pełnego potencjału programowania funkcyjnego, gdy ma się jedynie doświadczenie w imperatywnym programowaniu obiektowym. Lisp nie jest czysto funkcyjny, ale kładzie większy nacisk na programowanie funkcyjne niż język Python.

Tworzenie czystych funkcji

Gdy piszemy kod w stylu funkcyjnym, nasze funkcje są zaprojektowane tak, aby nie miały żadnych efektów ubocznych. Powinny jedynie pobierać dane wejściowe i zwracać wynik bez zachowywania stanu bądź modyfikowania czegokolwiek, co nie jest odzwierciedlone w wartości zwrotnej. Funkcje zgodne z tą koncepcją nazywane są *czysto funkcyjnymi*.

Zacznijmy od przykładu zwykłej funkcji, która nie jest czysto funkcyjna i usuwa ostatni element z listy:

```
def remove_last_item(mylist):
    """Usuwa ostatni element z listy."""
    mylist.pop(-1)  # Modyfikuje mylist
```

A oto czysta wersja tej samej funkcji:

```
def butlast(mylist):

    return mylist[:-1]  # Zwraca kopię mylist
```

Definiujemy funkcję `butlast()` w taki sposób, aby działała ona jak `butlast` w Lispie, zwracając listę bez ostatniego elementu, *nie* modyfikując oryginalnej listy. Zamiast tego zwraca kopię listy, w której wprowadzone zostały zmiany, umożliwiając zachowanie oryginału.

Praktyczne zalety programowania funkcyjnego to m.in.:

Modułowość Funkcyjny styl pisania kodu zmusza do wprowadzenia pewnego rozdziału podczas rozwiązywania poszczególnych problemów i ułatwia ponowne wykorzystywanie fragmentów kodu w innym kontekście. Ponieważ funkcja nie zależy od żadnej zewnętrznej zmiennej ani stanu, wywołanie jej z innego miejsca w kodzie jest bardzo proste.

Zwięzłość Programowanie funkcyjne jest zwykle mniej rozwlekłe niż inne paradygmaty programowania.

Współbieżność Czysto funkcyjne funkcje są bezpieczne wątkowo i mogą być uruchamiane równolegle. Niektóre języki funkcyjne robią to w sposób automatyczny, co może być bardzo pomocne w przyszłości, gdy pojawia się potrzeba skalowania aplikacji. Python nie oferuje jeszcze takiej możliwości.

Testowalność Testowanie programu funkcyjnego jest niesamowicie proste: wystarczy zestaw danych wejściowych i zestaw oczekiwanych

wyników. Funkcje są *idempotentne*, co oznacza, że każde z wielu wywołań tej samej funkcji z tymi samymi argumentami zwróci zawsze ten sam wynik.

Generatory

Generator jest obiektem, który zachowuje się jak iterator pod tym względem, że generuje i zwraca wartość w każdym wywołaniu metody `next()`, do momentu zgłoszenia wyjątku `StopIteration`. Generatory, wprowadzone po raz pierwszy w PEP 255, oferują prosty sposób tworzenia obiektów, które implementują *protokół iteratora*. Choć zasadniczo generatorów nie trzeba implementować w stylu funkcyjnym, ułatwia to ich pisanie oraz debugowanie i jest powszechną praktyką.

Aby utworzyć generator, wystarczy napisać zwykłą funkcję Pythona, która zawiera instrukcję `yield`. Python wykryje użycie instrukcji `yield` i oznaczy funkcję jako generator. Gdy nadchodzi czas wykonania instrukcji `yield`, funkcja zwraca wartość jak w przypadku użycia instrukcji `return`, z jedną istotną różnicą: interpreter zapisze odwołanie do stosu, które zostanie wykorzystane do wznowienia wykonania funkcji, gdy funkcja `next()` zostanie ponownie wywołana.

Gdy funkcje są wykonywane, łańcuch ich wykonań tworzy *stos* – mówi się, że wywołania funkcji są umieszczane na stosie. Gdy funkcja zwraca wartość, zostaje usunięta ze stosu i zwracana przez nią wartość zostaje przekazana do funkcji wywołującej. W przypadku generatora funkcja tak naprawdę nie zwraca wartości, a jedynie ją *przekazuje* (*yield*). Dlatego Python zapisuje stan funkcji w postaci odwołania do stosu, wznawiając wykonanie generatora od zapisanego miejsca, gdy potrzebna jest następna iteracja generatora.

Tworzenie generatora

Jak wspomnieliśmy, generator tworzymy, pisząc normalną funkcję i umieszczając w niej instrukcję `yield`. Listing 8.1 służy do utworzenia generatora o nazwie `mygenerator()`, który zawiera trzy instrukcje `yield`, co oznacza, że dokona iteracji podczas trzech kolejnych wywołań funkcji `next()`.

```
>> def mygenerator():
...     yield 1
...     yield 2
...     yield 'a'
...
>>> mygenerator()
<generator object mygenerator at 0x10d77fa50>
>>> g = mygenerator()
>>> next(g)
1
```