Wydanie III

python™

Programowanie funkcyjne w Pythonie

Jak pisać zwięzły, wydajny i ekspresywny kod

Steven F. Lott

Helion

‹packt›

Tytuł oryginału: Functional Python Programming: Use a functional approach to write succinct, expressive, and efficient Python code, 3rd Edition

Tłumaczenie: Radosław Meryk

ISBN: 978-83-289-0063-9

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/pytpf3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/pytpf3.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Rozdział 5

Funkcje wyższego rzędu

Bardzo ważną cechą paradygmatu programowania funkcyjnego są funkcje wyższego rzędu. Przyjrzymy się następującym trzem odmianom funkcji wyższego rzędu:

- Funkcje, które pobierają funkcje jako jeden (lub więcej) swoich argumentów.
- Funkcje, które zwracają funkcję.
- Funkcje, które pobierają funkcję i zwracają funkcję — tzn. połączenie dwóch poprzednich funkcji.

W tym rozdziale przyjrzymy się wbudowanym funkcjom wyższego rzędu. W kolejnych rozdziałach przyjrzymy się kilku modułom bibliotecznym oferującym funkcje wyższego rzędu.

Wśród funkcji, które pobierają funkcje i tworzą funkcje, można znaleźć zarówno złożone klasy wywoływalne, jak i funkcje dekoratorów. Dokładniejszy opis dekoratorów odłożymy do rozdziału 12. „Techniki projektowania dekoratorów".

W tym rozdziale przyjrzymy się następującym funkcjom:

- `max()` i `min()`;
- `map()`;
- `filter()`;
- `iter()`;
- `sorted()`.

Dodatkowo przyjrzymy się funkcji `itemgetter()` z modułu `operator`. Ta funkcja jest przydatna do wyodrębniania elementu z sekwencji.

Przyjrzymy się także formatowi wyrażeń lambda, których można użyć w celu uproszczenia korzystania z funkcji wyższego rzędu.

Funkcje `max()` i `min()` są redukcjami — tworzą pojedynczą wartość z kolekcji. Pozostałe funkcje to mapowania. Nie redukują wartości wejściowej do pojedynczej wartości.

Wskazówka

Funkcje `max()`, `min()` i `sorted()` mają zarówno zachowania domyślne, jak i zachowania funkcji wyższego rzędu. Funkcję można dostarczyć za pomocą argumentu `key=`. Dla tych funkcji istnieje konkretne zachowanie domyślne.

Funkcje `map()` i `filter()` przyjmują funkcję jako pierwszy argument pozycyjny. Tutaj funkcja jest obowiązkowa, ponieważ nie istnieje konkretne zachowanie domyślne.

Szereg funkcji wyższego rzędu jest dostępnych w module `itertools`. Przyjrzymy się temu modułowi w rozdziale 8. „Moduł itertools", a także w rozdziale 9. „Moduł itertools dla kombinatoryków — permutacje i kombinacje".

Ponadto w module `functools` jest dostępna funkcja ogólnego przeznaczenia `reduce()`. Ponieważ korzystanie z tej funkcji wymaga nieco więcej uwagi, przyjrzymy się jej bliżej w rozdziale 10. „Moduł functools". Powinniśmy unikać sytuacji, kiedy przekształcanie nieefektywnego algorytmu przeradza się w koszmar nadmiernego przetwarzania.

Wykorzystanie funkcji max() i min() do wyszukiwania ekstremów

Funkcje `max()` i `min()` „wiodą podwójne życie". Są prostymi funkcjami mającymi zastosowanie do kolekcji. Są również funkcjami wyższego rzędu. Ich domyślne zachowanie możemy zaobserwować w następującym kodzie:

```
>>> max(1, 2, 3)
3
>>> max((1,2,3,4))
4
```

Obie funkcje przyjmują nieokreśloną liczbę argumentów. Funkcje są zaprojektowane tak, aby akceptowały sekwencję lub obiekt iterowalny jako jedyny argument i lokalizowały wartość maksymalną (lub minimalną) tego obiektu iterowalnego. Po zastosowaniu do kolekcji mapowania lokalizują maksymalną (lub minimalną) wartość klucza.

Wykonują również bardziej wyrafinowane działania. Załóżmy, że mamy dane podróży z przykładów w rozdziale 4. „Praca z kolekcjami". Mamy funkcję generującą sekwencję krotek, która wygląda następująco:

```
[
((37.54901619777347, -76.33029518659048), (37.840832, -76.273834),
17.7246),
((37.840832, -76.273834), (38.331501, -76.459503), 30.7382),
((38.331501, -76.459503), (38.845501, -76.537331), 31.0756),
((36.843334, -76.298668), (37.549, -76.331169), 42.3962),
```

```
((37.549, -76.331169), (38.330166, -76.458504), 47.2866),
((38.330166, -76.458504), (38.976334, -76.473503), 38.8019)
]
```

Każda krotka w tej kolekcji składa się z trzech wartości: lokalizacji początkowej, końcowej i odległości pomiędzy nimi. Lokalizacje są podane za pomocą par złożonych z szerokości i długości geograficznej. Wschodnia szerokość geograficzna jest dodatnia, więc są to punkty wzdłuż wschodniego wybrzeża USA, około 76° na zachód. Odległości między punktami są wyrażone w milach morskich.

Istnieją trzy sposoby uzyskania maksymalnych i minimalnych odległości z tej sekwencji wartości. Oto one:

- Wyodrębnienie odległości za pomocą funkcji generatorowej. W ten sposób uzyskamy tylko odległości, ponieważ odrzuciliśmy pozostałe dwa atrybuty każdego odcinka. Takie rozwiązanie nie zadziała dobrze, jeśli będziemy mieli jakiekolwiek dodatkowe wymagania co do przetwarzania długości bądź szerokości geograficznej.
- Wykorzystanie wzorca *rozpakuj*(*przetwarzaj*(*opakuj*())). Za jego pomocą uzyskamy najdłuższy i najkrótszy odcinek. Na tej podstawie możemy w razie potrzeby wyodrębnić odległość lub punkt.
- Wykorzystanie `max()` i `min()` jako funkcji wyższego rzędu — wstawienie funkcji, która wykonuje ekstrakcję istotnych wartości odległości. Spowoduje to również zachowanie obiektów wejściowych ze wszystkimi ich atrybutami.

W celu zaprezentowania kontekstu poniżej zamieszczono skrypt, który buduje obiekt `trip` reprezentujący podróż:

```
>>> from Chapter04.ch04_ex1 import (
...     floats_from_pair, float_lat_lon, row_iter_kml, haversine, legs ... )
>>> import urllib.request
>>> data = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(data) as source:
...     path = floats_from_pair(float_lat_lon(row_iter_kml(source)))
...     trip = list(
...         (start, end, round(haversine(start, end), 4))
...         for start, end in legs(path)
...     )
```

Wynikowy obiekt `trip` jest listą zawierającą pojedyncze odcinki. Każdy odcinek jest trójelementową krotką składającą się z punktu początkowego, końcowego oraz odległości obliczanej za pomocą funkcji `haversine()`. Na podstawie ogólnej ścieżki punktów w oryginalnym pliku KML funkcja `leg()` tworzy pary punktów początek i koniec. W celu materializacji listy odcinków funkcja `list()` pobiera wartości z leniwie wartościowanego generatora.

Po uzyskaniu obiektu `trip` możemy wyodrębnić odległości i obliczyć ich wartości maksymalną i minimalną. Kod służący do tego celu i korzystający z funkcji generatorowej wygląda następująco:

```
>>> longest = max(dist for start, end, dist in trip)
>>> shortest = min(dist for start, end, dist in trip)
```

Aby wyodrębnić odpowiedni element z każdego odcinka należącego do krotki `trip`, użyliśmy funkcji generatorowej. Musieliśmy powtórzyć wyrażenie generatorowe, ponieważ wyrażenie `dist for start, end, dist in trip` może być wykorzystane tylko raz.

Oto wyniki uzyskane na podstawie większego zbioru danych niż ten, który pokazano wcześniej:

```
>>> longest
129.7748
>>> shortest
0.1731
```

Pomocne może okazać się odwołanie się do rozdziału 2. „Podstawowe pojęcia programowania funkcyjnego" w celu zapoznania się z przykładami wzorca projektowego `opakuj-przetwarzaj-rozpakuj`.

Poniżej zamieszczono wersję z wykorzystaniem wzorca *rozpakuj*(*przetwarzaj*(*opakuj*())).

```
from collections.abc import Iterator, Iterable
from typing import Any

def wrap(leg_iter: Iterable[Any]) -> Iterable[tuple[Any, Any]]:
    return ((leg[2], leg) for leg in leg_iter)

def unwrap(dist_leg: Tuple[Any, Any]) -> Any:
    distance, leg = dist_leg
    return leg
```

Z funkcji można skorzystać w następujący sposób:

```
>>> longest = unwrap(max(wrap(trip)))
>>> longest
((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)

>>> short = unwrap(min(wrap(trip)))
>>> short
((35.505665, -76.653664), (35.508335, -76.654999), 0.1731)
```

W ostatnim i najważniejszym formacie wykorzystaliśmy cechę wyższego rzędu funkcji `max()` i `min()`. Najpierw zdefiniujemy funkcję pomocniczą, a następnie wykorzystamy ją do zredukowania kolekcji odcinków do żądanych podsumowań za pomocą poniższego fragmentu kodu:

```
def by_dist(leg: Tuple[Any, Any, Any]) -> Any:
    lat, lon, dist = leg
    return dist
```

Możemy użyć tej funkcji jako wartości argumentu `key=` wbudowanej funkcji `max()`. Można to zrobić w następujący sposób:

```
>>> longest = max(trip, key=by_dist)
>>> longest
((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)

>>> short = min(trip, key=by_dist)
>>> short
((35.505665, -76.653664), (35.508335, -76.654999), 0.1731)
```

Funkcja `by_dist()` wybiera z krotki każdego odcinka trzy elementy i zwraca element reprezentujący odległość. Użyjemy jej z funkcjami `max()` i `min()`.

Funkcje `max()` i `min()` pobierają jako argumenty zarówno iterację, jak i funkcję. Wiele funkcji wyższego rzędu w Pythonie do dostarczenia funkcji, która będzie używana do wyodrębnienia niezbędnej wartości klucza, używa parametru ze słowem kluczowym `key=`.

Korzystanie z formatu wyrażeń lambda w Pythonie

W wielu przypadkach wydaje się, że definicja funkcji pomocniczej to zbyt wiele kodu. Często możemy sprowadzić funkcję `key=` do pojedynczego wyrażenia. Konieczność pisania instrukcji `def` i `return`, aby opakować pojedyncze wyrażenie, może wydawać się marnotrawstwem.

Python oferuje format lambda jako sposób na uproszczenie korzystania z funkcji wyższego rzędu. Format lambda pozwala zdefiniować niewielką funkcję anonimową. Treść funkcji ogranicza się do pojedynczego wyrażenia.

Oto przykład użycia prostego wyrażenia `lambda` jako funkcji `key=`:

```
>>> longest = max(trip, key=lambda leg: leg[2])
>>> shortest = min(trip, key=lambda leg: leg[2])
```

W tym przypadku do wyrażenia lambda przekażemy poszczególne trójelementowe krotki reprezentujące odcinki. Do zmiennej `leg` argumentu wyrażenia lambda jest podstawiana krotka, a następnie wyznaczana jest wartość `leg[2]`, która z trójelementowej krotki pobiera odległość.

W przypadkach, gdy wyrażenie lambda jest używane dokładnie raz, ten format jest idealny. Jeśli wykorzystujemy wyrażenia lambda wielokrotnie, powinniśmy unikać kopiowania i wklejania. W powyższym przykładzie wyrażenie lambda jest powtarzane, co może powodować potencjalne problemy z utrzymaniem oprogramowania. Jaka jest alternatywa?

Możemy przypisać wyrażenia lambda do zmiennych. Na przykład:

```
start = lambda x: x[0]
end = lambda x: x[1]
dist = lambda x: x[2]
```

Każdy z tych formatów wyrażeń lambda jest obiektem wywoływalnym, podobnym do zdefiniowanej funkcji. Można ich używać tak jak funkcji.

Poniższy przykład przedstawia interaktywną sesję:

```
>>> longest = ((27.154167, -80.195663), (29.195168, -81.002998),
129.7748)
>>> dist(longest)
129.7748
```

Oto dwa powody, dla których warto unikać tej techniki:

- PEP 8, standard stylu kodowania w Pythonie odradza przypisywanie obiektów lambda do zmiennych. Aby uzyskać więcej informacji zobacz *https://peps.python.org/pep-0008/*.
- Moduł `operator` udostępnia ogólny moduł pobierania elementów, `itemgetter()`. Jest to funkcja wyższego rzędu zwracająca funkcję, której możemy użyć zamiast obiektu lambda.

Aby rozszerzyć ten przykład, przyjrzymy się, jak uzyskać wartości szerokości lub długości geograficznej punktu początkowego lub końcowego.

Oto kontynuacja wcześniejszej interaktywnej sesji:

```
>>> from operator import itemgetter
>>> start = itemgetter(0)
>>> start(longest)
(27.154167, -80.195663)

>>> lat = itemgetter(0)
>>> lon = itemgetter(1) >>> lat(start(longest))
27.154167
```

Najpierw zaimportowaliśmy funkcję `itemgetter()` z modułu `operator`. Wartość zwracana przez tę funkcję to funkcja, która pobiera żądany element z sekwencji. W pierwszej części przykładu funkcja `start()` wyodrębni z sekwencji element 0.

Podobnie funkcja `itemgetter()` tworzy funkcje `lat()` i `lon()`. Zwróćmy uwagę, że złożoność zagnieżdżonych krotek w strukturze danych musi być dokładnie dopasowana do funkcji `itemgetter()`.

Nie ma wyraźnej przewagi z używania obiektów lambda lub funkcji `itemgetter()` jako sposobu wyodrębniania pól zamiast definiowania klasy `typing.NamedTuple` lub klasy danych. Z drugiej strony wyrażenia lambda (lub funkcja `itemgetter()`) pozwalają na korzystanie z notacji prefiksowej, która może być czytelniejsza w kontekście programowania funkcyjnego. Podobną korzyść możemy uzyskać dzięki użyciu do wyodrębnienia określonego atrybutu z klasy `typing.NamedTuple` lub klasy danych funkcji `operator.attrgetter`. Użycie funkcji `attrgetter` powiela nazwę. Na przykład klasa `typing.NamedTuple` z atrybutem `lat` może również używać wywołania `attrgetter ↪('lat')`. Może to nieco utrudnić zlokalizowanie wszystkich odwołań do atrybutu podczas refaktoryzacji.

Wyrażenia lambda i rachunek lambda

Gdyby Python był czysto funkcyjnym językiem programowania, byłoby konieczne wyjaśnienie rachunku lambda Churcha i techniki wymyślonej przez Haskella Curry'ego określanej jako **rozwijanie funkcji** (ang. *currying*). Python nie trzyma się jednak ściśle tego rodzaju rachunku lambda. Funkcje nie są rozwijane w celu zredukowania ich do jednoargumentowych wyrażeń lambda.

Format wyrażeń lambda w Pythonie nie ogranicza się do funkcji jednoargumentowych. Lambdy mogą mieć dowolną liczbę argumentów. Są jednak ograniczone do pojedynczego wyrażenia.

Rozwijanie funkcji można zaimplementować za pomocą funkcji `functools.partial`. Omówienie tego tematu odłożymy do rozdziału 10. „Moduł functools".

Korzystanie z funkcji map() w celu zastosowania funkcji do kolekcji

Funkcja skalarna mapuje wartości z dziedziny na zakres. Kiedy spojrzymy na przykład funkcji `math.sqrt()`, widzimy mapowanie wartości x typu `float` na inną wartość `float y = sqrt(x)`, taką że $y^2 = x$. Dziedzina dla modułu `math` jest ograniczona do wartości nieujemnych. W przypadku korzystania z modułu `cmath` można użyć dowolnych liczb, a wyniki mogą być liczbami zespolonymi.

Funkcja `map()` wyraża podobną koncepcję — mapuje wartości z jednej kolekcji, aby utworzyć inną kolekcję. To daje pewność, że podana funkcja zostanie użyta do zmapowania każdego pojedynczego elementu z kolekcji reprezentującej dziedzinę na kolekcję zakresu — idealny sposób zastosowania funkcji wbudowanej do zbioru danych.

Nasz pierwszy przykład obejmuje parsowanie bloku tekstu w celu uzyskania sekwencji liczb. Załóżmy, że mamy następujący fragment tekstu:

```
>>> text= """\
... 2 3 5 7 11 13 17 19 23 29
... 31 37 41 43 47 53 59 61 67 71
... 73 79 83 89 97 101 103 107 109 113
... 127 131 137 139 149 151 157 163 167 173
... 179 181 191 193 197 199 211 223 227 229
... """
```

Możemy zmienić strukturę tego tekstu za pomocą następującej funkcji generatorowej:

```
>>> data = list(
...     v
...     for line in text.splitlines()
...     for v in line.split()
... )
```

Wykonanie tego kodu spowoduje podzielenie tekstu na wiersze. Powyższy kod dzieli każdy wiersz na wyrazy rozdzielone spacjami i iteruje po ciągach uzyskanych w wyniku. Wynik ma następującą postać:

```
['2', '3', '5', '7', '11', '13', '17', '19', '23', '29',
'31', '37', '41', '43', '47', '53', '59', '61', '67', '71',
'73', '79', '83', '89', '97', '101', '103', '107', '109', '113',
'127', '131', '137', '139', '149', '151', '157', '163', '167',
'173', '179', '181', '191', '193', '197', '199', '211', '223', '227', '229']
```

Do każdej z wartości tekstowych nadal trzeba zastosować funkcję `int()`. Do tego doskonale nadaje się funkcja `map()`. Przyjrzyjmy się poniższemu przykładowi kodu:

```
>>> list(map(int, data))
[2, 3, 5, 7, 11, 13, 17, 19, ..., 229]
```

Funkcja `map()` zastosowała funkcję `int()` do każdej wartości w kolekcji. Wynik jest sekwencją złożoną z liczb, a nie sekwencją ciągów znaków.

Wyniki funkcji `map()` są iterowalne. Funkcja `map()` może przetwarzać dowolny obiekt iterowalny.

Idea jest taka, że przy użyciu funkcji `map()` może być zastosowana do elementów kolekcji każda funkcja Pythona. Istnieje wiele wbudowanych funkcji, które można wykorzystać w kontekście przetwarzania z mapowaniem.

Wykorzystanie wyrażeń lambda i funkcji map()

Powiedzmy, że chcemy dokonać konwersji odległości z mil morskich na lądowe. Chcemy pomnożyć odległość każdego odcinka przez 6076,12/5280, czyli 1,150780.

Do wyodrębnienia danych ze struktury danych użyjemy kilku funkcji `itemgetter`. Operacje wyodrębniania możemy połączyć z obliczaniem nowych wartości. Takie obliczenia możemy wykonać za pomocą funkcji `map()` w następujący sposób:

```
>>> from operator import itemgetter
>>> start = itemgetter(0)
>>> end = itemgetter(1)
>>> dist = itemgetter(2)
>>> sm_trip = map(
...     lambda x: (start(x), end(x), dist(x) * 6076.12 / 5280),
...     trip
... )
```

Zdefiniowaliśmy wyrażenie lambda, które zostanie zastosowane do każdego odcinka podróży za pośrednictwem funkcji `map()`. Wyrażenie lambda używa innych wyrażeń lambda w celu wydzielenia z każdego odcinka wartości początku, końca i odległości. Wyrażenie oblicza skorygowaną odległość i na podstawie wartości początku, końca i odległości w milach lądowych tworzy nową krotkę odcinka.

Dokładnie takie działanie wykonuje następujące wyrażenie generatorowe:

```
>>> sm_trip = (
...     (start(x), end(x), dist(x) * 6076.12 / 5280)
...     for x in trip
... )
```

W wyrażeniu generatorowym wykonaliśmy identyczne przetwarzanie dla każdego elementu kolekcji.

Użycie wbudowanej funkcji `map()` lub wyrażenia generatorowego daje identyczne wyniki i charakteryzuje się prawie identyczną wydajnością. Wybór co do zastosowania wyrażeń lambda, nazwanych krotek, zdefiniowanych funkcji, funkcji `operator.itemgetter()` lub wyrażeń generatorowych zależy w całości od sposobu, w jaki chcemy doprowadzić do tego, aby tworzony przez nas program był zwięzły i czytelny.

Użycie funkcji map() w odniesieniu do wielu sekwencji

Czasami mamy dwie kolekcje danych, które muszą być do siebie równoległe. W rozdziale 4. „Praca z kolekcjami" widzieliśmy, że za pomocą funkcji `zip()` można przeplatać dwie sekwencje w celu utworzenia sekwencji par. W wielu przypadkach w istocie próbujemy zrobić coś takiego:

```
map(function, zip(jeden_obiekt_iterowalny, inny_obiekt_iterowalny))
```

Utworzyliśmy krotki argumentów na podstawie dwóch (lub większej liczby) równoległych obiektów iterowalnych i zastosowaliśmy funkcję do tych krotek argumentów. Może to być niezręczne, ponieważ parametry określonej funkcji `function()`, będą pojedynczą krotką dwuelementową. Oznacza to, że wartości argumentów nie zostaną zastosowane do każdego parametru.

W związku z tym w celu dekompozycji krotki na dwa pojedyncze parametry możemy zastosować następującą technikę:

```
(
    function(x, y)
    for x, y in zip(jeden_obiekt_iterowalny, inny_obiekt_iterowalny)
)
```

W tym przykładzie zastąpiliśmy funkcję `map()` równoważnym wyrażeniem generatorowym. Instrukcja `for x, y` dekomponuje dwuelementowe krotki, co pozwala zastosować poszczególne elementy krotki do każdego parametru funkcji.

Istnieje jednak lepsze podejście. Przyjrzyjmy się konkretnemu przykładowi podejścia alternatywnego.

W rozdziale 4. „Praca z kolekcjami" przyjrzeliśmy się danym dotyczącym podróży, które wyodrębniliśmy z pliku XML jako ciąg punktów pośrednich. Chcieliśmy stworzyć z tej listy punktów odcinki, które zawierają dane na temat początku i końca.

Poniżej znajduje się uproszczona wersja, w której użyto funkcji `zip()` do dwóch wycinków określonej sekwencji:

```
>>> waypoints = range(4)
>>> zip(waypoints, waypoints[1:])
<zip object at ...>

>>> list(zip(waypoints, waypoints[1:]))
[(0, 1), (1, 2), (2, 3)]
```

Stworzyliśmy sekwencję par, które wybraliśmy z pojedynczej, płaskiej listy. Każda para ma dwie sąsiednie wartości. Funkcja `zip()` prawidłowo zatrzymuje się, gdy krótsza

lista się wyczerpie. Ten wzorzec `zip(x, x [1:])` działa tylko dla sekwencji zmaterializowanych oraz obiektów iterowalnych utworzonych za pomocą funkcji `range()`. Wzorzec nie sprawdzi się dla obiektów iterowalnych, ponieważ operacja tworzenia wycinków nie została dla nich zaimplementowana.

Stworzyliśmy pary, żeby zastosować do każdej z nich funkcję `haversine()`. W ten sposób obliczymy odległość pomiędzy dwoma punktami na ścieżce. Oto jak wygląda jedna sekwencja kroków:

```
>>> from Chapter04.ch04_ex1 import (
...     floats_from_pair, float_lat_lon, row_iter_kml, haversine
... )
>>> import urllib.request

>>> data = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(data) as source:
...     path_gen = floats_from_pair(
...     float_lat_lon(row_iter_kml(source)))
...     path = list(path_gen)

>>> distances_1 = map(
...     lambda s_e: (s_e[0], s_e[1], haversine(*s_e)),
...     zip(path, path[1:])
... )
```

Stworzyliśmy listę punktów trasy i oznaczyliśmy ją za pomocą zmiennej `path`. Jest to uporządkowana sekwencja par szerokości i długości geograficznej. Ponieważ zamierzamy użyć wzorca projektowego `zip(path, path[1:])`, musimy mieć zmaterializowaną sekwencję, a nie prosty obiekt iterowalny.

Wynikami funkcji `zip()` są pary, które mają początek i koniec. Chcemy, aby wynik był trójką składającą się z początku, końca i odległości. Zastosowane wyrażenie lambda dekomponuje wejściową dwuelementową krotkę złożoną z danych na temat początku i końca i tworzy nową, trójelementową krotkę zawierającą początek, koniec i odległość.

Jak wspomniano wcześniej, możemy to uprościć, używając sprytnej własności funkcji `map()`, jak pokazano poniżej:

```
>>> distances_2 = map(
...     lambda s, e: (s, e, haversine(s, e)),
...     path, path[1:]
... )
```

Zauważmy, że dostarczyliśmy do funkcji `map()` funkcję i dwa obiekty iterowalne. Funkcja `map()` pobierze następny element z każdego obiektu iterowalnego i zastosuje te dwie wartości jako argumenty do podanej funkcji. W tym przypadku podana funkcja jest wyrażeniem lambda, które tworzy pożądaną trójelementową krotkę złożoną z początku, końca i odległości.

Formalna definicja funkcji `map()` mówi, że funkcja ta wykonuje przetwarzanie typu „mapa gwiazd" dla nieokreślonej liczby obiektów iterowalnych. Pobiera elementy z każdego obiektu iterowalnego, aby utworzyć krotkę wartości argumentów dla podanej funkcji. Oszczędza to nam konieczności dodawania funkcji `zip` w celu łączenia sekwencji.

Wykorzystanie funkcji filter() do przekazywania lub odrzucania danych

Zadaniem funkcji `filter()` jest użycie funkcji decyzyjnej zwanej predykatem do każdej wartości w kolekcji. Gdy funkcja predykatu zwraca `True`, wartość jest przekazywana; w przeciwnym razie jest odrzucana. Moduł `itertools` zawiera funkcję `filterfalse()`, która jest odmianą funkcji `filter()`. Użycie funkcji `filterfalse()` z modułu `itertools` zaprezentowano w rozdziale 8. „Moduł itertools".

Możemy zastosować tę funkcję do naszych danych podróży, aby utworzyć podzbiór odcinków o długości ponad 50 mil morskich:

```
>>> long_legs = list(
...     filter(lambda leg: dist(leg) >= 50, trip)
... )
```

Predykat lambda zwróci `True` dla długich odcinków, które zostaną zwrócone. Krótkie odcinki zostaną odrzucone. Wynik to 14 odcinków, które pomyślnie przeszły ten test odległości.

W przetwarzaniu tego rodzaju wyraźnie oddzielono zasadę filtrowania (`lambda leg: dist(leg) >= 50`) od innego przetwarzania, które tworzy obiekt `trip` lub analizuje długie odcinki.

W ramach kolejnego, prostego przykładu przyjrzyjmy się poniższemu fragmentowi kodu:

```
>>> filter(lambda x: x % 3 == 0 or x % 5 == 0, range(10))
<filter object at ...>
>>> sum(_)
23
```

Aby sprawdzić, czy liczba jest wielokrotnością trzech lub wielokrotnością pięciu, zdefiniowaliśmy proste wyrażenie lambda. Zastosowaliśmy tę funkcję do obiektu iterowalnego `range(10)`. Wynikiem jest iterowalna sekwencja liczb, które „przeszły" warunek reguły decyzyjnej.

Liczby, dla których wyrażenie lambda ma wartość `True`, to `[0, 3, 5, 6, 9]`, więc te wartości zostaną przekazane dalej. Ponieważ wyrażenie lambda ma wartość `False` dla wszystkich innych liczb, to zostaną one odrzucone.

Wskazówka

Zmienna _ jest specjalną cechą REPL Pythona. Domyślnie jest ustawiana na wynik wyrażenia. W poprzednim przykładzie wynik wyrażenia `filter(...)` został przypisany do zmiennej _. W następnym wierszu instrukcja `sum(_)` pobrała wartości z wyniku wyrażenia `filter(...)`.

Ta własność jest dostępna tylko w REPL i istnieje po to, aby zaoszczędzić nam trochę pisania podczas interaktywnej eksploracji złożonych funkcji.

Można to również zrobić za pomocą wyrażenia generatorowego, uruchamiając następujący kod:

```
>>> list(x for x in range(10) if x % 3 == 0 or x % 5 == 0)
[0, 3, 5, 6, 9]
```

Możemy to sformalizować za pomocą następującej notacji zbioru składanego (ang. *set comprehension*):

$$\{x \mid 0 \leq x < 10 \wedge (x \equiv 0\, mod\, 3 \vee x \equiv 0 mod\, 5)\}$$

Oznacza ona, że budujemy zbiór wartości x takich, że x należy do `range(10)` oraz `x% 3 == 0 lub x% 5 == 0`. Istnieje elegancka symetria pomiędzy funkcją filter() a formalnym, matematycznym rozumieniem zbioru składanego.

Często chcemy użyć funkcji `filter()` ze zdefiniowanymi funkcjami zamiast wyrażeń lambda. Oto przykład wielokrotnego użycia zdefiniowanego wcześniej predykatu:

```
>>> from Chapter02.ch02_ex1 import isprimeg

>>> list(filter(isprimeg, range(100)))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
↪73, 79, 83, 89, 97]
```

W tym przykładzie zaimportowaliśmy z innego modułu funkcję o nazwie `isprimeg()`. Następnie zastosowaliśmy tę funkcję do kolekcji wartości, aby przekazać liczby pierwsze i odrzucić z kolekcji wszystkie liczby, które nie są liczbami pierwszymi.

Może to być bardzo niewydajny sposób generowania tabeli liczb pierwszych. Powierzchowna prostota jest czymś, co prawnicy nazywają **atrakcyjną uciążliwością**. Wydaje się, że może być interesująca, ale się dobrze nie skaluje. Funkcja `isprimeg()` dubluje wszystkie „wysiłki testowe" dla każdej nowej wartości. Dla zapewnienia wielokrotnego wykorzystania testu sprawdzającego, czy liczby są pierwsze, potrzebujemy jakiegoś rodzaju pamięci podręcznej. Lepszym algorytmem jest sito Eratostenesa. Ten algorytm

zapamiętuje wcześniej znalezione liczby pierwsze i wykorzystuje je, aby zapobiec konieczności ponownego obliczania.

Więcej informacji na temat sprawdzania liczb pierwszych oraz tego algorytmu znajdowania niewielkich liczb pierwszych można znaleźć na stronie *https://primes.utm.edu/prove/prove2_1.html*.

Użycie funkcji filter() do identyfikacji wartości odstających

W poprzednim rozdziale zdefiniowaliśmy kilka przydatnych funkcji statystycznych do obliczenia średniej i odchylenia standardowego oraz normalizacji wartości. Możemy wykorzystać te funkcje do zlokalizowania wartości odstających z naszych danych dotyczących podróży. Do wartości odległości każdego odcinka podróży możemy zastosować funkcje `mean()` i `stdev()`, aby uzyskać średni rozkład i odchylenie standardowe.

Następnie możemy użyć funkcji `z()` w celu obliczenia znormalizowanej wartości dla każdego odcinka. Jeśli znormalizowana wartość jest większa niż 3, dane są bardzo odległe od średniej. Jeśli odrzucimy te odstające wartości, uzyskamy bardziej jednorodny zbiór danych, w którym istnieje mniejsze prawdopodobieństwo występowania błędów raportowania lub pomiaru.

Oto jak możemy sobie z tym poradzić:

```
>>> from Chapter04.ch04_ex3 import mean, stdev, z

>>> dist_data = list(map(dist, trip))
>>> μ_d = mean(dist_data)
>>> σ_d = stdev(dist_data)

>>> outlier = lambda leg: abs(z(dist(leg), μ_d, σ_d)) > 3

>>> list(filter(outlier, trip))
```

Dla każdego odcinka w kolekcji `trip` zmapowaliśmy funkcję odległości. `dist()` jest funkcją utworzoną przez wywołanie `itemgetter(2)`. Ponieważ chcemy wykonać kilka operacji z wynikiem, musimy zmaterializować obiekt listy. Nie możemy polegać na iteratorze, ponieważ pierwsza funkcja w tej sekwencji kroków wykorzysta wszystkie wartości iteratora. Wartości z tego wyodrębniania możemy następnie użyć do obliczenia statystyk populacji μ_d i σ_d zawierających średnie i odchylenie standardowe.

Na podstawie wartości średniej i odchylenia standardowego użyliśmy do filtrowania danych wyrażenia lambda `outlier`. Jeśli znormalizowana wartość jest zbyt duża, dane odstają. Próg „zbyt daleko od średniej” może się różnić w zależności od rodzaju

rozkładu. W przypadku rozkładu normalnego prawdopodobieństwo tego, że wartość mieści się w granicach trzech odchyleń standardowych od średniej, wynosi 0,997.

Wynikiem wyrażenia `list(filter(outlier, trip))` jest lista złożona z dwóch odcinków, które są dość długie w porównaniu z resztą odcinków w populacji. Średni odcinek wynosi około 34 mil morskich, przy standardowym odchyleniu 24 mil morskich.

Wskazówka

Dość złożony problem można rozłożyć na szereg niezależnych funkcji, z których każda może być łatwo przetestowana w izolacji. Nasze przetwarzanie to kompozycja prostszych funkcji. Takie podejście prowadzi do zwięzłego, ekspresywnego programowania funkcyjnego.

Funkcja iter() z wartością „strażnika”

Wbudowana funkcja `iter()` tworzy iterator po obiekcie klasy reprezentującej kolekcję. Funkcja `iter()` działa dla klas `list`, `dict` i `set`. Tworzy obiekt iteratora dla elementów w przetwarzanej kolekcji. W większości przypadków pozwalamy, aby niejawnie realizowała to instrukcja `for`. Jednak istnieją sytuacje, kiedy musimy jawnie utworzyć iterator. Jednym z przykładów może być oddzielenie głowy od ogona kolekcji.

Inne zastosowania funkcji `iter()` obejmują budowanie iteratorów w celu konsumowania wartości utworzonych przez obiekt wywoływalny (na przykład funkcję) aż do znalezienia wartości strażnika. Tę funkcję czasami wykorzystuje się razem z funkcją `read()` obiektu `file` w celu konsumowania elementów tak długo, dopóki nie zostanie znaleziona wartość znaku końca linii lub znaku końca pliku. Wyrażenie takie jak `iter(file.read, '\ n')` będzie oceniać wartość przekazanej funkcji, dopóki nie zostanie znaleziona wartość strażnika `'\ n'`. Z tej własności należy korzystać ostrożnie: jeśli strażnik nie zostanie znaleziony, konstrukcja może bez końca próbować czytać ciągi o zerowej długości.

Dostarczenie wywoływalnej funkcji do metody `iter()` może być dość trudne, ponieważ funkcja, którą dostarczamy, musi wewnętrznie utrzymywać pewien stan. W programowaniu funkcyjnym jest to na ogół niepożądane.

Ukryty stan jest jednak cechą otwartego pliku. Na przykład każde wywołanie funkcji `read()` lub `readline()` przesuwa wewnętrzny stan do następnego znaku lub następnego wiersza.

Innym przykładem jawnej iteracji jest sposób, w jaki metoda `pop()` mutowalnego obiektu kolekcji dokonuje stanowej zmiany w obiekcie kolekcji. Oto przykład użycia metody `pop()`:

```
>>> source = [1, 2, 3, None, 4, 5, 6]
>>> tail = iter(source.pop, None)
>>> list(tail)
[6, 5, 4]
```

Zmienną `tail` ustawiono na iterator po liście `[1, 2, 3, None, 4, 5, 6]`, która będzie przeglądana przez funkcję `pop()`. Domyślnym zachowaniem metody `pop()` jest `pop(-1)`, czyli elementy są pobierane w odwrotnej kolejności. To powoduje zmianę stanu obiektu listy: za każdym razem, gdy zostanie wywołana funkcja `pop()`, z listy zostanie usunięty element — co ją zmieni. Po znalezieniu wartości strażnika iterator zatrzymuje zwracanie wartości. Jeśli go nie znajdzie, następuje zgłoszenie wyjątku `IndexError`.

Tego rodzaju wewnętrzne zarządzanie stanem jest czymś, czego chcielibyśmy uniknąć. W związku z tym nie będziemy nalegali na korzystanie z tej funkcji.

Wykorzystanie funkcji sorted() do porządkowania danych

Kiedy trzeba generować wyniki w zdefiniowanym porządku, Python daje nam dwie możliwości. Możemy stworzyć obiekt listy i użyć metody `list.sort()` w celu zwrócenia elementów w odpowiednim porządku. Alternatywą jest użycie funkcji `sorted()`. Ta funkcja działa z każdym obiektem iterowalnym, ale w ramach operacji sortowania tworzy końcowy obiekt listy.

Z funkcji `sorted()` można korzystać na dwa sposoby. Można ją po prostu zastosować do kolekcji. Można jej także użyć jako funkcji wyższego rzędu za pomocą argumentu `key=`.

Załóżmy, że mamy dane podróży z przykładów w rozdziale 4. „Praca z kolekcjami". Mamy funkcję, która generuje sekwencję krotek zawierających dane początku, końca i odległości każdego odcinka podróży. Dane mają następującą postać:

```
[
((37.54901619777347, -76.33029518659048), (37.840832, -76.273834),
17.7246),
 ((37.840832, -76.273834), (38.331501, -76.459503), 30.7382),
 ((38.331501, -76.459503),  (38.845501, -76.537331), 31.0756),
 ((36.843334, -76.298668), (37.549, -76.331169), 42.3962),
 ((37.549, -76.331169), (38.330166, -76.458504), 47.2866),
 ((38.330166, -76.458504), (38.976334, -76.473503), 38.8019)
]
```

Domyślne zachowanie funkcji `sorted()` można zaobserwować w następującej interaktywnej sesji:

```
>>> sorted(dist(x) for x in trip)
[0.1731, 0.1898, 1.4235, 4.3155, ... 86.2095, 115.1751, 129.7748]
```

Użyliśmy wyrażenia generatorowego (`dist(x) for x in trip`), aby wyodrębnić odległości z danych podróży. `dist()` jest funkcją utworzoną przez wywołanie `itemgetter(2)`. Następnie posortowaliśmy tę iterowalną kolekcję liczb, aby uzyskać odległości od 0,17 mili morskiej do 129,77 mili morskiej.

Jeśli chcemy zachować odcinki i odległości w ich oryginalnych trójelementowych krotkach, możemy zastosować do funkcji `sorted()` funkcję `key()`, która określa sposób sortowania krotek, jak pokazano w poniższym fragmencie kodu:

```
>>> sorted(trip, key=dist)
[((35.505665, -76.653664), (35.508335, -76.654999), 0.1731), ...
```

Korzystając z funkcji `dist()`, aby wyodrębnić odległość z każdej krotki, posortowaliśmy dane podróży. Pokazana wcześniej funkcja `dist()` jest tworzona przez funkcję `itemgetter()` w następujący sposób:

```
>>> from operator import itemgetter
>>> dist = itemgetter(2)
```

Innym rozwiązaniem mogłoby być użycie wyrażenia lambda `leg: leg[2]`, aby wybrać z krotki określoną wartość. Podanie nazwy, `dist`, nieco lepiej wyjaśnia, który element jest wybierany z krotki.

Pisanie funkcji wyższego rzędu — przegląd

W tym podrozdziale przyjrzymy się zagadnieniu projektowania własnych funkcji wyższego rzędu. Zanim zagłębimy się w bardziej złożone rodzaje wzorców projektowych, podsumujemy kilka procesów. Zaczniemy od przyjrzenia się typowym przekształceniom danych, takim jak:

- opakowanie obiektów w celu tworzenia bardziej złożonych obiektów;
- rozpakowywanie złożonych obiektów na komponenty;
- spłaszczanie struktury;
- nadawanie struktury płaskiej sekwencji.

Powyższe wzorce pomagają zwizualizować sposoby projektowania funkcji wyższego rzędu w Pythonie.

Warto również przypomnieć, że definicja klasy `Callable` jest funkcją, która zwraca obiekt wywoływalny. Przyjrzymy się jej jako sposobowi pisania elastycznych funkcji, do których można wstrzykiwać parametry konfiguracyjne.

Dokładniejszy opis dekoratorów odłożymy do rozdziału 12. „Techniki projektowania dekoratorów". Dekorator również jest funkcją wyższego rzędu, ale wykorzystuje jedną funkcję i zwraca inną, co czyni go bardziej złożonym w porównaniu z przykładami zamieszczonymi w tym rozdziale. Zaczniemy od opracowania wysoce spersonalizowanych wersji funkcji `map()` i `filter()`.

Pisanie mapowań i filtrów wyższego rzędu

Dwie wbudowane funkcje wyższego rzędu w Pythonie — `map()` i `filter()` — ogólnie rzecz biorąc, obsługują prawie wszystkie obiekty, które do nich przekażemy. Trudno zoptymalizować je w ogólny sposób, aby osiągnąć wyższą wydajność. Podobnym funkcjom, takim jak `imap()`, przyjrzymy się w rozdziale 14. „Moduły Multiprocessing, Threading i Concurrent.Futures".

Istnieją trzy w dużym stopniu równoważne sposoby wyrażania mapowania. Załóżmy, że mamy jakąś funkcję `f(x)` i pewną kolekcję obiektów *C*. Oto sposoby obliczania mapowania z wartości dziedziny w C na wartość zakresu:

- Funkcja `map()`:

  ```
  map(f, C)
  ```

- Wyrażenie generatorowe:

  ```
  (f(x) for x in C)
  ```

- Funkcja generatorowa z instrukcją `yield`:

  ```
  from collections.abc import Callable, Iterable, Iterator
  from typing import Any

  def mymap(f: Callable[[Any], Any], C: Iterable[Any]) ->
  Iterator[Any]:
      for x in C:
          yield f(x)
  ```

Zaprezentowanej powyżej funkcji `mymap()` można użyć jako wyrażenia z funkcją do zastosowania oraz iterowalnym źródłem danych:

```
mymap(f, C)
```

Podobnie istnieją trzy sposoby zastosowania funkcji `filter` do kolekcji. Wszystkie są równoważne:

- Funkcja `filter()`:

  ```
  filter(f, C)
  ```

- Wyrażenie generatorowe:

  ```
  (x for x in C if f(x))
  ```

- Funkcja generatorowa z instrukcją `yield`:

  ```
  from collections.abc import Callable, Iterable, Iterator
  from typing import Any

  def myfilter(f: Callable[[Any], bool], C: Iterable[Any]) ->
  Iterator[Any]:
      for x in C:
          if f(x):
              yield x
  ```

  Zaprezentowanej powyżej funkcji `myfilter()` można użyć jako wyrażenia z funkcją do zastosowania oraz iterowalnym źródłem danych:

  ```
  myfilter(f, C)
  ```

Pomiędzy wymienionymi sposobami stosowania filtrowania występują pewne różnice w wydajności. Często zastosowanie funkcji `map()` i `filter()` jest rozwiązaniem najszybszym. Co ważniejsze, istnieją różne rodzaje rozszerzeń, które pasują do tych projektów mapowania i filtrowania. Oto one:

- Aby zmodyfikować sposób przetwarzania, możemy stworzyć bardziej zaawansowaną funkcję `g(x)`, która będzie zastosowana do każdego elementu, lub możemy zastosować funkcję do całej kolekcji przed przetwarzaniem. Jest to najbardziej ogólne podejście i dotyczy wszystkich trzech projektów. W tym rozwiązaniu zainwestujemy większość naszej energii projektowej w programowanie funkcyjne. Możemy zdefiniować nową funkcję na bazie istniejącej funkcji `f(x)` lub możemy podjąć decyzję o refaktoryzacji pierwotnej funkcji. We wszystkich przypadkach wysiłek projektowy włożony w programowanie funkcyjne wydaje się przynosić największe korzyści.
- Możemy dostosować pętlę `for` wewnątrz wyrażenia generatorowego lub funkcji generatorowej. Jedną z oczywistych poprawek jest połączenie mapowania i filtrowania w jedną operację poprzez rozszerzenie wyrażenia generatorowego za pomocą klauzuli `if`. Możemy również połączyć funkcje `mymap()` i `myfilter()`, aby połączyć mapowanie z filtrowaniem. Wymaga to pewnej ostrożności, aby wynikowa funkcja nie była zlepkiem różnych własności.

Gdy oprogramowanie ewoluuje i dojrzewa, na ogół zachodzą w nim głębokie zmiany, które często przekształcają strukturę danych przetwarzanych przez pętlę. Istnieje wiele wzorców projektowych, w tym opakowywanie, rozpakowywanie (lub wyodrębnianie), spłaszczanie i nadawanie struktury. Kilku z tych technik przyjrzeliśmy się w poprzednich rozdziałach.

W kolejnych punktach zaprezentuję sposoby projektowania własnych funkcji wyższego rzędu.

Zaczniemy od rozpakowania złożonych danych z jednoczesnym zastosowaniem funkcji mapowania. W każdym przykładzie warto przyjrzeć się miejscom, w których powstaje złożoność, i zdecydować, czy wynikowy kod jest naprawdę zwięzły i wyrazisty.

Rozpakowywanie danych podczas mapowania

Gdy używamy takiej konstrukcji jak `(f(x) for x, y in C)`, posługujemy się wieloma podstawieniami w instrukcji for, tak aby rozpakować wielowartościową krotkę, a następnie zastosować funkcję do jej elementów. Całe wyrażenie jest mapowaniem. Jest to typowa optymalizacja Pythona wykonywana w celu zmiany struktury i zastosowania funkcji.

Do zaprezentowania tego mechanizmu wykorzystamy dane dotyczące podróży z rozdziału 4. „Praca z kolekcjami". Oto konkretny przykład rozpakowywania podczas mapowania:

```
from collections.abc import Callable, Iterable, Iterator
from typing import Any, TypeAlias

Conv_F: TypeAlias = Callable[[float], float]
Leg: TypeAlias = tuple[Any, Any, float]

def convert(
    conversion: Conv_F,
    trip: Iterable[Leg]) -> Iterator[float]:
    return (
        conversion(distance)
        for start, end, distance in trip
    )
```

Ta funkcja wyższego rzędu będzie obsługiwana przez funkcje konwersji, które można zastosować do surowych danych w następujący sposób:

```
from collections.abc import Callable
from typing import TypeAlias

Conversion: TypeAlias = Callable[[float], float]
```

```
to_miles: Conversion = lambda nm: nm * 6076.12 / 5280

to_km: Conversion = lambda nm: nm * 1.852

to_nm: Conversion = lambda nm: nm
```

Funkcje konwersji zostały zdefiniowane jako wyrażenia lambda i przypisane do zmiennych. Niektóre narzędzia do analizy statycznej wygenerują z tego powodu ostrzeżenia, ponieważ zgodnie ze standardem PEP-8 takie operacje nie są zalecane.

Oto przykład wyodrębnienia odległości i zastosowania do niej funkcji konwersji:

```
>>> convert(to_miles, trip)
<generator object ...>
>>> miles = list(convert(to_miles, trip))
>>> trip[0]
((37.54901619777347, -76.33029518659048), (37.840832, -76.273834),
17.7246)
>>> miles[0]
20.397120559090908
>>> trip[-1]
((38.330166, -76.458504), (38.976334, -76.473503), 38.8019)
>>> miles[-1]
44.652462240151515
```

Podczas rozpakowywania wynik będzie sekwencją wartości zmiennoprzecinkowych. Oto uzyskane wyniki:

```
[20.397120559090908, 35.37291511060606, ..., 44.652462240151515]
```

Powyższa funkcja `convert()` jest specyficzna dla struktury danych opisującej podróż: początek-koniec-odległość, ponieważ ta trójelementowa krotka jest dekomponowana w pętli `for`.

Podczas mapowania wzorca projektowego możemy stworzyć bardziej ogólne rozwiązanie dla tego rodzaju rozpakowywania. Jest ono nieco bardziej złożone. Po pierwsze potrzebujemy ogólnej funkcji dekompozycji podobnej do pokazanej w poniższym fragmencie kodu:

```
from collections.abc import Callable
from operator import itemgetter
from typing import TypeAlias

Selector: TypeAlias = Callable[[tuple[Any, ...]], Any]

fst: Selector = itemgetter(0)

snd: Selector = itemgetter(1)

sel2: Selector = itemgetter(2)
```

Chcielibyśmy móc wyrazić `f(sel2(s_e_d)) for s_e_d in trip`. To wymaga kompozycji funkcyjnej: łączymy funkcję, na przykład `to_miles()`, z selektorem, na przykład `sel2()`.

Bardziej opisowe nazwy często są bardziej przydatne od nazw generycznych. Zadanie zmiany nazwy pozostawiam jako ćwiczenie dla czytelnika do samodzielnego wykonania. Kompozycję funkcyjną w Pythonie możemy wyrazić, używając dodatkowego wyrażenia lambda, na przykład:

```
from collections.abc import Callable

to_miles_sel2: Callable[[tuple[Any, Any, float]], float] = (
    lambda s_e_d: to_miles(sel2(s_e_d))
)
```

To doprowadza nas do nieco dłuższej, ale bardziej ogólnej wersji rozpakowywania:

```
>>> miles2 = list(
...     to_miles_sel2(s_e_d) for s_e_d in trip
... )
```

Z tym wyrażeniem generatorowym możemy porównać funkcję wyższego rzędu `convert()`. Obie konstrukcje wykonują szereg przekształceń. Funkcja `convert()` „ukrywa" szczegóły przetwarzania — kompozycję krotki, czyli początek, koniec i odległość — za pomocą instrukcji `for`, która dekomponuje krotkę. Wyrażenie generatorowe ujawnia tę dekompozycję poprzez włączenie do definicji funkcji złożonej funkcji `sel2()`.

Żadne z tych rozwiązań nie jest pod jakimś względem „lepsze" od drugiego. Reprezentują one dwa podejścia do eksponowania lub ukrywania szczegółów. W kontekście tworzenia konkretnej aplikacji bardziej pożądane może być ujawnienie (lub ukrycie) szczegółów.

Ta sama zasada projektowa sprawdza się podczas tworzenia filtrów hybrydowych zamiast mapowania. W klauzuli `if` zwróconego wyrażenia generatorowego zastosujemy filtr.

Aby tworzyć jeszcze bardziej złożone funkcje, możemy połączyć mapowanie z filtrowaniem. Chociaż jest to atrakcyjne podczas tworzenia bardziej złożonych funkcji, nie zawsze jest wartościowe. Zastosowanie funkcji złożonej nie zawsze dorównuje wydajnością zagnieżdżonemu użyciu prostych funkcji `map()` i `filter()`. Ogólnie rzecz biorąc, chcemy tworzyć bardziej złożoną funkcję, jeśli implementuje koncepcję, dzięki której zrozumienie oprogramowania staje się łatwiejsze.

Opakowywanie dodatkowych danych podczas mapowania

Kiedy korzystamy z takiej konstrukcji jak `((f(x), x) for x in C)`, używamy opakowania do utworzenia wielowartościowej krotki, a jednocześnie stosujemy mapowanie. Jest to powszechnie stosowana technika zachowywania pochodnych wyników poprzez tworzenie większych konstrukcji. Ma to tę zaletę, że pozwala uniknąć konieczności ponownego obliczania, a jednocześnie nie wymaga utrzymywania złożonych obiektów z wewnętrzną zmianą stanu. W tym przypadku zmiana stanu jest strukturalna i bardzo widoczna.

Oto część przykładu pokazanego w rozdziale 4. „Praca z kolekcjami" wykorzystanego do utworzenia danych podróży na podstawie ścieżki punktów. Kod wygląda następująco:

```
>>> from Chapter04.ch04_ex1 import (
...     floats_from_pair, float_lat_lon, row_iter_kml, haversine, legs
... )
>>> import urllib.request
>>> data = "file:./Winter%202012-2013.kml"

>>> with urllib.request.urlopen(data) as source:
    ...     path = floats_from_pair(float_lat_lon(row_iter_kml(source)))
...     trip = tuple(
...     (start, end, round(haversine(start, end), 4))
...     for start, end in legs(path)
...     )
```

Możemy nieco zmodyfikować ten kod, aby utworzyć funkcję wyższego rzędu, która oddziela opakowanie od innych funkcji. Refaktoryzacja mogłaby polegać na utworzeniu funkcji, która konstruuje nową krotkę obejmującą krotkę wejściową i odległość. Funkcję możemy zdefiniować w następujący sposób:

```
from collections.abc import Callable, Iterable, Iterator
from typing import TypeAlias

Point: TypeAlias = tuple[float, float]
Leg_Raw: TypeAlias = tuple[Point, Point]
Point_Func: TypeAlias = Callable[[Point, Point], float]
Leg_D: TypeAlias = tuple[Point, Point, float]

def cons_distance(
        distance: Point_Func,
        legs_iter: Iterable[Leg_Raw]) -> Iterator[Leg_D]:
    return (
        (start, end, round(distance(start,end), 4))
        for start, end in legs_iter
    )
```

Ta funkcja dekomponuje każdy odcinek na dwie zmienne: `start` i `end`. Te zmienne są egzemplarzami klasy `Point`, zdefiniowanymi jako krotki złożone z dwóch wartości zmiennoprzecinkowych. Będą użyte z określoną funkcją `distance()` do obliczania odległości pomiędzy punktami. Funkcja jest obiektem wywoływalnym, który pobiera dwa obiekty `Point` i zwraca zmiennoprzecinkowy wynik. Wynik zbuduje trójelementową krotkę, która zawiera dwa wejściowe obiekty `Point` oraz obliczony wynik w formacie zmiennoprzecinkowym.

Następnie możemy przepisać nasz kod w celu zastosowania funkcji `haversine()` obliczającej odległość:

```
>>> source_url = "file:./Winter%202012-2013.kml"
>>> with urllib.request.urlopen(source_url) as source:
...     path = floats_from_pair(
...         float_lat_lon(row_iter_kml(source))
...     )
...     trip2 = tuple(
...         cons_distance(haversine, legs(iter(path)))
...     )
```

Wyrażenie generatorowe zastąpiliśmy funkcją wyższego rzędu `cons_distance()`. Funkcja nie tylko przyjmuje funkcję jako argument, ale także zwraca wyrażenie generatorowe. W niektórych aplikacjach ten bardziej rozbudowany i bardziej złożony etap przetwarzania jest pomocnym sposobem na pominięcie niepotrzebnych szczegółów.

W rozdziale 10. „Moduł functools" pokażemy, jak użyć funkcji `partial()` w celu ustawienia wartości parametru `R` funkcji `haversine()`, zmieniającej jednostki, w jakich obliczamy odległość.

Spłaszczanie danych podczas mapowania

W rozdziale 4. „Praca z kolekcjami" przyjrzeliśmy się algorytmom, które spłaszczały zagnieżdżoną strukturę krotki złożonej z krotek w jeden obiekt iterowalny. Naszym celem w tym czasie była po prostu restrukturyzacja niektórych danych, bez żadnego rzeczywistego przetwarzania. Możemy jednak tworzyć rozwiązania hybrydowe, które łączą funkcję z operacją spłaszczania.

Załóżmy, że mamy blok tekstu, który chcemy przekonwertować na płaską sekwencję liczb. Tekst ma następującą postać:

```
>>> text = """2 3 5 7 11 13 17 19 23 29
... 31 37 41 43 47 53 59 61 67 71
... 73 79 83 89 97 101 103 107 109 113
... 127 131 137 139 149 151 157 163 167 173
... 179 181 191 193 197 199 211 223 227 229
... """
```

Każdy wiersz to blok 10 liczb. Chcemy rozpakować wiersze, aby utworzyć płaską sekwencję liczb.

Można to zrobić za pomocą dwuczęściowej funkcji generatorowej w następującej postaci:

```
>>> data = list(
...     v
...     for line in text.splitlines()
...         for v in line.split()
... )
```

Ten kod rozdziela tekst na wiersze i iteruje po tych wierszach. Każdy wiersz jest natomiast podzielony na iterowalne słowa. Wynik tego kodu jest listą ciągów znaków o następującej postaci:

```
['2', '3', '5', '7', '11', '13', '17', '19', '23', '29', '31', '37',
'41', '43', '47', '53', '59', '61', '67', '71', '73', '79', '83',
'89', '97', '101', '103', '107', '109', '113', '127', '131', '137',
'139', '149', '151', '157', '163', '167', '173', '179', '181', '191',
'193', '197', '199', '211', '223', '227', '229']
```

Powyższy kod można zoptymalizować, a optymalizacja dotyczy tekstu w tym konkretnym formacie. Zadanie to pozostawiam jako ćwiczenie dla czytelnika do samodzielnego wykonania.

W celu przekonwertowania ciągów znaków na liczby trzeba zastosować funkcję konwersji, a także rozwinąć wejściowy format o strukturze bloków, używając następującego fragmentu kodu:

```
from collections.abc import Callable, Iterator
from typing import TypeAlias

Num_Conv: TypeAlias = Callable[[str], float]

def numbers_from_rows(
        conversion: Num_Conv,
        text: str) -> Iterator[float]:
    return (
        conversion(value)
        for line in text.splitlines()
        for value in line.split()
    )
```

Ta funkcja zawiera argument `conversion`, który jest funkcją stosowaną do każdej generowanej wartości. Wartości są tworzone przez spłaszczanie za pomocą algorytmu zaprezentowanego wcześniej.

Możemy użyć funkcji `numbers_from_rows()` w wyrażeniu następującego typu:

```
>>> list(numbers_from_rows(float, text))
```

W tym przypadku w celu utworzenia listy wartości zmiennoprzecinkowych na podstawie bloku tekstu użyliśmy wbudowanej funkcji `float()`.

Do dyspozycji mamy wiele alternatyw wykorzystujących mieszankę funkcji wyższego rzędu z wyrażeniami generatorowymi. Na przykład możemy skorzystać z wyrażenia w następującej postaci:

```
>>> text = (value
...    for line in text.splitlines()
...    for value in line.split()
... )
>>> numbers = map(float, text)
>>> list(numbers)
```

To może pomóc nam w zrozumieniu ogólnej struktury algorytmu. Stosowana reguła, tzw. **chunking** (dosł. kawałkowanie), polega na podsumowaniu szczegółów funkcji o opisowej nazwie. Dzięki temu podsumowaniu szczegóły są abstrakcyjne. Możemy więc pracować z funkcją jako składową w szerszym kontekście. Podczas gdy często korzystamy z funkcji wyższego rzędu, czasami czytelniejsze jest posługiwanie się wyrażeniami generatorowymi.

Strukturyzacja danych podczas filtrowania

W poprzednich trzech przykładach połączyliśmy dodatkowe przetwarzanie z mapowaniem. Łączenie przetwarzania z filtrowaniem nie wydaje się być tak ekspresywne jak łączenie z mapowaniem. Poniżej zaprezentujemy szczegółowy przykład, aby pokazać, że chociaż jest przydatny, nie wydaje się, żeby był tak atrakcyjny jak łączenie mapowania z przetwarzaniem.

W rozdziale 4. „Praca z kolekcjami" przyjrzeliśmy się algorytmom strukturyzacji. Możemy z łatwością połączyć filtrowanie z algorytmem strukturyzacji w jedną, złożoną funkcję. Poniżej zamieszczono wersję preferowanej funkcji do grupowania wyjścia obiektu iterowalnego:

```
from collections.abc import Iterator
from typing import TypeVar
ItemT = TypeVar("ItemT")

def group_by_iter(
        n: int,
        iterable: Iterator[ItemT]
) -> Iterator[tuple[ItemT, ...]]:
    def group(n: int, iterable: Iterator[ItemT]) -> Iterator[ItemT]:
        for i in range(n):
            try:
                yield next(iterable)
            except StopIteration:
```

```
                return

    while row := tuple(group(n, iterable)):
        yield row
```

Powyższy kod próbuje utworzyć *n*-elementową krotkę pobraną z obiektu iterowalnego. Jeśli w krotce są jakieś elementy, są one przekazywane jako część wynikowego obiektu iterowalnego. Zasadniczo funkcja działa rekurencyjnie na pozostałych elementach z wejściowego obiektu iterowalnego. Ponieważ rekurencja w Pythonie ma pewne ograniczenia, zoptymalizowaliśmy strukturę wywołania ogonowego, przekształcając ją w jawną instrukcję `while`.

Wynikiem działania funkcji `group_by_iter()` jest sekwencja *n*-elementowych krotek. W poniższym przykładzie za pomocą funkcji filtrującej utworzymy sekwencję liczb, a następnie pogrupujemy je w siedmioelementowych krotkach:

```
>>> from pprint import pprint
>>> data = list(
...     filter(lambda x: x % 3 == 0 or x % 5 == 0, range(1, 50))
... )
>>> data
[3, 5, 6, 9, 10, ..., 48]
>>> grouped = list(group_by_iter(7, iter(data)))
>>> pprint(grouped)
[(3, 5, 6, 9, 10, 12, 15),
 (18, 20, 21, 24, 25, 27, 30),
 (33, 35, 36, 39, 40, 42, 45),
 (48,)]
```

Możemy połączyć grupowanie i filtrowanie w jedną funkcję, która wykona obie te operacje w jednym ciele funkcji. Modyfikacja funkcji `group_by_iter()` wygląda następująco:

```
from collections.abc import Callable, Iterator, Iterable
from typing import Any, TypeAlias

ItemFilterPredicate: TypeAlias = Callable[[Any], bool]

def group_filter_iter(
        n: int,
        predicate: ItemFilterPredicate, items: Iterator[ItemT]
) -> Iterator[tuple[ItemT, ...]]:
    def group(n: int, iterable: Iterator[ItemT]) -> Iterator[ItemT]:
        for i in range(n):
            try:
                yield next(iterable)
            except StopIteration:
                return
        subset = filter(predicate, items)
```

```
    # ^-- Dodano w celu zastosowania filtra
    while row := tuple(group(n, subset)):
                                # ^-- Zmieniono w celu wykorzystania filtra
        yield row
```

Do funkcji `group_by_iter()` dodaliśmy jeden wiersz. To zastosowanie funkcji `filter()` tworzy podzbiór wyników. Zamieniliśmy wiersz `while := tuple(group(n, subset)):` w celu wykorzystania podzbioru elementów zamiast całej wejściowej kolekcji.

Powyższa funkcja `group_filter_iter()` stosuje funkcję predykatu filtra do źródłowego obiektu iterowalnego podanego za pomocą parametru `items`. Ponieważ wyjście filtra jest samo w sobie nieścisłym obiektem iterowalnym, wartość `subset` nie jest obliczana z góry; wartości są tworzone w razie potrzeby. Większa część tej funkcji jest identyczna z wersją pokazaną wcześniej.

Możemy nieco uprościć kontekst, w którym korzystamy z tej funkcji: możemy porównać jawne użycie funkcji `filter()` z powyższą łączoną funkcją, w której funkcja `filter()` jest wywoływana niejawnie. Porównanie zaprezentowano na poniższym przykładzie:

```
>>> rule: ItemFilterPredicate = lambda x: x % 3 == 0 or x % 5 == 0
>>> groups_explicit = list(
...     group_by_iter(7, filter(rule, range(1, 50)))
... )
>>> groups = list(
...     group_filter_iter(7, rule, iter(range(1, 50)))
... )
```

W powyższym kodzie zastosowaliśmy predykat filtru i pogrupowaliśmy wyniki w jedno wywołanie funkcji. W przypadku funkcji `filter()` stosowanie filtra w połączeniu z innym przetwarzaniem rzadko przynosi oczywiste korzyści. Wydaje się, że odrębna, widoczna funkcja `filter()` jest bardziej przydatna od funkcji łączonej.

Budowanie funkcji wyższego rzędu z wykorzystaniem obiektów wywoływalnych

Funkcje wyższego rzędu można zdefiniować jako klasy wywoływalne. Koncepcja ta bazuje na idei pisania funkcji generatorowych. Piszemy obiekty wywoływalne, ponieważ potrzebujemy stanowych cech Pythona, takich jak zmienne egzemplarza. Podczas tworzenia funkcji wyższego rzędu oprócz używania instrukcji możemy również zastosować statyczną konfigurację. W szczególności do zmieniania własności obiektów wywoływalnych bardzo dobrze nadaje się wzorzec projektowy **Strategia**.

W definicji klasy wywoływalnej istotne jest to, że obiekt `class` utworzony przez instrukcję `class` definiuje funkcję, która generuje inną funkcję. Zwykle używamy obiektu wywoływalnego do stworzenia funkcji złożonej, która łączy dwie inne funkcje w stosunkowo złożoną konstrukcję.

Aby się o tym przekonać, weźmy pod uwagę następującą klasę:

```
from collections.abc import Callable
from typing import Any

class NullAware:
    def __init__(self, some_func: Callable[[Any], Any]) -> None:
        self.some_func = some_func

    def __call__(self, arg: Any) -> Any:
        return None if arg is None else self.some_func(arg)
```

Powyższa klasa służy do tworzenia nowej funkcji, która obsługuje wartości puste. Podczas tworzenia egzemplarza tej klasy wywoływana jest funkcja `some_func`. Jedynym ograniczeniem jest to, aby funkcja `some_func` spełniała warunek `Callable[[Any], Any]`. Oznacza to, że funkcja będąca argumentem przyjmuje jeden argument i zwraca pojedynczy wynik. Wynikowy obiekt jest wywoływalny. Oczekiwany jest pojedynczy, opcjonalny argument. Implementacja metody `__call__()` uwzględnia użycie jako argumentów obiektów `None`. Ta metoda powoduje, że wynikowy obiekt spełnia warunek `Callable[[Optional[Any]], Optional[Any]]`.

Na przykład w wyniku oceny wyrażenia `NullAware(math.log)` zostanie utworzona nowa funkcja, którą można zastosować do wartości argumentów. Metoda `__init__()` zapisze przekazaną funkcję w obiekcie wynikowym. Ten obiekt jest funkcją, którą można następnie wykorzystać do przetwarzania danych.

Powszechnym podejściem jest stworzenie nowej funkcji i zapisanie jej w celu przyszłego wykorzystania poprzez przypisanie jej nazwy w następujący sposób:

```
import math

null_log_scale = NullAware(math.log)

null_round_4 = NullAware(lambda x: round(x, 4))
```

Pierwsza instrukcja za instrukcją `import` tworzy nową funkcję i przypisuje jej nazwę `null_log_scale()`. Druga instrukcja tworzy obsługującą wartości puste funkcję `null_round_4`. Funkcja używa jako wewnętrznej wartości funkcji obiektu lambda, który będzie zastosowany, jeśli parametr nie jest równy `None`. Następnie możemy użyć tej funkcji w innym kontekście. Przyjrzyjmy się poniższemu przykładowi:

```
>>> some_data = [10, 100, None, 50, 60]
>>> scaled = map(null_log_scale, some_data)
```

```
>>> [null_round_4(v) for v in scaled]
[2.3026, 4.6052, None, 3.912, 4.0943]
```

Metoda `__call__()` z tego przykładu bazuje w całości na ocenie wartości wyrażenia. To elegancki i schludny sposób definiowania złożonych funkcji składających się z bardziej niskopoziomowych funkcji składowych.

Zapewnienie dobrego projektu funkcyjnego

Idea bezstanowego programowania funkcyjnego wymaga pewnej ostrożności przy korzystaniu z obiektów Pythona. Obiekty zazwyczaj są stanowe. W rzeczywistości można się spierać, że celem programowania obiektowego jest enkapsulacja zmian stanu wewnątrz definicji klasy. Z tego powodu, gdy używamy definicji klas Pythona do przetwarzania kolekcji, możemy odnieść wrażenie, że zasady programowania funkcyjnego i programowania imperatywnego przeciągają nas w przeciwnych kierunkach.

Zaletą używania obiektów wywoływalnych do tworzenia funkcji złożonej jest nieco prostsza składnia wykorzystania wynikowej, złożonej funkcji. Kiedy zaczynamy pracować z iterowalnymi mapowaniami lub redukcjami, musimy pamiętać o tym, w jaki sposób i w jakim celu wprowadzamy obiekty stanowe.

Rozważmy dość złożoną funkcję, która ma następujące własności:

- Stosuje filtr do źródła elementów.
- Stosuje mapowanie do elementów, które przechodzą przez filtr.
- Oblicza sumę mapowanych wartości.

Moglibyśmy spróbować zdefiniować tę funkcję jako prostą funkcję wyższego rzędu, ale przy trzech oddzielnych wartościach parametrów jej użycie byłoby kłopotliwe. Zamiast tego utworzymy obiekt wywoływalny konfigurowany przez funkcje filtrowania i mapowania.

Używanie obiektów do konfigurowania obiektu to wykorzystywany w programowaniu obiektowym wzorzec projektowy **Strategia**. Oto definicja klasy, która w celu utworzenia obiektu wywoływalnego wymaga funkcji filtrowania i mapowania:

```
from collections.abc import Callable, Iterable

class Sum_Filter:
    __slots__ = ["filter", "function"]

    def __init__(self,
            filter: Callable[[float], bool],
            func: Callable[[float], float]) -> None:
        self.filter = filter
        self.function = func
```

```
    def __call__(self, iterable: Iterable[float]) -> float:
        return sum(
            self.function(x)
            for x in iterable
            if self.filter(x)
        )
```

Ta klasa ma w każdym obiekcie dokładnie dwa atrybuty. To nakłada kilka ograniczeń na zdolność do używania funkcji jako obiektu stanowego. Nie zapobiega to wszystkim modyfikacjom obiektu wynikowego, ale ogranicza nas do zaledwie dwóch atrybutów. Próba dodania atrybutów powoduje wyjątek.

Metoda inicjująca `__init__()` przechowuje nazwy zmiennych: `filter` i `func` w zmiennych egzemplarza obiektu. Metoda `__call__()` zwraca wartość na podstawie wyrażenia generatorowego, które używa dwóch wewnętrznych definicji funkcji. Funkcja `self.filter()` służy do przekazywania lub odrzucania elementów. Funkcja `self.function()` służy do transformacji obiektów przekazywanych przez funkcję `filter()`.

Egzemplarz tej klasy jest funkcją, która ma wbudowane dwie funkcje strategii. Egzemplarz tworzymy w następujący sposób:

```
count_not_none = Sum_Filter(
   lambda x: x is not None,
   lambda x: 1
)
```

Zbudowaliśmy funkcję o nazwie `count_not_none()`, która zlicza w sekwencji wartości inne niż `None`. Robi to, używając wyrażenia lambda do przekazywania wartości innych niż `None` i funkcji, która używa stałej 1 zamiast rzeczywistych wartości.

Ogólnie rzecz biorąc, ten obiekt `count_not_none()` będzie zachowywał się jak dowolna inna funkcja Pythona. Funkcję `count_not_None()` możemy wykorzystać w następujący sposób:

```
>>> some_data = [10, 100, None, 50, 60]
>>> count_not_none(some_data)
4
```

Na tym przykładzie pokazałem technikę wykorzystania niektórych własności programowania obiektowego Pythona do tworzenia obiektów wywoływalnych wykorzystywanych w funkcyjnym podejściu do projektowania i budowania oprogramowania. Możemy oddelegować pewną złożoność do zaawansowanej funkcji. Istnienie jednej funkcji z wieloma własnościami upraszcza zrozumienie kontekstu, w którym jest używana funkcja.

Przegląd wybranych wzorców projektowych

Funkcje `max()`, `min()` i `sorted()` mają domyślne zachowanie bez funkcji `key=`. Można je spersonalizować, dostarczając funkcji, która definiuje sposób obliczania klucza na podstawie dostępnych danych. W wielu naszych przykładach funkcja `key=` wykonywała proste wyodrębnianie dostępnych danych. To nie jest wymagane. Funkcja `key=` może robić cokolwiek.

Wyobraźmy sobie następującą metodę: `max(trip, key=random.randint())`. Ogólnie rzecz biorąc, staramy się nie korzystać z funkcji `key=`, które robią coś tak niejasnego.

Użycie funkcji `key=` jest powszechnie stosowanym wzorcem projektowym. Możemy bez trudu zapewnić projektowanym funkcjom zgodność z tym wzorcem.

Przyjrzeliśmy się również, w jaki sposób stosowanie funkcji wyższego rzędu może uprościć format lambda. Istotną zaletą korzystania z formatu wyrażeń lambda jest bardzo ścisłe przestrzeganie paradygmatu funkcyjnego. Podczas pisania bardziej konwencjonalnych funkcji możemy tworzyć programy imperatywne, które mogą zaśmiecać zwięzły i ekspresyjny projekt funkcyjny.

Przyjrzeliśmy się kilku rodzajom funkcji wyższego rzędu, które działają z kolekcjami wartości. W poprzednich rozdziałach wspominaliśmy o kilku różnych wzorcach projektowych dla funkcji wyższego rzędu, które mają zastosowanie do obiektów kolekcji i obiektów skalarnych. Poniżej znajduje się ich ogólna klasyfikacja:

- **Zwracanie generatora**. Funkcja wyższego rzędu może zwrócić wyrażenie generatorowe. Uważamy funkcję za wyższego rzędu, ponieważ nie zwraca wartości skalarnych lub kolekcji wartości. `Niektóre z takich funkcji wyższego rzędu akceptują również funkcje jako argumenty.
- **Działanie jako generator**. W niektórych przykładach funkcji wykorzystaliśmy instrukcję `yield`, aby przekształcić je w pełnoprawne funkcje generatorowe. Wartość funkcji generatorowej jest iterowalną kolekcją ocenianych leniwie wartości. Sugerujemy, że funkcja generatorowa jest zasadniczo nieodróżnialna od funkcji, która zwraca wyrażenie generatorowe. Obie są wartościowane leniwie. Obie mogą emitować sekwencję wartości. Z tego powodu funkcje generatorowe także uznajemy za funkcje wyższego rzędu. Do tej kategorii należą funkcje wbudowane, takie jak `map()` i `filter()`.

- **Materializacja kolekcji**. Niektóre funkcje muszą zwracać obiekt zmaterializowanej kolekcji: listę, krotkę, zbiór lub mapowanie. Tego rodzaju funkcje mogą być funkcjami wyższego rzędu, jeśli wykorzystują funkcję jako część argumentów. W przeciwnym razie są to zwykłe funkcje, które przetwarzają kolekcje.
- **Redukowanie kolekcji**. Niektóre funkcje działają z obiektem iterowalnym i tworzą skalarny wynik. Przykładem mogą być funkcje `len()` i `sum()`. Gdy przyjmujemy funkcję jako argument, możemy tworzyć redukcje wyższego rzędu. Do tego tematu powrócimy w następnym rozdziale.
- **Wartości skalarne**. Niektóre funkcje działają na pojedynczych elementach danych. Jeśli przyjmują inną funkcję jako argument, mogą to być funkcje wyższego rzędu.

Podczas projektowania oprogramowania możemy wybierać spośród tych ustalonych wzorców projektowych.

Podsumowanie

W tym rozdziale przyjrzeliśmy się dwóm redukcjom, które są funkcjami wyższego rzędu: `max()` i `min()`. Przyjrzeliśmy się także dwóm centralnym funkcjom wyższego rzędu, `map()` i `filter()`. Omówiliśmy także funkcję `sorted()`.

Przyjrzeliśmy się także sposobom wykorzystania funkcji wyższego rzędu do przekształcania struktury danych. Możemy wykonać kilka typowych transformacji, z opakowywaniem, rozpakowywaniem, spłaszczaniem i nadawaniem struktury różnego typu sekwencji włącznie.

Przyjrzeliśmy się trzem sposobom definiowania własnych funkcji wyższego rzędu. Oto one:

- Instrukcja `def`. Podobna do wyrażenia lambda, które przypisujemy do zmiennej.
- Definiowanie klasy wywoływalnej jako rodzaju funkcji emitującej funkcje złożone.

Do tworzenia funkcji złożonych możemy również używać dekoratorów. Powrócimy do tego tematu w rozdziale 12. „Techniki projektowania dekoratorów”.

W następnym rozdziale przyjrzymy się idei czysto funkcyjnej iteracji z wykorzystaniem rekurencji. Użyjemy struktur Pythona do wprowadzenia kilku popularnych ulepszeń w stosunku do technik czysto funkcyjnych. Przyjrzymy się również związanemu z tym problemowi wykonywania redukcji ze zbiorów do pojedynczych wartości.

Ćwiczenia

Ćwiczenia zamieszczone w tej książce są oparte na kodzie udostępnionym przez wydawnictwo Packt Publishing w serwisie GitHub. Można go pobrać na stronie *https://github.com/PacktPublishing/Functional-Python-Programming-3rd-Edition*.

W niektórych przypadkach można zauważyć, że kod udostępniony w serwisie GitHub zawiera częściowe rozwiązania niektórych ćwiczeń. Można je potraktować jako wskazówki zachęcające czytelników do zbadania innych rozwiązań.

W wielu przypadkach, aby potwierdzić, że kod faktycznie rozwiązuje problem, w ćwiczeniach będzie potrzebne napisanie przypadków testów jednostkowych. Często są one prawie identyczne z przypadkami testów jednostkowych, które już są dostępne w repozytorium GitHub. Aby potwierdzić, że funkcja działa, czytelnik będzie musiał zastąpić przykładową nazwę funkcji z książki własnym rozwiązaniem.

Klasyfikacja stanu

Aplikacja webowa może obejmować wiele różnych serwerów, bazę danych i mieć zainstalowane różne komponenty oprogramowania. Osoby odpowiedzialne za niezawodność witryny powinny móc się dowiedzieć, czy wszystko poprawnie działa. Kiedy coś ulegnie awarii, będą one chciały poznać szczegóły.

W ramach monitorowania odpowiedzialna aplikacja może gromadzić informacje o stanie różnych komponentów i podsumowywać stan w postaci ogólnej wartości „kondycji" systemu. Chodzi o to, aby wykonać swego rodzaju „redukcję" informacji o stanie.

Każda usługa składowa ma adres URL, do którego można wysłać polecenie `ping` w celu uzyskania informacji o stanie. Wyniki mogą przyjąć jedną z czterech wartości:

- Całkowity brak odpowiedzi. Usługa nie działa. System jest w złym stanie.
- Odpowiedź uzyskana poza dopuszczalnym oknem czasowym. Nawet jeśli jest to odpowiedź `"working"` wskazująca na to, że usługa działa, to jej działanie jest zakłócone.
- Odpowiedź `"not working"`. Taka odpowiedź wskazuje na prawie tak samo zły stan systemu, jak w przypadku całkowitego braku reakcji. W systemie występuje poważny problem, ale skoro została udzielona odpowiedź, to znaczy, że oprogramowanie monitorujące działa.
- Odpowiedź `"working"`. Wskazuje na to, że system jest sprawny. To jest idealna odpowiedź.

Stan tworzy kolekcja trójelementowych krotek: (*usługa*, *stan*, *czas_odpowiedzi*). Element *usługa* jest nazwą taką, jak „podstawowa baza danych", „router" lub nazwa dowolnej innej usługi wchodzącej w skład rozproszonej aplikacji webowej. Element *stan* to wartość tekstowa `"working"` bądź `"not working"`. Element *czas_odpowiedzi* to liczba milisekund oczekiwania na odpowiedź. Typowe wartości to 10 – 50.

Ogólna wartość „kondycji" systemu jest jedną z następujących wartości:

- `Stopped` — przynajmniej jedna usługa nie odpowiada.
- `Degraded` — istnieje co najmniej jedna usługa, która odpowiedziała po czasie dłuższym niż dopuszczalny przedział dla działającej usługi wynoszący 50 milisekund lub mniej. Może również istnieć co najmniej jedna usługa, która udzieliła odpowiedzi `"not working"`.
- `Running` — wszystkie usługi działają i odpowiadają w ciągu 50 milisekund.

Oto dwie możliwe implementacje takiego systemu:

- Napisz cztery funkcje filtrujące. Zastosuj filtry do sekwencji wartości stanu i policz, ile odpowiedzi pasuje do każdego filtra. Na podstawie liczby dopasowań zdecyduj, jaka jest ogólna kondycja systemu.
- W celu określenia kondycji systemu napisz następujące mapowanie: 2 dla wskazania `Stopped`, 1 dla wskazania `Degraded` lub 0 dla wszystkich innych komunikatów o stanie usługi. Maksymalna wartość tego wektora oznacza ogólną kondycję systemu.

Zaimplementuj oba warianty rozwiązania. Porównaj wynikowy kod pod kątem przejrzystości i ekspresywności.

Klasyfikacja stanu, część II

W poprzednim ćwiczeniu usługi zgłaszały tekstową wartość stanu `"working"` bądź `"not working"`.

Zanim przejdziesz dalej, dokończ poprzednie ćwiczenie lub opracuj działający projekt rozwiązania poprzedniego ćwiczenia.

Ze względu na ulepszenia technologiczne wartości stanu dla niektórych usług zawierają trzecią wartość: `"degraded"`. Ma to takie same implikacje jak długi czas odpowiedzi usługi. Może to zmienić projekt. Z pewnością zmieni implementację.

Opracuj implementację, która z wdziękiem obsługuje koncepcję dodatkowych lub odrębnych komunikatów o stanie. Chodzi o odizolowanie sprawdzania komunikatów o stanie od funkcji, którą można łatwo wymienić. Na przykład możemy zacząć od trzech

funkcji oceniających wartości stanu: `is_stopped()`, `is_degraded()` i `is_working()`. Gdy jest wymagana zmiana, możemy napisać nową wersję, `is_degraded_2()`, której można użyć zamiast starej funkcji `is_degraded()`.

Celem jest stworzenie aplikacji, która nie wymaga zmiany w implementacji żadnej konkretnej funkcji. Zamiast tego dodawane są nowe funkcje. Te nowe funkcje w celu realizacji rozszerzonych celów będą korzystać z funkcji istniejących.

Optymalizacja parsera plików

W punkcie „Spłaszczanie danych podczas mapowania", aby wyodrębnić ciąg liczb z tekstu zawierającego spacje, użyliśmy następującego wyrażenia:

```
(
    v
    for line in text.splitlines()
        for v in line.split()
)
```

Definicja metody `split()` zawiera znak `\n`, który jest wykorzystywany również przez metodę `splitlines()`. Wygląda na to, że kod tej metody można zoptymalizować tak, aby używał wyłącznie metody `split()`.

Po wykonaniu tego zadania zmień źródłowy tekst w przykładzie na:

```
>>> text = """2,3,5,7,11,13,17,19,23,29
... 31,37,41,43,47,53,59,61,67,71
... 73,79,83,89,97,101,103,107,109,113
... 127,131,137,139,149,151,157,163,167,173

... 179,181,191,193,197,199,211,223,227,229
... """
```

Powyższy tekst możemy sparsować za pomocą jednokrotnego użycia metody `split()`. Wymaga to przekształcenia pojedynczej, długiej sekwencji wartości na postać wielu wierszy i kolumn.

Czy takie rozwiązanie jest szybsze niż użycie metod `splitlines()` i `split()`?

Chcesz tworzyć wydajny kod?
Naucz się programowania funkcyjnego!

Mimo że Python nie jest typowym językiem programowania funkcyjnego, umożliwia pisanie kodu w sposób właściwy dla tego podejścia. W efekcie można tworzyć zwięzłe i eleganckie programy, które działają szybciej i zużywają mniej zasobów. Jeśli uważasz, że te argumenty uzasadniają zapoznanie się z funkcyjnym podejściem do programowania w Pythonie, to ta książka jest dla Ciebie.

Dzięki temu praktycznemu podręcznikowi zrozumiesz, kiedy i dlaczego warto zastosować myślenie funkcyjne, a także jak korzystać z technik funkcyjnych w różnych scenariuszach. Dowiesz się również, jakie narzędzia i biblioteki przeznaczone do tego celu są dostępne w Pythonie i jak używać wyrażeń generatorowych, list składanych i dekoratorów. W tym wydaniu znalazły się nowe rozdziały dotyczące złożonych obiektów bezstanowych, funkcji kombinatorycznych i pakietu toolz, zawierającego zbiór modułów wspomagających pisanie programów funkcyjnych. Umieszczono tu ponadto sporo ciekawych przykładów, dotyczących choćby eksploracyjnej analizy danych i ich czyszczenia.

W książce między innymi:

- najciekawsze biblioteki i wbudowane funkcje wyższego rzędu w Pythonie
- tworzenie funkcji generatorowych i leniwe wartościowanie
- implementacja dekoratorów do kompozycji funkcyjnej
- podpowiedzi typów w Pythonie
- obsługa współbieżności i implementacja usług sieciowych
- biblioteka PyMonad i tworzenie symulacji z obsługą stanów

Steven F. Lott programuje w Pythonie od ponad 30 lat; używa go do tworzenia różnego rodzaju narzędzi i aplikacji. Jest autorem cenionych książek o programowaniu. Uważa siebie za technonomadę i mieszka na łodzi, zwykle zacumowanej gdzieś na wschodnim wybrzeżu Stanów Zjednoczonych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-0063-9
9 788328 900639
Cena: 89,00 zł
<packt>