ГЛАВА 15

Основы функционального программирования

15.1 Принципы функционального программирования

Ключевые положения

Функциональное программирование представляет собой методику написания программного обеспечения, в центре внимания которой находятся функции. Функции могут присваиваться переменным, 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 объекты первого класса, т. е. все, что можно делать с "данными", можно делать и с функциями (наподобие передачи 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 почти повсеместно распространенный в императивных языках подход, при котором одной и той же переменной последовательно присваиваются различные 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, например 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.

15.2 Оператор *lambda*, функции *map*, *filter*, *reduce* и другие

Прежде чем продолжить, 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, разделенных запятой, *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
```

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.)

```
1 >>> lambda_function = lambda x, y: x + y
2 >>> lambda_function(5,7)
3 12
4 >>> func = lambda_function
5 >>> func(3,4)
6 7
```

```
7 >>> dic = {'функция1': lambda_function}
8 >>> dic['функция1'](7,8)
9 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.

Функция *map*

Нередко во время написания программы появляется необходимость преобразовать некую последовательность в другую. Для этих целей в Python имеется встроенная 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, *`последовательности`* — это одна или несколько разделенных запятыми итерируемых последовательностей, т. е. списки, кортежи, диапазоны или строковые данные.

```
1 >>> seq = (1, 2, 3, 4, 5, 6, 7, 8, 9)
2 >>> seq2 = (5, 6, 7, 8, 9, 0, 3, 2, 1)
3 >>> result = map(lambda_function, seq, seq2)
4 >>> result
5 <map object at 0x000002897F7C5B38>
6 >>> list(result)
7 [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, что 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.

Для того 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 ФП предлагают универсальную функцию `filter`, получающую набор элементов для фильтрации, и фильтрующую функцию, которая определяет, нужно 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, *`последовательность`* — это итерируемая последовательность, т. е. список, кортеж, диапазон или строковые данные.

Например, в *главе 5* была описана функция `is_even` для определения четности числа. Приведем ее еще раз в сжатом виде:

```
is_even = lambda x: x % 2 == 0
```

Для того 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*

Когда требуется обработать список значений таким образом, чтобы свести процесс к единственному результату, используется функция `reduce`. Функция `reduce` находится в модуле `functools` стандартной библиотеки. Чтобы показать, как она работает, приведем ее целиком.

```
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; *`инициализатор`* — это параметрическая переменная, которая получает начальное значение для накопителя. Начальным значением может быть значение любого примитивного типа данных либо мутирующий объект — список, кортеж и т. д. Начальное значение инициирует накапливающую переменную, которая до своего возвращения 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. Скажем, 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 так, как показано в приведенном ниже интерактивном сеансе:

```
>>> sentences = ["Варкалось.",
>>> ...           "Хливкие шорьки пырялись по наве, и",
>>> ...           "хрюкотали зелюки, как мюмзики в мове."]
```

```
>>> wsum = lambda acc, sentence: acc + 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 следующие после него. По определению функции более высокого порядка не имеют побочных эффектов.
- Они представляют собой элементарные операции. Глядя на цикл `for`, нужно 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 уме. "Этот код преобразует каждый элемент в новую последовательность. Этот отбрасывает некоторые элементы. А этот объединяет оставшиеся элементы в результат".
- Они имеют большое количество похожих функций, которые предоставляют возможности, служащие 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.

```
1 >>> lazy = enumerate(['а','б','в'])
2 >>> list(lazy)
3 [(0, 'а'), (1, 'б'), (2, 'в')]
```

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

```
1 >>> convert = lambda tup: tup[1].upper() + str(tup[0])
2 >>> lazy = map(convert, enumerate(['а','б','в']))
3 >>> list(lazy)
4 ['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, `tup[1]` — строковое значение элемента.

15.3 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*; подробнее о включении в последовательность *см. в главе 7* этой книги; в информатике эта операция также 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 включения в список, или *спискового включения*:

```
[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; *выражение2* — это логическое выражение или предикативная функция с участием переменной. Для того чтобы все стало понятно, приведем простой пример возведения списка 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 списковое включение эквивалентно следующему 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. Полезное свойство 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 списку, словарю, множеству и итерируемой последовательности. В табл. 15.1 приведены примеры.

Таблица 15.1. 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 генераторное включение в последовательность. *Генератор* — это объект, который можно последовательно обойти (обычно при помощи инструкции `for`), но чьи значения предоставляются только тогда, когда они требуются, с помощью ленивого вычисления

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, 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` применено списковое включение:

```
1 >>> seq = (1, 2, 3, 4, 5, 6, 7, 8, 9)
2 >>> seq2 = (5, 6, 7, 8, 9, 0, 3, 2, 1)
3 >>> result = [x + y for x, y in zip(seq, seq2)]
4 >>> result
5 [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.)

Списковое включение 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.

15.4 Замыкание

Функции более высокого порядка не только получают функции на входе, но и могут порождать новые функции на выходе. Кроме того, они в состоянии запоминать значения из своего лексического контекста, даже когда поток управления программы больше не находится в этом контексте. Это называется *лексическим замыканием*, или просто *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 каррингом. *Карринг* (или каррирование, curring) — 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
```

Для того 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 быть вызвана 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. Шаблон 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 собственной среды.

Замыкания могут также использоваться для управления внутренним состоянием функции. Предположим, что требуется функция, накапливающая 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, применения 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)
5
7
10
```

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 могут изменяться, т. е. мутировать. Программисты называют это *однократным присваиванием*. 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 шаблоном.

15.5 Функциональное ядро программы на основе конвейера

Главная задача этого раздела — показать один полезный архитектурный шаблон под названием "функциональное ядро — императивная оболочка", в котором функциональный код концентрируется внутри, а императивный код выносится наружу в попытке свести на нет недостатки каждого из них. Известно, что функциональные языки слабы при взаимодействии с "реальным миром", в частности с вводом данных пользователем, взаимодействием с графическим интерфейсом или другими операциями ввода-вывода. В рамках такого подхода весь код, связанный с вводом-выводом, выталкивается наружу, и внутри остается только функционально-ориентированный код.

Указанный подход задействует возможности Python по работе с функциями в рамках функциональной парадигмы, в которой функциями можно манипулировать точно так же, как и любыми другими объектами: передавать в качестве аргументов в другие функции, возвращать из функций и включать в последовательности в качестве их элементов.

Функциональный стиль программирования очень близок к тому, как размышляет человек во время решения задачи. "Пусть дано x. Для того чтобы решить задачу, необходимо выполнить с этими данными серию преобразований. Сначала применить к ним функцию f_1 и получить результирующие данные x'. Затем применить к новым данным другую функцию, f_2, и получить новые результирующие данные x'' и т. д.".

Как оказалось, такой образ мыслей отлично укладывается в то, что называется *конвейером обработки данных* (рис. 15.1). Конвейер обработки данных состоит из связанных между собой узлов, т. е. функций. Узел характеризуется набором входных и выходных каналов, по которым могут передаваться объекты. Узел ожидает появления определенного набора объектов на своем входном канале, после чего проводит вычисления и порождает объект (объекты) на своем выходном канале, которые передаются в следующий узел в конвейере.

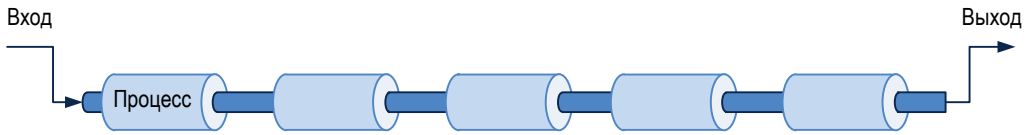

РИС. 15.1. Конвейер обработки данных

Конвейер позволяет имплементировать архитектурный шаблон "функциональное ядро — императивная оболочка" в полной мере, выталкивая императивный код наружу. Более того, он позволяет, во-первых, четче отслеживать входы и выходы каждого шага внутри конвейера, во-вторых, легко отлаживать любой шаг и соответственно отлавливать дефекты, вставляя шаг debug, и в-третьих, главное — концентрировать всю логику в одном месте, которую легко можно улавливать одним взглядом.

В функциональных языках конвейеры находят широкое применение, и для их имплементации даже существуют специальные синтаксические конструкции. Вот как выглядит конвейер в языке F#:

```
2
|> ( fun x -> x + 5)
|> ( fun x -> x * x)
|> ( fun x -> x.ToString() )
```

Здесь входные данные, в нашем случае число 2, последовательно обрабатываются серией лямбда-функций. Аналогичный конвейер можно имплементировать на языке Python, но для этого нужно написать специальную функцию, и, разумеется, это будет функция более высокого порядка:

```
# Конвейер обработки данных
def pipe(data, *fseq):
    for fn in fseq:
        data = fn(data)
    return data
```

Приведенный ниже пример демонстрирует работу конвейера:

```
pipe(2,
     lambda x: x + 5,
     lambda x: x * x,
     lambda x: str(x))
```

или в более удобном виде:

```
def add(x):      return lambda y: x + y
def square(x):   return x * x
def tostring(x): return str(x)

pipe(2,
     add(5),
     square,
     tostring)
```

Число 2 проходит серию преобразований, и в результате будет получено строковое значение '49'. По сравнению с функцией `reduce`, в которой переданная в качестве аргумента одна-единственная редуцирующая функция по очереди применяется к последовательности данных; в функции `pipe`, наоборот, последовательность функций применяется к обновляемым данным.

Функция `pipe` получает два аргумента: входные данные `data` и последовательность функций `fseq`. Во время первой итерации цикла `for` данные передаются в первую функцию из последовательности. Эта функция обрабатывает данные и возвращает результат, замещая переменную `data` новыми данными. Затем эти новые данные отправляются во вторую функцию и т. д. до тех пор, пока не будут выполнены все функции последовательности. По завершению своей работы функция `pipe` возвращает итоговые данные. Это и есть конвейер обработки данных.

ПРИМЕЧАНИЕ

В приведенном выше примере функции `pipe` использован оператор *упаковки* `*`. В зависимости от контекста оператор `*` служит для упаковки получаемых нескольких аргументов в одну параметрическую переменную либо распаковки списка передаваемых в функцию аргументов.

Когда он используется в параметре функции, как в приведенном выше примере, он служит для упаковки всех аргументов в одну параметрическую переменную. Например,

```
def my_sum(*args): # Упаковка в список
    return sum(args)

my_sum(1, 2, 3, 4, 5)
```

Когда он используется при вызове функции, он служит для разложения передаваемого списка на отдельные аргументы. Например,

```
def fun(a, b, c, d):
    print(a, b, c, d)

my_list = [1, 2, 3, 4]
fun(*my_list) # Разложение на четыре аргумента
```

Перед тем как рассматривать функциональный конвейер обработки данных детально, сначала для сравнения следует привести имплементацию конвейера в парадигме объектно-ориентированного программирования (программа 15.1).

Программа 15.1 (data_pipeline.py)

```
class Factory:
    def process(self, input):
        raise NotImplementedError

class Extract(Factory):
    def process(self, input):
        print("Идет извлечение...")
        output = {}
        return output

class Parse(Factory):
    def process(self, input):
        print("Идет разбор...")
        output = {}
        return output

class Load(Factory):
    def process(self, input):
        print("Идет загрузка...")
        output = {}
        return output

pipe = {
    "Извлечь"   : Extract(),
    "Разобрать" : Parse(),
    "Загрузить" : Load(),
}
inputs = {}
# Конвейерная обработка
for name, instance in pipe.items():
    inputs = instance.process(inputs)
```

Вывод программы

```
Идет извлечение...
Идет разбор...
Идет загрузка...
```

Здесь в цикле `for` результат на выходе из предыдущего шага подается на вход следующего шага, как того и требует стандартный конвейер. Однако проблема с объектно-ориентированным подходом заключается в том, что он неявно привносит всю свою объект-

но-ориентированную среду, о чем емко и шутливо высказался Джо Армстронг, создатель функционально-ориентированного языка Erlang:

"Вы хотели получить банан, но получили банан вместе с гориллой, которая его держит, и все джунгли в придачу".

То есть вам приходится соблюдать всю эту церемонию с `def`, `self`, `__init__`, атрибутами класса и атрибутами экземпляра, инстанцированиями, методами и прочими отличительными особенностями ООП.

В следующих рубриках *"В центре внимания"* будут рассмотрены примеры использования конвейера обработки данных на основе функциональной парадигмы программирования.

В ЦЕНТРЕ ВНИМАНИЯ

Функциональная имплементация вычисления факториала числа

В *главе 12*, посвященной рекурсии, была приведена рекурсивная имплементация алгоритма нахождения факториала числа. В приведенном далее примере показана нерекурсивная версия алгоритма вычисления факториала (`factorial`) и его рекурсивная версия на основе более эффективной хвостовой рекурсии (`factorial_rec`). Детали имплементации обеих функций в данном случае не важны. Программы приводятся в качестве примеров, на которых будет продемонстрирована работа конвейера обработки данных. Результат выполнения программы показан ниже.

Программа 15.2 **(factorial_functional.py)**

```
 1 # Эта программа демонстрирует
 2 # функциональную версию функции factorial из главы 12
 3
 4 def main():
 5     # Конвейер (ядро с нерекурсивным алгоритмом факториала)
 6     pipe(int(input('Введите неотрицательное целое число: ')),
 7          lambda n: (n, reduce(lambda x, y: x * y, range(1, n + 1))),
 8          lambda tup:
 9              print(f'Факториал числа {tup[0]} равняется {tup[1]}'))

# Вызвать главную функцию
main()
```

Вывод программы (ввод показан жирным шрифтом)

```
Введите неотрицательное целое число: 4 Enter
Факториал числа 4 равняется 24
```

В строке 8 лямбда-функция в последнем узле конвейера получает кортеж, состоящий из введенного пользователем числа и полученного результата.

В приведенную ниже расширенную версию программы вычисления факториала добавлена валидация входных данных, и алгоритмы выделены в отдельные функции. Чуть позже будет дано пояснение.

Программа 15.3 (factorial_functional2.py)

```
# Эта программа демонстрирует
# функциональную версию функции factorial из главы 12

def get_int(msg=''):
    return int(input(msg))

def main():
    # Алгоритм 1. Рекурсивная версия с хвостовой рекурсией
    def factorial_rec(n):
        fn = lambda n, acc=1: acc if n == 0 else fn(n - 1, acc * n)
        return n, fn(n)

    # Алгоритм 2. Нерекурсивная версия
    def factorial(n):
        return n, reduce(lambda x, y: x * y, range(1, n + 1))

    # Ввод данных
    def indata():
        def validate(n):  # Валидация входных данных
            if not isinstance(n, int):
                raise TypeError("Число должно быть целым.")
            if not n >= 0:
                raise ValueError("Число должно быть >= 0.")
            return n
        msg = 'Введите неотрицательное целое число: '
        return pipe(get_int(msg), validate)

    # Вывод данных
    def outdata():
        def fn(data):
            n, fact = data
            print(f'Факториал числа {n} равняется {fact}')
        return fn

    # Конвейер (функциональное ядро)
    pipe(indata(),     # вход: -       выход: int
         factorial,    # вход: int     выход: кортеж
         outdata())    # вход: кортеж  выход: -

# Вызвать главную функцию
main()
```

Вывод программы (вод показан жирным шрифтом)

```
Введите неотрицательное целое число: 4 Enter
Факториал числа 4 равняется 24
```

Функциональным ядром программы 15.2 являются строки 36–38:

```
pipe(indata(),
     factorial,
     outdata())
```

Они представлены конвейером из трех узлов, т. е. функциями `indata`, `factorial` и `outdata`. Функция `indata` занимается получением данных от пользователя, которые затем передаются по конвейеру дальше. Функция `factorial` является собственно обрабатывающим алгоритмом, в данном случае нерекурсивной функцией вычисления факториала, которая получает данные, их обрабатывает и передает по конвейеру дальше. И функция `outdata` получает данные и показывает их пользователю. Обратите внимание, что функция `indata` имеет собственный конвейер, который состоит из получения данных от пользователя и их валидации.

Следует отметить два важных момента. Во-первых, передаваемые от узла к узлу данные должны соответствовать какому-то определенному протоколу. Во-вторых, количество узлов может быть любым.

Такая организация программного кода:

- позволяет менять узлы конвейера на другие с целью тестирования различных и более эффективных имплементаций алгоритмов. Например, вместо нерекурсивной функции `factorial` можно поместить рекурсивную функцию `factorial_rec`:

  ```
  pipe(indata(), factorial_rec, outdata())
  ```

- облегчает проведение отладки программы, позволяя на каждом стыке вставлять отладочный код с целью проверки промежуточных результатов и тестирования производительности отдельных узлов.

Например, рассмотрим вторую возможность — отладку. В этом случае можно написать вспомогательную функцию `check`:

```
def check(data):
    print(data)
    return data
```

И затем ее вставить в конвейер, чтобы проверить результаты работы отдельных узлов конвейера:

```
pipe(indata(), check, factorial, check, outdata())
```

Если выполнить программу в таком варианте, то будут получены следующие результаты.

Вывод программы (ввод показан жирным шрифтом)

```
Введите неотрицательное целое число: 4 Enter
4
(4, 24)
Факториал числа 4 равняется 24
```

Как видно из результатов, на вход в функцию `factorial` поступает введенное пользователем значение 4, а на выходе из нее возвращается кортеж с исходным числом и полученным результатом (4, 24). Этот результат показывает, что программа работает, как и ожидалось. Альтернативно вместо проверочной функции можно написать функцию-таймер, которая могла бы хронометрировать отдельные узлы конвейера.

Приведем еще пару примеров с аналогичной организацией программного кода на основе функционального ядра в виде конвейера.

В ЦЕНТРЕ ВНИМАНИЯ

Функциональная имплементация вычисления последовательности Фибоначчи

Программа 15.4 **(fibonacci_functional.py)**

```
# Эта программа демонстрирует
# функциональную версию функции fibonacci из главы 12

def main():
    # Алгоритм
    def fibonacci(n, x=0, y=1):
        # Функция fib возвращает n-е число последовательности.
        fib = lambda n, x=0, y=1: x if n <= 0 else fib(n - 1, y, x + y)
        # Функция reduce собирает результаты в список acc
        acc = []
        reduce(lambda _, y: acc.append(fib(y)), range(n + 1))
        return n, acc

    # Валидация входных данных
    def validate(n):
        if not isinstance(n, int):
            raise TypeError("Число должно быть целым.")
        if not n >= 0:
            raise ValueError("Число должно быть нулем или положительным.")
        if n > 10:
            raise ValueError("Число должно быть не больше 10.")
        return n

    # Ввод данных
    def indata():
        msg = 'Введите неотрицательное целое число не больше 10: '
        return pipe(get_int(msg), validate)
```

```
    # Вывод данных
    def outdata():
        def fn(data):
            n, seq = data
            msg = f'Первые {n} чисел последовательности Фибоначчи:'
            print(msg)
            [print(el) for el in seq]
        return fn

    # Конвейер (функциональное ядро)
    pipe(indata(), fibonacci, outdata())

# Вызвать главную функцию.
main()
```

Вывод программы (ввод показан жирным шрифтом)

```
Введите неотрицательное целое число не больше 10: 10 Enter
Первые 10 чисел последовательности Фибоначчи:
1
1
2
3
5
8
13
21
34
55
```

В ЦЕНТРЕ ВНИМАНИЯ

Функциональная имплементация суммирования диапазона значений последовательности

Программа 15.5 **(range_sum_functional.py)**

```
# Эта программа демонстрирует
# функциональную версию функции range_sum из главы 12

def main():
    # Алгоритм
    def range_sum(data):
        seq, params = data
        fn = lambda start, end: 0 if start > end \
                                  else seq[start] + fn(start + 1, end)
```

```
        return fn(*params)

    # Ввод данных
    def indata():
        seq = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        params = (2,5)   # params - это параметры start, end
        return seq, params

    # Вывод данных
    def outdata():
        def f(data):
            msg = 'Сумма значений со 2 по 5 позицию равняется '
            print(msg, format(data), sep='')
        return f

    # Конвейер (функциональное ядро)
    pipe(indata(), range_sum, outdata())

# Вызвать главную функцию.
main()
```

Вывод программы

```
Сумма значений со 2 по 5 позицию равняется 18
```

Приведенный в настоящей главе материал носит ознакомительный характер и предназначен для того, чтобы продемонстрировать возможности функциональной парадигмы программирования на Python с целью дальнейших самостоятельных исследований и побудить начинающих программистов заняться языком Python углубленно. Вопросы для повторения в настоящей главе отсутствуют, а некоторые расширенные возможности языка Python, такие как мемоизация, ленивые вычисления, генераторы, сопоставление с шаблоном, мультиметоды и прочие темы, не рассмотрены. Основная задача этой главы — познакомить с возможностями Python по работе с функциями, которыми можно манипулировать точно так же, как и любыми другими объектами: присваивать переменным, передавать в качестве аргументов в другие функции, возвращать из функций и включать в последовательности в качестве их элементов.