда и единственно очевидный способ выразить решение — это когда задача поддается подходу «разделяй и властвуй», то есть если можно выполнить похожее вычисление над двумя половинами (или, во всяком случае, несколькими сходного размера частями) более крупной коллекции. В таком случае глубина рекурсии составляет всего `O(log N)` от размера коллекции, что вряд ли будет чрезмерно глубоким. Например, алгоритм быстрой сортировки (quicksort) весьма изящно выражается без каких-либо переменных состояния или циклов, а целиком через рекурсию:

```
def quicksort(lst):
    "Quicksort over a list-like sequence"
    if len(lst) == 0:
        return lst
    pivot = lst[0]
    pivots = [x for x in lst if x == pivot]
    small = quicksort([x for x in lst if x < pivot])
    large = quicksort([x for x in lst if x > pivot])
    return small + pivots + large
```

В теле функции используются некоторые имена для хранения удобных значений, но они никогда не мутируют. Это было бы не столь читаемо, но при желании определение можно записать одним выражением. Фактически превратить это в стабильную итеративную версию довольно трудно и, несомненно, менее интуитивно.

В качестве общего совета: хорошей практикой является поиск возможностей рекурсивного выражения — и особенно версий, избегающих необходимости в переменных состояния или изменяемых коллекциях данных — всякий раз, когда задача выглядит делимой на подзадачи. В Python обычно не лучшая идея использовать рекурсию лишь как «итерацию другими средствами».

Устранение циклов (Eliminating Loops)

Просто ради удовольствия быстро посмотрим, как можно убрать все циклы из любой программы на Python. В большинстве случаев это плохая идея — как для читаемости, так и для производительности, — но стоит увидеть, насколько просто это сделать систематически, чтобы получить тему для размышления о случаях, когда это действительно хорошая идея.

Если мы просто вызываем функцию внутри цикла `for`, нам на помощь приходит встроенная функция высшего порядка `map()`:

```
for e in it:    # цикл на операторах (statement-based loop)
    func(e)
```

Следующий код полностью эквивалентен функциональной версии, за исключением того, что здесь нет многократного переназначения переменной `e`, а значит, нет состояния:

```
map(func, it)   # "loop" на основе map()
```

Похожий прием доступен и для функционального подхода к последовательному потоку выполнения. Большая часть императивного программирования состоит из операторов, которые сводятся к «сделай это, затем то, затем еще вот