Основы функционального программирования на Python

Принципы функционального программирования

*КЛЮЧЕВЫЕ ПОЛОЖЕНИЯ:*

Функциональное программирование представляет собой методику написания программного обеспечения, в центре внимания которой находятся функции. Функции могут присваиваться переменным, они могут передаваться в другие функции и порождать новые функции. **Python имеет богатый и мощный арсенал инструментов, которые облегчают разработку функционально-ориентированных программ.**

В последние годы почти все известные процедурные и объектно-ориентированные языки программирования стали поддерживать средства функционального программирования (ФП). И язык Python не исключение.

Когда говорят о ФП, прежде всего имеют в виду следующее:

* Функции – это «*граждане* *более высокого сорта*», т.е., все, что можно делать с «данными», можно делать и с функциями (в том числе передача функции другой функции в качестве аргумента).
* Использование *рекурсии* в качестве основной структуры контроля потока управления. В некоторых языках не существует иной конструкции цикла, кроме рекурсии.
* Акцент на *обработке последовательностей*. Списки с рекурсивным обходом подсписков часто используются в качестве замены циклов.
* «Чистые» функциональные языки избегают побочных эффектов. Это *исключает присваивания*, почти повсеместно распространенный в императивных языках подход, при котором за одной и той же переменной последовательно закрепляются разные значения для отслеживания состояния программы.
* ФП не одобряет или совершенно запрещает инструкции, используя вместо этого *вычисление выражений* (т.е. функций с аргументами). В предельном случае, одна программа есть одно выражение (плюс дополнительные определения).
* ФП акцентируется на том, *что* должно быть вычислено, а не *как*.

Функциональное программирование представляет собой методику написания программного обеспечения, в центре внимания которой находятся функции. В парадигме ФП объектами первого класса являются функции. Они обрабатываются таким же образом, что и любой другой примитивный тип данных, такой как строковый и числовой. Функции могут получать другие функции в виде аргументов и на выходе возвращать новые функции. Функции, имеющие такие признаки, называются *функциями более высокого порядка* из-за их высокой выразительной мощи. И вам непременно следует воспользоваться их чудесной выразительностью.

Программистам чаще приходится работать с последовательностями значений, такими как списки и кортежи, или же контейнерами, такими как словари и множества. Как правило, в файлах хранятся большие объемы текстовых или числовых данных, которые затем загружаются в программу в соответствующие структуры данных и обрабатываются. Python имеет богатый и мощный арсенал инструментов, которые облегчают их обработку в функциональном стиле.

Далее будут представлены несколько таких встроенных функций.

Оператор lambda, функции map, filter, reduce и другие

Прежде чем продолжить, сначала следует познакомиться с еще одним ключевым словом языка Python. Он позволяет определять еще один тип функций.

Оператор lambda

Помимо стандартного определения функции, которое состоит из заголовка функции с ключевым словом def и блока инструкций, в Python имеется возможность создавать короткие однострочные функции с использованием оператора lambda, которые называются *лямбда-функциями*. Вот общий формат определения лямбда-функции:

lambda список\_аргументов: выражение

В данном формате *список\_аргументов* – это список аргументов, отделенных запятой, и *выражение* – значение либо любая порция программного кода, которая в результате дает значение. Например, следующие два определения функций эквивалентны:

**def standard\_function(x, y):**

**return x + y**

**standard\_function(5, 7)**

и

lambda x, y: x + y

Но в отличие от стандартной функции, после определения лямбда-функции ее можно сразу же применить, к примеру, в интерактивном режиме:

(lambda x, y: x+y)(5, 7)

12

Либо, что более интересно, присвоить ее переменной, передать в другую функцию, вернуть из функции, разместить в качестве элемента последовательности или применить в программе, как обычную функцию. Приведенный ниже интерактивный сеанс это отчасти демонстрирует. (Для удобства добавлены номера строк.)

lambda\_function = **lambda** x, y: x + y  
lambda\_function(5,7)  
12

func = lambda\_function  
func(3,4)  
7

dic = {'функция1': lambda\_function}  
dic['функция1'](7,8)  
15

Здесь в строке 1 определяется лямбда-функция и присваивается переменной, которая теперь ссылается на лямбда-функцию. В строке 2 она применяется с двумя аргументами. В строке 4 ссылка на эту функцию присваивается еще одной переменной, и затем пользуясь этой переменной данная функция вызывается еще раз. В строке 7 создается словарь, в котором в качестве значения задана ссылка на эту функцию, и затем, обратившись к этому значению по ключу, эта функция применяется в третий раз.

Нередко во время написания программы появляется необходимость преобразовать некую последовательность в другую. Для этих целей в Python имеется встроенная функция map.

Функция map

При написании программы очень часто возникает задача, которая состоит в том, чтобы применить специальную функцию для всех элементов в последовательности. В функциональном программировании она называется отображением от англ. map.

Встроенная в Python функция map – это функция более высокого порядка, которая предназначена для выполнения именно такой задачи. Она позволяет обрабатывать одну или несколько последовательностей с использованием заданной функции. Вот общий формат функции map:

map(функция, последовательности)

В данном формате *функция* – это ссылка на стандартную функцию либо лямбда-функция, и *последовательности* – это одна или несколько отделенных запятыми итерируемых последовательностей, т.е. списки, кортежи, диапазоны или строковые данные.

seq = (1, 2, 3, 4, 5, 6, 7, 8, 9)  
seq2 = (5, 6, 7, 8, 9, 0, 3, 2, 1)  
result = map(lambda\_function, seq, seq2)  
result  
<map object at 0x000002897F7C5B38>

list(result)  
[6, 8, 10, 12, 14, 6, 10, 10, 10]

В приведенном выше интерактивном сеансе в строках 1 и 2 двум переменным, seq и seq2, присваиваются две итерируемые последовательности. В строке 3 переменной result присваивается результат применения функции map, в которую в качестве аргументов были переданы ранее определенная лямбда-функция и две последовательности. Обратите внимание, что функция map возвращает объект-последовательность map, о чем говорит строка 5. Особенность объекта-последовательности map состоит в том он может предоставлять свои элементы, только когда они требуются, используя *ленивые вычисления*.

Ленивые вычисления – это стратегия вычисления, согласно которой вычисления следует откладывать до тех пор, пока не понадобится их результат. Программистам часто приходится обрабатывать последовательности, состоящие из десятков тысяч и даже миллионов элементов. Хранить их в оперативной памяти, когда в определенный момент нужен всего один элемент, не имеет никакого смысла. Ленивые вычисления позволяют генерировать ленивые последовательности, которые при обращении к ним предоставляют следующий элемент последовательности. Чтобы показать ленивую последовательность, в данном случае результат работы примера, необходимо эту последовательность «вычислить». В строке 6 объект map вычисляется во время преобразования в список.

Функция filter

Функции более высокого порядка часто используются для фильтрации данных. Языки функционального программирования предлагают универсальную функцию filter, получающую набор элементов для фильтрации, и фильтрующую функцию, которая определяет, нужно ли исключить конкретный элемент из последовательности или нет. Встроенная в Python функция filter выполняет именно такую задачу. В результирующем списке будут только те значения, для которых значение функции для элемента последовательности истинно. Вот общий формат функции filter:

filter(предикативная\_функция, последовательность)

В данном формате *предикативная\_функция* – это ссылка на стандартную функцию либо лямбда-функция, которая возвращает истину либо ложь, и *последовательность* – это итерируемая последовательность, т.е. список, кортеж, диапазон или строковые данные.

Например, ниже приведена однострочная функция is\_even для определения четности числа:

is\_even = **lambda** x: x % 2 == 0

Чтобы отфильтровать все числа последовательности и оставить только четные, применим функцию filter:

seq = (1, 2, 3, 4, 5, 6, 7, 8, 9)  
filtered = filter(is\_even, seq)  
list(filtered)  
[2, 4, 6, 8]

Приведенный выше фрагмент кода можно переписать по-другому, поместив лямбда функцию в качестве первого аргумента:

filtered = iter(filter(**lambda** x: x % 2 == 0, seq))  
list(filtered)  
[2, 4, 6, 8]

И снова, в обоих случаях функция filter возвращает ленивый объект-последовательность, который нужно вычислить, чтобы увидеть результат. В иной ситуации в программе может иметься процесс, который потребляет по одному элементу за один раз. В этом случае в него можно подавать по одному элементу, вызывая встроенную функцию next.

next(filtered)  
2  
next(filtered)  
4  
...

**Примечание**. Для предотвращения выхода за пределы ленивой последовательности необходимо отслеживать возникновение ошибки StopIteration. Например,

seq = sequence  
**try**:  
 total = next(seq)  
**except** StopIteration:  
 **return**

Функция reduce

Наконец, когда требуется обработать список значений таким образом, чтобы свести процесс к единственному результату, для этого используется функция reduce. Функция reduce имеется в модуле functools стандартной библиотеки, но здесь она будет приведена целиком, чтобы показать, как она работает:

**def** reduce(fn, seq, initializer=**None**):  
 it = iter(seq)  
 value = next(it) **if** initializer **is** **None** **else** initializer  
 **for** element **in** it:  
 value = fn(value, element)  
 **return** value

Вот общий формат функции reduce:

reduce(функция, последовательность, инициализатор)

В данном формате *функция* – это ссылка на редуцирующую функцию; ею может быть стандартная функция либо лямбда-функция, *последовательность* – это итерируемая последовательность, т.е. список, кортеж, диапазон или строковые данные, и *инициализатор* – это параметрическая переменная, которая получает начальное значение для накопителя. Начальным значением может быть значение любого примитивного типа данных либо мутабельный объект – список, кортеж и т.д. Начальное значение инициирует накапливающую переменную, которая прежде чем она будет возвращена, будет обновляться редуцирующей функцией по каждому элементу в списке.

Переданная при вызове функция вызывается в цикле для каждого элемента последовательности. Например, функция reduce может применяться для суммирования числовых значений в списке. Например, вот так:

seq = (1, 2, 3, 4, 5, 6, 7, 8, 9)  
get\_sum = **lambda** a, b: a + b  
summed\_numbers = reduce(get\_sum, seq)  
summed\_numbers  
45

Вот еще один пример. Если sentences – это список предложений, и требуется подсчитать общее количество слов в этих предложениях, то можно написать, как показано в приведенном ниже интерактивном сеансе:

sentences = ["Варкалось.",   
... "Хливкие шорьки пырялись по наве, и",   
... "хрюкотали зелюки, как мюмзики в мове."]  
wsum = **lambda** aсс, sentence: aсс + len(sentence.split())  
number\_of\_words = reduce(wsum, sentences, 0)  
number\_of\_words  
13

В лямбда-функции, на которую ссылается переменная wsum, строковый метод split разбивает предложение на список слов, функция len подсчитывает количество элементов в получившемся списке и прибавляет его в накапливающую переменную.

В чем преимущества функций более высокого порядка?

* Они нередко состоят из одной строки.
* Все важные компоненты итерации – объект-последовательность, операция и возвращаемое значение – находятся в одном месте.
* Программный код в обычном цикле может повлиять на переменные, определенные перед ним, или которые следуют после него. По определению эти функции не имеют побочных эффектов.
* Они представляются собой элементарные операции. Глядя на цикл for, приходится построчно отслеживать его логику. При этом в качестве опоры для создания своего понимания программного кода приходится отталкиваться от нескольких структурных закономерностей. Напротив, функции более высокого порядка одновременно являются строительными блоками, которые могут быть интегрированы в сложные алгоритмы, и элементами, которые читатель кода может мгновенно понять и резюмировать в своем уме. «Этот код преобразовывает каждый элемент в новую последовательность. Этот отбрасывает некоторые элементы. А этот объединяет оставшиеся элементы в единый результат».
* Они имеют большое количество похожих функций, которые предоставляют возможности, которые служат дополнением к их основному поведению. Например, any, all или собственные их версии.

Приведем еще пару полезных функций.

Функция zip

Встроенная функция zip объединяет отдельные элементы из каждой последовательности в кортежи, т.е. она возвращает итерируемую последовательность, состоящую из кортежей. Вот общий формат функции zip:

zip(последовательность, последовательность, ...)

В данном формате *последовательность* – это итерируемая последовательность, т.е. список, кортеж, диапазон или строковые данные. Функция zip возвращает ленивый объект-последовательность, который нужно вычислить, чтобы увидеть результат. Приведенный ниже интерактивный сеанс это демонстрирует:

x = 'абв'  
y = 'эюя'  
zipped = zip(x, y)  
list(zipped)  
[('а', 'э'), ('б', 'ю'), ('в', 'я')]

В сочетании с оператором \* эта функция используется для распаковки объединенной последовательности (в виде пар, троек и т.д.) в отдельные кортежи. Приведенный ниже интерактивный сеанс это демонстрирует:

>>> x2, y2 = zip(\*zip(x, y))  
>>> x2  
('а', 'б', 'в')  
>>> y2  
('э', 'ю', 'я')  
>>> x == ''.join(x2) **and** y == ''.join(y2)  
**True**

Функция enumerate

Встроенная функция enumerate возвращает индекс элемента и сам элемент последовательности в качестве кортежа. Вот общий формат функции enumerate:

enumerate(последовательность)

В данном формате *последовательность* – это итерируемая последовательность, т.е. список, кортеж, диапазон или строковые данные. Функция enumerate возвращает ленивый объект-последовательность, который нужно вычислить, чтобы увидеть результат.

Например, в приведенном ниже интерактивном сеансе показано применение этой функции к списку букв. В результате ее выполнения будет получена ленивая последовательность со списком кортежей, где каждый кортеж представляет собой индекс и значение буквы.

lazy = enumerate(['а','б','в'])  
list(lazy)  
[(0, 'а'), (1, 'б'), (2, 'в')]

В строке 2 применена функция list, которая преобразовывает ленивую последовательность в список. Функция enumerate также позволяет применять заданную функцию к каждому элементу последовательности с учетом индекса:

convert = **lambda** tup: tup[1].upper() + str(tup[0])  
lazy = map(convert, enumerate(['а','б','в']))  
list(lazy)  
['А0', 'Б1', 'В2']

Функция convert в строке 1 переводит строковое значение второго элемента кортежа в верхний регистр и присоединяет к нему преобразованное в строковый тип значение первого элемента. Здесь tup – это кортеж, в котором tup[0] – это индекс элемента, и tup[1] – строковое значение элемента.

Включение в последовательность

Операции отображения и фильтрации встречаются так часто, что во многих языках программирования предлагаются способы написания этих выражений в более простых формах. Например, в языке Python возвести список чисел в квадрат можно следующим образом:

squared\_numbers = [x\*x **for** x **in** numbers]

Python поддерживает концепцию под названием «включение в последовательность» (от англ. comprehension, в информатике эта операция так же называется *описанием последовательности*), которая суть изящный способ преобразования одной последовательности в другую. Во время этого процесса элементы могут быть условно включены и преобразованы заданной функцией. Вот один из вариантов общего формата операции включения в список:

[выражение for переменная in список if выражение2]

В данном общем формате *выражение* – это выражение или функция с участием *переменной*, которые возвращают значение, *переменная* – это элемент последовательности, *список* – это обрабатываемый список, и *выражение2* – это логическое выражение или предикативная функция с участием *переменной*. Чтобы все стало понятно, приведем простой пример возведения список в квадрат без условия:

numbers = [1, 2, 3, 4, 5]  
squared\_numbers = [x\*x **for** x **in** numbers]  
squared\_numbers  
[1, 4, 9, 16, 25]

Приведенное выше включение в список эквивалентно следующему ниже фрагменту программного кода:

squared\_numbers = []  
**for** x **in** numbers:  
 squared\_numbers.append(x\*x)  
squared\_numbers  
[1, 4, 9, 16, 25]

Такая форма записи называется *синтаксическим сахаром*, т.е. добавленная синтаксическая конструкция, позволяющая записывать выражения в более простых и кратких формах. Неплохой аспект конструкций включения в последовательность состоит еще и в том, что они легко читаются на обычном языке, благодаря чему программный код становится чрезвычайно понятным.

В конструкции включения в последовательность используется математическая запись построения последовательности. Такая запись в теории множеств и логике называется определением интенсионала множества и описывает множество путем определения условия, которое должно выполняться для всех его членов. В сущности, в терминах этих областей науки, выполняя данную операцию в Python, мы «описываем интенсионал» соответственно списка, словаря, множества и итерируемой последовательности. Ниже приведены примеры описания интенсионала соответственно списка, словаря, множества и итерируемой последовательности.

**Таблица 1.** Формы описания интенсионала

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| [x\*x for x in numbers] | Описание списка |
| {x:x\*x for x in numbers} | Описание словаря |
| {x\*x for x in numbers}  set(x\*x for x in numbers) | Описание множества |
| (x\*x for x in numbers) | Описание последовательности. Такая форма записи создает генератор последовательности. *Генератор* – это объект, который можно последовательно обойти (обычно при помощи инструкции for), но чьи значения предоставляются только тогда, когда они требуются, используя ленивое вычисление. |

Отметим, что приведенные в таблице выражения (за исключением описания словаря) отличаются только ограничивающими символами: квадратные скобки применяются для описания списка, фигурные скобки – для описания словаря или множества и круглые скобки – для описания итерируемой последовательности.

Таким образом, примеры из разделов о функциях map и filter легко можно переписать с использованием включения в последовательность. Например, в строке 3 приведенного ниже интерактивного сеанса вместо функции map применена операция включения в список:

seq = (1, 2, 3, 4, 5, 6, 7, 8, 9)  
seq2 = (5, 6, 7, 8, 9, 0, 3, 2, 1)  
result = [x + y **for** x, y **in** zip(seq, seq2)]  
result  
[6, 8, 10, 12, 14, 6, 10, 10, 10]

Обратите внимание на квадратные скобки в определении – они сигнализируют, что в результате этой операции будет создан список. Также стоит обратить внимание, что при использовании в данной конструкции нескольких последовательностей применяется встроенная функция zip, которая в данном случае объединяет соответствующие элементы каждой последовательности в двухэлементные кортежи. (Если бы последовательностей было три, то они объединялись бы в кортежи из трех элементов и т.д.)

Включение в список применено и в приведенном ниже примере вместо функции filter:

result = [x **for** x **in** seq **if** is\_even(x)]  
result  
[2, 4, 6, 8]

Квадратные скобки в определении сигнализируют, что в результате этой операции будет создан список. Какой способ обработки последовательностей применять – с использованием функций более высокого порядка или включений, зачастую является предметом личных предпочтений.

Замыкание

Функции более высокого порядка не только получают функции на входе, но и могут порождать новые функции на выходе. Они даже в состоянии запоминать ссылку на значение в функции, которую они генерируют. Это называется *замыканием*. Функция, имеющая замыкание, может «запоминать» и получать доступ к среде вложенных в нее значений.

Используя замыкания, можно разделить исполнение функции со многими аргументами на большее количество шагов. Эта операция называется каррированием и обязана своим названием Хаскелю Каррингу. *Каррирование* – это преобразование функции многих аргументов в функцию, берущую свои аргументы по одному. Например, предположим, ваш программный код имеет приведенную ниже стандартную функцию adder:

**def** adder(n, m):  
 **return** n + m

Чтобы сделать ее каррированной, она должна быть переписана следующим образом:

**def** adder(n):  
 **def** fn(m):  
 **return** n + m  
 **return** fn

Это же самое можно выразить при помощи лямбда-функций:

adder = **lambda** n: **lambda** m: n + m

Обратите внимание, что в последнем примере используются две вложенные лямбда-функции, каждая из которых принимает всего один аргумент. В такой записи функция adder теперь может вызываться всего с одним аргументом. Выражение adder(3) возвращает не число, а новую, *каррированную функцию*. Во время вызова функции adder со значением 3 в качестве первого аргумента ссылка на значение 3 запоминается в каррированной функции. А дальше происходит следующее:

sum\_three = adder(3)  
sum\_three  
<function \_\_main\_\_.<**lambda**>.<locals>.<**lambda**>>  
sum\_three(1)  
4

В приведенном выше примере каррированная функция adder(3) присваивается переменной sum\_three, которая теперь на нее ссылается. Если вызвать функцию sum\_three, передав ей второй аргумент, то она вернет результат сложения двух аргументов 3 и 1.

Замыкания также используются для генерирования набора связанных функций по шаблону. Использование шаблона функции помогает делать программный код более читаемым и избегать дублирования. Давайте посмотрим на приведенный ниже пример:

**def** power\_generator(base):  
 **return** **lambda** x: pow(x, base)

Функция power\_generator может применяться для генерации разных функций, которые вычисляют степень:

square = power\_generator(2) # функция возведения в квадрат  
square(2)  
4  
cube = power\_generator(3) # функция возведения в куб  
cube(2)  
8

Отметим, что функции square и cube сохраняют значение переменной base. Эта переменная существовала только в среде power\_generator, несмотря на то, что эти возвращенные функции абсолютно независимы от функции power\_generator. Напомним еще раз: замыкание – это функция, которая имеет доступ к некоторым переменным за пределами своей собственной среды.

Замыкания могут также использоваться для управления внутренним состоянием функции. Давайте предположим, что требуется функция, которая накапливает сумму всех чисел, которые ей предоставляются. Один из способов это сделать состоит в использовании глобальной переменной:

COUNT = 0  
  
**def** count\_add(x):  
 **global** COUNT  
 COUNT += x  
 **return** COUNT

Как мы убедились, применение глобальных переменных следует избегать, потому что они загрязняют пространство имен программы. Более чистый подход состоит в использовании замыкания, чтобы включить ссылку на накапливающую переменную:

**def** make\_adder():  
 n = 0   
  
 **def** fn(x):  
 **nonlocal** n  
 n += x  
 **return** n  
  
 **return** fn

Такой подход позволяет создавать несколько счетчиков без применения глобальных переменных. Обратите внимание, что в этом примере использовано ключевое слово nonlocal, которое объявляет, что переменная n не является локальной для вложенной функции fn. В приведенном ниже интерактивном сеансе показано, как это работает:

my\_adder = make\_adder()  
print(my\_adder(5)) # напечатает 5  
print(my\_adder(2)) # напечатает 7 (5 + 2)  
print(my\_adder(3)) # напечатает 10 (5 + 2 + 3)

Некоторые языки программирования строго функциональны; весь код эквивалентен чистым математическим функциям. Эти языки заходят настолько далеко, что являются вневременными, причем порядок операторов в программном коде не вмешивается в поведение кода. В этих языках все присвоенные переменным значения являются немутируемыми. Такое присваивание называется *однократным*. Поскольку состояние программы отсутствует, то и нет момента времени, когда переменная может измениться. Вычисления в строгой функциональной парадигме просто сводятся к вычислению функций и сопоставлению с шаблонами.

Рекомендации по ФП на языке Python

Понятие ФП несколько различается по строгости формулировки. Одни понимают применение только функций, *немутируемость* и наведение мостов с периферией (вводом-выводом). Другие определяют ФП строже и наряду с немутируемостью говорят о применении только *чистых функций*. Но в любом случае программирование в функциональном стиле не тождественно функциональному программированию. Применение первоклассных функций, лямбд, итераторов, включений, каррирования и сопоставления с шаблонами вовсе не означает немутируемость и чистые функции.

*Программирование в функциональном стиле не тождественно функциональному программированию.*

Что делает функции нечистыми?

* Глобальные мутации, т.е. внесение изменений в глобальное состояние,
* Недетерминированность функций, т.е. которые для одинаковых входных значений могут возвращать разные результаты, и
* Операции ввода-вывода.

Пример глобальной мутации:

**def** append\_one(xs):  
 xs.append(1)  
 **return** xs  
   
xs = []  
print(append\_one(xs)) # Результат: [1]  
print(append\_one(xs)) # Результат: [1, 1]

Пример недетерминированности:

**import** random  
  
print(random.random())  
# Результат: 0.2334551699781765  
print(random.random())  
# Результат: 0.7073046734532323

Пример операции ввода-вывода:

**with** open('temp.txt', 'w') **as** f:  
 f.write('Hello!')  
   
**with** open('temp.txt', 'r') **as** f:  
 print(f.read()) # Результат: Hello!  
   
**with** open('temp.txt', 'w') **as** f:  
 f.write('Hi!')  
   
**with** open('temp.txt', 'r') **as** f:  
 print(f.read()) # Результат: Hi!

Из чистых функций вытекает ссылочная (референциальная) прозрачность. Говорят, что программа или математическое выражение ссылочно прозрачны, если любое подвыражение можно заменить его значением, и это не приведет к изменению значения целого, т. е. скрытые побочные эффекты отсутствуют. Математические рассуждения, преобразования и доказательства корректности могут быть справедливыми только для выражений, обладающих этим свойством. А программы, написанные на обычных императивных языках, не являются ссылочно прозрачными, так как присваивание значений глобальным переменным, в некоторых случаях и локальным, вызывает скрытые побочные эффекты.

Ссылочная прозрачность (1) улучшает тестопригодность программ, т.е. поведение подпрограмм не зависит от контекста, повторный запуск приложения дает одинаковый результаты как следствие отсутствия мутаций, (2) обеспечивается модульность, т.е. поведение функций не зависит от контекста, и чистые функции можно легко составлять в композиции, строя новые формы поведений, (3) упрощает обеспечение конкурентности из-за отсутствия необходимости в синхронизации, т.к. отсутствие совместных мутируемых данных делает синхронизацию ненужной.

Однако, ФП имеет свои *недостатки*, такие как новизна парадигмы и иногда ухудшение производительности программ. Но в нашем случае главный недостаток состоит в том, что язык Python, как таковой, не является языком функционального программирования. Например, в нем нет библиотеки по работе с неизменяемыми структурами данных и оптимизации стека под хвостовую рекурсию. Однако эффективное функциональное программирование на Python вполне возможно.

*Эффективное функциональное программирование на Python вполне возможно.*

В отличие от объектно-ориентированного программирования, которое строит сложные формы поведения с помощью наследования, ФП опирается на *композицию функций*. Этот принцип перекликается с философией Unix, состоящей из 2 правил:

* *Правило композиции* - строить программы так, чтобы иметь возможность легко их соединять с другими программами.
* *Правило модульности* - писать простые части, которые можно соединять чистыми интерфейсами.

Указанные выше два простых правила делают ненужными архитектурные шаблоны и принципы ООП, заменяя их функциями! А что, спросите вы, и классы тоже? В Python использование классов не противоречит ФП, если в них отсутствует мутирующие интерфейсы.

Пример класса с мутирующим интерфейсом:

**class** Dog:  
 **def** \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 **def** bark(self):  
 print('bark bark!') # !!  
  
 **def** birthday(self):  
 self.age += 1 # !!  
  
 **def** setbuddy(self, buddy):  
 self.buddy = buddy # !!

Пример класса без мутирующего интерфейса:

**class** Dog:  
 **def** \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 **def** description(self):  
 **return** f'{self.name} is {sound} years old'   
  
 **def** speak(self):  
 **return** f'{self.name} says {sound}'

Но лучше использовать замороженные dataclasses и копирование, где необходимо. Иными словами, все классы должны быть замороженными dataclasses.

**from** dataclasses **import** dataclass  
  
@dataclass(frozen=**True**)  
**class** Position:  
 name: str  
 lon: float = 0.0  
 lat: float = 0.0  
   
>>> pos = Position('Oslo', 10.8, 59.9)  
>>> pos.name  
'Oslo'  
>>> pos.name = 'Stockholm'  
dataclasses.FrozenInstanceError: cannot assign to field 'name'

При всем при этом dataclasses могут быть вполне себе умными!

@dataclass(frozen=**True**)  
**class** Coord:  
 row: int  
 col: int  
  
 **def** to\_tupple(self) -> Tuple[int, int]:  
 **return** self.row, self.col  
  
 @classmethod  
 **def** from\_str(cls, str\_coord: str) -> 'Coord':  
 xl\_row, xl\_col = coordinate\_ro\_tuple(str\_coord)  
 **return** cls(xl\_row - 1, xl\_col - 1)  
  
 **def** shift\_down(self, shift: int) -> 'Coord':  
 **return** self.\_\_class\_\_(self.row + shift, self.col)

Также следует использовать сторонние функциональные библиотеки (например, [toolz](https://toolz.readthedocs.io/en/latest/_modules/toolz/functoolz.html)), которые обеспечивают более оптимальную композиционность функций.

**from** toolz **import** pipe  
  
# h(g(f(data))) --> pipe(data, f, g, h)

Как вариант, использовать декларативные включения в список, включения в словарь и включения в множество в качестве замены функций map и filter, хотя эта рекомендация является факультативной.

И применять архитектурный шаблон «*немутируемое ядро - мутируемая оболочка*» (aka «функциональное ядро - императивная оболочка»), который позволяет выносить мутацию во вне и производить ее на границах приложения.

Выводы

Функциональное программирование сконцентрировано вокруг немутируемости и чистых функций. Чистота позволяет производить код, который более пригоден для тестирования, функциональных композиций и управления в конкурентной обстановке. Следует избегать мутирующих интерфейсов и стремиться использовать замороженные dataclasses, сторонние библиотеки наподобие toolz и включения, при этом оставаясь идиоматичным.