```
>>> list(zip([1, 2, 3], [2, 3, 4, 5]))          # N sekwencji: zwraca krotki N-elementowe
[(1, 2), (2, 3), (3, 4)]
# map przekazuje połączone elementy do funkcji, przycina do najkrótszej sekwencji
>>> list(map(abs, [ 2,-1, 0, 1, 2]))           # Jedna sekwencja: wywołanie funkcji 1-argumentowej
[2, 1, 0, 1, 2]
>>> list(map(pow, [1, 2, 3], [2, 3, 4, 5]))    # N sekwencji: wywołanie funkcji N-argumentowej
[1, 8, 81]
```

Choć mechanizmy te powstały w nieco innym celu, to warto zwrócić uwagę na analogię, z jaką działają funkcje `map` i `zip` na większej liczbie sekwencji, ponieważ tę właściwość wykorzystamy w kolejnym przykładzie.

Tworzymy własną implementację funkcji map

Choć funkcje wbudowane `map` i `zip` są szybkie i wygodne w użyciu, zawsze istnieje możliwość zaemulowania ich na własną rękę. W poprzednim rozdziale na przykład mieliśmy okazję stworzyć funkcję emulującą funkcję wbudowaną `map`, obsługującą tylko jeden argument sekwencji. Uzupełnienie tej implementacji o obsługę większej liczby sekwencji nie stanowi jednak problemu.

```
# map(func, seqs...): emulacja z użyciem funkcji zip

def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        res.append(func(*args))
    return res

print(mymap(abs, [ 2,-1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))
```

Ta wersja intensywnie wykorzystuje specjalną składnię `*args` służącą do przekazywania argumentów między wywoływanymi funkcjami. Funkcja rozpakowuje swoje argumenty (będące sekwencjami, a właściwie obiektami iterowanymi) za pomocą funkcji `zip`, łącząc je w nowe sekwencje, po czym wywołuje funkcję przekazaną w pierwszym argumencie z użyciem tych nowych sekwencji jako argumentów. Wykorzystujemy zaobserwowany wyżej fakt, że operacja `zip` jest w zasadzie zagnieżdżoną operacją `map`. Test poniżej definicji funkcji sprawdza działanie naszego generatora dla dwóch sekwencji (wyniki są identyczne z tymi, które zwróciłaby wbudowana funkcja `map`).

```
[2, 1, 0, 1, 2]
[1, 8, 81]
```

W rzeczywistości ta wersja wykorzystuje klasyczny wzorzec znany z *list składanych*, budując listę wyników w pętli `for`. Naszą funkcję `map` możemy zatem zbudować w sposób bardziej zwarty, wykorzystując jednowierszowe wyrażenie listy składanej.

```
# Użycie listy składanej

def mymap(func, *seqs):
    return [func(*args) for args in zip(*seqs)]

print(mymap(abs, [ 2,-1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))
```

Uruchomione testy zwracają taki sam wynik jak poprzednio, ale kod jest krótszy i prawdopodobnie będzie szybszy (więcej informacji na temat pomiarów wydajności przedstawię w punkcie „Pomiary wydajności implementacji iteratorów” w dalszej części rozdziału). Obie z przedsta-