Steven F. Lott

Python

Programowanie funkcyjne

Helion

Packt›

Tytuł oryginału: Functional Python Programming, Second Edition

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-5069-4

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

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

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

Printed in Poland

Spis treści

5

ROZDZIAŁ

Funkcje wyższego rzędu

Bardzo ważną cechą paradygmatu programowania funkcyjnego są funkcje wyższego rzędu. Są to funkcje, które akceptują funkcje jako argumenty lub zwracają funkcje jako wyniki. Python oferuje kilka rodzajów funkcji wyższego rzędu. Przyjrzymy się im oraz ich niektórym logicznym rozszerzeniom.

Jak można zauważyć, istnieją trzy odmiany 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.

Python oferuje kilka funkcji wyższego rzędu tego pierwszego typu. W tym rozdziale przyjrzymy się wbudowanym funkcjom wyższego rzędu tego rodzaju. W kolejnych rozdziałach przyjrzymy się kilku modułom bibliotecznym, które oferują funkcje wyższego rzędu.

Koncepcja funkcji, która zwraca inne funkcje, może wydawać się nieco dziwna. Jednak gdy spojrzymy na klasę `Callable`, możemy zauważyć, że definicja klasy jest funkcją, która przy ocenie wartości zwraca obiekty `Callable`. Jest to jeden z przykładów funkcji, która tworzy inną funkcję.

Wśród funkcji, które pobierają funkcje i tworzą funkcje, można znaleźć zarówno złożone klasy wywoływalne, jak i funkcje dekoratorów. Koncepcję dekoratorów wprowadzimy w tym rozdziale, ale bardziej wnikliwą ich analizę odłożymy do rozdziału 11. „Techniki projektowania dekoratorów”.

Czasami chcielibyśmy, aby w Pythonie były wersje wyższego rzędu funkcji przetwarzania kolekcji opisanych w poprzednim rozdziale. W tym rozdziale w celu wykonania redukcji określonych pól wyodrębnionych z większej krotki zaprezentujemy wzorzec projektowy *`zredukuj(wyodrębnij())`*. Przyjrzymy się również definiowaniu własnej wersji tych powszechnie wykorzystywanych funkcji przetwarzania kolekcji.

W tym rozdziale przyjrzymy się następującym funkcjom:

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

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

Szereg funkcji wyższego rzędu jest dostępnych w module `itertools`. Przyjrzymy się temu modułowi w rozdziale 8. „Moduł itertools" oraz w rozdziale 9. „Dodatkowe techniki itertools".

Ponadto w module `functools` jest dostępna funkcja ogólnego przeznaczenia `reduce()`. Przyjrzymy się jej w rozdziale 10. „Moduł funktools", ponieważ nie jest tak powszechnie stosowana jak inne funkcje wyższego rzędu opisane w tym rozdziale.

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

> Funkcje `max()`, `min()` i `sorted()` mają zarówno zachowania domyślne, jak i zachowania funkcji wyższego rzędu. Funkcję można dostarczyć za pomocą argumentu `key=`. Funkcje `map()` i `filter()` przyjmują funkcję jako pierwszy argument pozycyjny.

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.

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.
- Wykorzystanie wzorca *`rozpakuj(przetwarzaj(opakuj()))`*. Za jego pomocą uzyskamy najdłuższy i najkrótszy odcinek. Na tej podstawie możemy wydobyć element opisujący odległość, jeśli to on jest potrzebny.
- Wykorzystanie `max()` i `min()` jako funkcji wyższego rzędu — wstawienie funkcji, która wykonuje ekstrakcję istotnych wartości odległości.

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

```
from ch02_ex3 import (
    float_from_pair, lat_lon_kml, limits, haversine, legs
)
path = float_from_pair(float_lat_lon(row_iter_kml(source)))
trip = tuple(
    (start, end, round(haversine(start, end), 4))
        for start, end in legs(iter(path)))
```

Powyższy skrypt wymaga, aby argument `source` był otwartym plikiem zawierającym punkty danych w formacie KML. Najważniejszy obiekt `trip` to krotka złożona z pojedynczych odcinków. 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.

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:

```
>>> long = max(dist for start, end, dist in trip)
>>> short = min(dist for start, end, dist in trip)
```

Aby wyodrębnić odpowiedni element z każdego odcinka należącego do krotki `trip`, użyliśmy funkcji generatorowej. Musieliśmy powtórzyć funkcję generatorową, ponieważ każde wyrażenie generatorowe może być użyte tylko raz.

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

```
>>> long
129.7748
>>> short
0.1731
```

Poniżej zamieszczono wersję z wykorzystaniem wzorca *`rozpakuj(przetwarzaj(opakuj()))`*. Aby to było jasne, w przykładzie wykorzystano funkcje o nazwach `wrap()` i `unwrap()`. Oto funkcje i ich wywołania:

```
from typing import Iterator, Iterable, Tuple, Any

Wrapped = Tuple[Any, Tuple]
def wrap(leg_iter: Iterable[Tuple]) -> Iterable[Wrapped]:
    return ((leg[2], leg) for leg in leg_iter)
def unwrap(dist_leg: Tuple[Any, Any]) -> Any:
    distance, leg = dist_leg
    return leg

long = unwrap(max(wrap(trip)))
short = unwrap(min(wrap(trip)))
```

W przeciwieństwie do poprzedniej wersji funkcje `max()` i `min()` lokalizują wszystkie atrybuty odcinków o najdłuższej i najkrótszej odległości. Zamiast po prostu wyodrębniać odległości, najpierw umieszczamy odległości w każdej opakowanej krotce. Możemy następnie użyć domyślnych formatów funkcji `min()` i `max()` w celu przetworzenia dwóch krotek, które zawierają odległość i szczegóły odcinka. Po przetworzeniu możemy usunąć pierwszy element, pozostawiając tylko szczegóły odcinka.

Wynik ma następującą postać:

```
((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)
((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
```

```
long = max(trip, key=by_dist)
short = min(trip, key=by_dist)
```

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. Wszystkie funkcje wyższego rzędu w Pythonie do dostarczenia funkcji, która będzie używana do wyodrębnienia niezbędnej wartości klucza, używają parametru ze słowem kluczowym `key=`.

Aby lepiej wyobrazić sobie sposób, w jaki funkcja `max()` używa funkcji `key`, możemy skorzystać z poniższego kodu:

```
from typing import Iterable, Any, Callable

def max_like(trip: Iterable[Any], key: Callable) -> Any:
    wrap = ((key(leg), leg) for leg in trip)
    return sorted(wrap)[-1][1]
```

Funkcje `max()` i `min()` zachowują się tak, jakby wynik podanej funkcji `key()` był używany do opakowania każdego elementu w sekwencji w dwuelementową krotkę. Po posortowaniu dwuelementowych krotek wybranie pierwszej z nich (jako minimum) lub ostatniej (jako maksimum) pozwala na zwrócenie krotki z wartością ekstremalną. Można ją zdekomponować, aby odzyskać oryginalną wartość.

Aby funkcja `key()` była opcjonalna, należy określić jej wartość domyślną `lambda x: x`.

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`:

```
long = max(trip, key=lambda leg: leg[2])
short = min(trip, key=lambda leg: leg[2])
```

Do zastosowanego wyrażenia lambda zostanie przekazany element sekwencji. 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. 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:

```
>>> leg = ((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)
>>> start = lambda x: x[0]
>>> end = lambda x: x[1]
>>> dist = lambda x: x[2]
>>> dist(leg)
129.7748
```

Python oferuje dwa sposoby przypisywania opisowych nazw do elementów krotek: nazwane krotki (obiekty `namedtuple`) i kolekcje wyrażeń lambda. Oba są równoważne. Możemy używać wyrażeń lambda zamiast nazwanych krotek.

Aby rozszerzyć ten przykład, przyjrzymy się, jak uzyskać wartości szerokości lub długości geograficznej punktu początkowego lub końcowego. Można to zrobić poprzez zdefiniowanie dodatkowych wyrażeń lambda.

Oto kontynuacja wcześniejszej interaktywnej sesji:

```
>>> start(leg)
(27.154167, -80.195663)

>>> lat = lambda x: x[0]
>>> lon = lambda x: x[1]
>>> lat(start(leg))
27.154167
```

Nie ma wyraźnej przewagi z używania jako sposobu na wyodrębnianie pól wyrażeń lambda zamiast nazwanych krotek. Zbiór obiektów lambda do wyodrębniania pól wymaga zdefiniowania więcej linijek kodu niż nazwana krotka. Z drugiej strony wyrażenia lambda pozwalają na korzystanie z notacji prefiksowej, która może być czytelniejsza w kontekście programowania funkcyjnego. Co ważniejsze, jak przekonamy się później w przykładzie użycia funkcji `sorted()`, z wyrażeń lambda można korzystać bardziej efektywnie niż z atrybutów nazwanych krotek w funkcjach `sorted()`, `min()` i `max()`.

Wyrażenia lambda i rachunek lambda

W książce poświęconej czysto funkcyjnemu językowi programowania byłoby konieczne wyjaśnienie rachunku lambda 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.

Używając funkcji `functools.partial`, możemy zaimplementować rozwijanie funkcji. 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 jest ograniczona do wartości dodatnich. Mapowanie można wykonać poprzez obliczenie lub interpolację tabeli.

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. Wyniki te mają 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, 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]
```

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.

Takie obliczenia możemy wykonać za pomocą funkcji `map()` w następujący sposób:

```
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:

```
((start(x), end(x), dist(x)*6076.12/5280) for x in trip)
```

W wyrażeniu generatorowym wykonaliśmy identyczne przetwarzanie dla każdego elementu kolekcji.

Ważną różnicą pomiędzy funkcją `map()` a wyrażeniem generatorowym jest to, że w funkcji `map()` można posługiwać się definicją wyrażenia lambda lub funkcji wielokrotnego użytku. Lepszym rozwiązaniem jest wykorzystanie następującego kodu:

```
to_miles = lambda x: start(x), end(x), dist(x)*6076.12/5280
trip_m = map(to_miles, trip)
```

W tym wariancie oddzieliliśmy transformację `to_miles` od procesu stosowania tej transformacji do danych.

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(one_iterable, another_iterable))
```

Utworzyliśmy krotki argumentów na podstawie dwóch (lub większej liczby) równoległych obiektów iterowalnych i zastosowaliśmy funkcję do tych krotek argumentów. Możemy również spojrzeć na to w następujący sposób:

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

W tym przypadku zastąpiliśmy funkcję `map()` równoważnym wyrażeniem generatorowym.

Moglibyśmy uogólnić całość do następującej postaci:

```
def star_map(function, *iterables)
    return (function(*args) for args in zip(*iterables))
```

Istnieje jednak lepsze podejście. W istocie nie potrzebujemy tych technik. 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 specjalnego rodzaju obiektu iterowalnego:

```
>>> waypoints = range(4)
>>> zip(waypoints, waypoints[1:])
<zip object at 0x101a38c20>
>>> list(_)
[(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()`.

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 ch02_ex3 import (lat_lon_kml, float_from_pair, haversine)

path = tuple(float_from_pair(lat_lon_kml()))
distances_1 = map(
    lambda s_e: (s_e[0], s_e[1], haversine(*s_e)),
    zip(path, path[1:])
)
```

Załadowaliśmy niezbędną sekwencję punktów do 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.

Wykorzystanie funkcji filter() do przekazywania lub odrzucania danych

Zadaniem funkcji `filter()` jest użycie funkcji decyzyjnej zwanej predykatem do każdej wartości w kolekcji. Decyzja `True` oznacza, że wartość zostanie przekazana; w przeciwnym razie wartość zostanie odrzucona. 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= 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 0x101d5de50>
>>> 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.

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*):

$$\{x \mid 0 \le x < 10 \wedge (x \bmod 3 = 0 \vee x \bmod 5 = 0)\}$$

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 ch01_ex1 import isprimeg
>>> list(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 funkcję z innego modułu 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.

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 stats import mean, stdev, z

dist_data = list(map(dist, trip))
μ_d = mean(dist_data)
 _d = stdev(dist_data)
outlier = lambda leg: z(dist(leg), μ_d,  _d) > 3
```

```
print("Elementy odstające", list(filter(outlier, trip)))
```

Dla każdego odcinka w kolekcji `trip` zmapowaliśmy funkcję odległości. Ponieważ robimy z wynikiem kilka rzeczy, musimy zmaterializować obiekt listy. Nie możemy polegać na iteratorze, ponieważ pierwsza funkcja go wyczerpie. Zmaterializowanej listy możemy następnie użyć do obliczenia statystyk populacji μ`_d` i `_d` zawierających średnie i odchylenie standardowe.

Na podstawie statystyk użyliśmy do filtrowania danych wyrażenia lambda `outlier`. Jeśli znormalizowana wartość jest zbyt duża, dane odstają.

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. Żadna podróż nie może mieć znormalizowanej odległości mniejszej niż –1,407.

> 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 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()`:

```
>>> tail = iter([1, 2, 3, None, 4, 5, 6].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. 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),
 ((35.028175, -76.682495), (35.031334, -76.682663), 0.1898),
 ((27.154167, -80.195663), (29.195168, -81.002998), 129.7748)
]
```

Korzystając z wyrażenia `distlambda`, aby wyodrębnić odległość z każdej krotki, posortowaliśmy dane podróży. Funkcja `dist` jest zdefiniowana następująco:

```
dist = lambda leg: leg[2]
```

To pokazuje możliwość użycia prostego wyrażenia lambda w celu rozłożenia złożonej krotki na jej elementy składowe.

Pisanie funkcji wyższego rzędu

Możemy zidentyfikować trzy odmiany funkcji wyższego rzędu. Oto one:

- Funkcje, które pobierają funkcję jako jeden z argumentów.
- Funkcje, które zwracają funkcję. Popularnym przykładem tego rodzaju funkcji wyższego rzędu są obiekty klasy `Callable`. Funkcja, która zwraca wyrażenie generatorowe, może być uważana za funkcję wyższego rzędu.
- Funkcje, które pobierają funkcję jako argument i zwracają inną funkcję. Częstym tego przykładem jest funkcja `functools.partial()`. Omówienie tego tematu odłożymy do rozdziału 10. „Moduł functools". Drugim przykładem jest dekorator. Opiszemy go w rozdziale 11. „Techniki projektowania dekoratorów".

Rozszerzymy te proste wzorce, używając funkcji wyższego rzędu, aby jednocześnie przekształcać strukturę danych. Możemy wykonać kilka typowych przekształceń, na przykład:

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

Powszechnie wykorzystywanym przykładem funkcji zwracającej obiekt wywoływalny jest egzemplarz klasy `Callable`. Przyjrzymy się mu jako sposobowi pisania elastycznych funkcji, do których można wstrzykiwać parametry konfiguracyjne.

W tym rozdziale zaprezentujemy również proste dekoratory. Dokładniejszy opis dekoratorów odłożymy do rozdziału 11. „Techniki projektowania dekoratorów".

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. Funkcjom Pythona 3.4, takim jak `imap()`, `ifilter()` oraz `ifilterfalse()`, przyjrzymy się w rozdziale 8. „Moduł itertools".

Istnieją trzy w dużym stopniu równoważne sposoby wyrażania mapowania. Załóżmy, że mamy jakąś funkcję `f(x)` i pewną kolekcję obiektów `C`. Mamy trzy całkowicie równoważne sposoby wyrażenia mapowania. Oto one:

- Funkcja `map()`:

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

- Wyrażenie generatorowe:

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

- Funkcja generatorowa z instrukcją `yield`:

```
def mymap(f, C):
    for x in C:
        yield f(x)
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`:

```
def myfilter(f, C):
    for x in C:
        if f(x):
            yield x
myfilter(f, C)
```

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:

- 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 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.

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.

Podczas projektowania mapowań, które łączą zbyt wiele transformacji w jednej funkcji, trzeba zachować ostrożność. O ile to możliwe, należy unikać tworzenia funkcji, które nie są zwięzłe lub nie wyrażają pojedynczej koncepcji. Ponieważ Python nie ma kompilatora optymalizacyjnego, możemy być zmuszeni do ręcznej optymalizacji wolno działających aplikacji poprzez łączenie funkcji. Tego rodzaju optymalizacje powinny być wykonywane w ostateczności, dopiero po sprofilowaniu wolno działającego programu.

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 typing import Callable, Iterable, Tuple, Iterator, Any

Conv_F = Callable[[float], float]
Leg = 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:

```
to_miles = lambda nm: nm*5280/6076.12
to_km = lambda nm: nm*1.852
to_nm = lambda nm: nm
```

Z tej funkcji można następnie skorzystać w celu wyodrębnienia odległości i zastosowania do niej funkcji konwersji:

```
convert(to_miles, trip)
```

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:

```
fst = lambda x: x[0]
snd = lambda x: x[1]
sel2 = lambda x: x[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()`. Kompozycję funkcyjną w Pythonie możemy wyrazić, używając dodatkowego wyrażenia lambda, na przykład:

```
to_miles = lambda s_e_d: to_miles(sel2(s_e_d))
```

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

```
(to_miles(s_e_d) for s_e_d in trip)
```

Chociaż ta druga wersja jest nieco bardziej ogólna, nie wydaje się być zbyt przydatna.

Należy zapamiętać, że funkcja wyższego rzędu `convert()` przyjmuje funkcję jako argument i zwraca jako wynik funkcję generatorową. Funkcja `convert()` nie jest funkcją generatorową, ponieważ niczego nie generuje. Wynikiem funkcji `convert()` jest wyrażenie generatorowe, które trzeba ocenić w celu skumulowania pojedynczych wartości. Użyliśmy wskazówki typu `Iterator[float]`, aby podkreślić, że wynikiem jest iterator — podklasa funkcji generatorowych w Pythonie.

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 zapisywania obliczonych wyników w celu stworzenia konstrukcji, w których nie ma konieczności powtarzania obliczeń oraz utrzymywania złożonych obiektów zmieniających stan.

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

```
from ch02_ex3 import (
    float_from_pair, lat_lon_kml, limits, haversine, legs
)

path = float_from_pair(float_lat_lon(row_iter_kml(source)))
trip = tuple(
    (start, end, round(haversine(start, end), 4))
    for start, end in legs(iter(path))
)
```

Możemy nieco zmodyfikować ten kod, aby utworzyć funkcję wyższego rzędu, która oddziela opakowanie od innych funkcji. Funkcję możemy zdefiniować w następujący sposób:

```
from typing import Callable, Iterable, Tuple, Iterator

Point = Tuple[float, float]
Leg_Raw = Tuple[Point, Point]
Point_Func = Callable[[Point, Point], float]
Leg_D = 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:

```
path = float_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.

Oto nieco inny format tej funkcji:

```
from typing import Callable, Iterable, Tuple, Iterator, Any
Point = Tuple[float, float]
Leg_Raw = Tuple[Point, Point]
Point_Func = Callable[[Point, Point], float]
Leg_P_D = Tuple[Leg_Raw, ...]

def cons_distance3(
        distance: Point_Func,
        legs_iter: Iterable[Leg_Raw]) -> Iterator[Leg_P_D]:
    return (
        leg + (round(distance(*leg), 4),) # 1-tuple
        for leg in legs_iter
    )
```

Budowanie nowego obiektu na podstawie starego jest w tej wersji nieco czytelniejsze. Wynikowy iterator używa odcinków obiektu `trip` zdefiniowanych jako `Leg_Raw`, czyli krotki złożonej z dwóch punktów. Oblicza odległość odcinka w celu zbudowania wynikowej trójelementowej krotki z oryginalnym obiektem `Leg_Raw` i powiązaną z nim odległością.

Ponieważ obie funkcje `cons_distance()` pobierają funkcję jako argument, możemy skorzystać z tej cechy, aby zapewnić alternatywną formułę odległości. Na przykład możemy użyć metody `math.hypot(lat(start)-lat(end), lon(start)-lon(end))`, aby obliczyć mniej poprawną odległość pomiędzy płaszczyznami wzdłuż każdego odcinka.

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 stworzyć 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']
```

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

```
from numbers import Number
from typing import Callable, Iterator

Num_Conv = Callable[[str], Number]

def numbers_from_rows(
        conversion: Num_Conv, text: str) -> Iterator[Number]:
    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:

```
print(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:

```
map(float,
    value
    for line in text.splitlines()
        for value in line.split()
)
```

To może pomóc nam w zrozumieniu ogólnej struktury algorytmu. Stosowana reguła to tzw. **chunking** (dosł. kawałkowanie) — możemy wyodrębnić szczegóły funkcji o opisowej nazwie i pracować z funkcją w nowym 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 typing import Iterator, Tuple

def group_by_iter(n: int, items: Iterator) -> Iterator[Tuple]:
    row = tuple(next(items) for i in range(n))
    while row:
        yield row
        row = tuple(next(items) for i in range(n))
```

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 jest stosunkowo niewydajna, zoptymalizowaliśmy ją do jawnej pętli `while`.

Z funkcji można skorzystać w następujący sposób:

```
group_by_iter(7,
    filter(lambda x: x%3==0 or x%5==0, range(100))
)
```

Spowoduje to zgrupowanie wyników funkcji `filter()` do obiektu iterowalnego utworzonego przez funkcję `range()`.

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

```
def group_filter_iter(
        n: int, pred: Callable, items: Iterator) -> Iterator:
```

```
    subset = filter(pred, items)
    row = tuple(next(subset) for i in range(n))
    while row:
        yield row
        row = tuple(next(subset) for i in range(n))
```

Ta funkcja 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:

```
group_filter_iter(
    7,
    lambda x: x%3==0 or x%5==0,
    range(1,100)
)
```

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.

Pisanie funkcji generatorowych

Wiele funkcji można zwięźle wyrazić jako wyrażenie generatorowe. Wcześniej widzieliśmy, że prawie każdy rodzaj mapowania lub filtrowania może być zrealizowany jako wyrażenie generatorowe. Można je również zrealizować za pomocą wbudowanej funkcji wyższego rzędu, takiej jak `map()`, `filter()`, lub za pomocą funkcji generatorowej. Biorąc pod uwagę wiele funkcji generatorowych, musimy zachować ostrożność, aby nie odstępować od przewodniej zasady programowania funkcyjnego: bezstanowej oceny wartości funkcji.

Używanie Pythona do programowania funkcyjnego oznacza balansowanie pomiędzy programowaniem czysto funkcyjnym a programowaniem imperatywnym. Należy zidentyfikować i wyizolować miejsca, w których trzeba sięgnąć do imperatywnego kodu Pythona, ponieważ nie istnieje czysto funkcyjna alternatywa.

Gdy potrzebujemy instrukcji Pythona, jesteśmy zobligowani do pisania funkcji generatorowych. Wymienione poniżej funkcje nie są dostępne w wyrażeniach generatorowych:

- Kontekst `with` do pracy z zasobami zewnętrznymi. Powrócimy do tego tematu w rozdziale 6. „Rekurencje i redukcje” przy okazji omawiania parsowania plików.
- Instrukcja `while` do iterowania w nieco bardziej elastyczny sposób od instrukcji `for`. Przykład jej działania pokazano wcześniej, w punkcie „Spłaszczanie danych podczas mapowania”.

- Instrukcja `break` lub `return` do zaimplementowania wyszukiwania, które wcześniej kończy pętlę.
- Konstrukcja `try-except` do obsługi wyjątków.
- Wewnętrzna definicja funkcji. Przyglądaliśmy się temu w kilku przykładach w rozdziale 1. „Zrozumieć programowanie funkcyjne" i w rozdziale 2. „Podstawowe pojęcia programowania funkcyjnego". Powrócimy do tego tematu w rozdziale 6. „Rekurencje i redukcje".
- Naprawdę złożona sekwencja `if-elif`. Próba wyrażenia więcej niż jednej alternatywy za pomocą wyrażeń warunkowych `if-else` może doprowadzić do skomplikowania kodu.
- Istnieją także mniej używane konstrukcje Pythona, takie jak `for-else`, `while-else`, `try-else` oraz `try-else-finally`. Są to wszystko własności poziomu instrukcji, które nie są dostępne w wyrażeniach generatorowych.

Instrukcja `break` jest najczęściej używana do wcześniejszego zakończenia przetwarzania kolekcji. Możemy zakończyć przetwarzanie po pierwszym elemencie, który spełnia określone kryteria. Jest to wersja funkcji `any()` używana do znalezienia wartości o danej właściwości. Możemy również zakończyć po przetworzeniu większej liczby elementów kolekcji, ale nie wszystkich.

Znalezienie jednej wartości można zwięźle wyrazić jako `min(`*`złożone_wyrażenie`*`)` lub `max(`*`złożone_`*
`wyrażenie``)`. W takich przypadkach staramy się przeanalizować wszystkie wartości, aby upewnić się, że prawidłowo znaleźliśmy wartość minimalną lub maksymalną.

Istnieją również sytuacje — gdy interesuje nas tylko pierwsza wartość `True` — kiedy wystarcza nam funkcja `first(`*`funkcja`*`, `*`kolekcja`*`)`. Chcielibyśmy, aby przetwarzanie zakończyło się jak najwcześniej, by zapobiec konieczności wykonywania zbędnych obliczeń.

Możemy zdefiniować taką funkcję w następujący sposób:

```
def first(predicate: Callable, collection: Iterable) -> Any:
    for x in collection:
        if predicate(x): return x
```

Iterowaliśmy po kolekcji, stosując podaną funkcję predykatu. Jeśli wynik predykatu ma wartość `True`, funkcja zwraca powiązaną wartość kolekcji i przestaje przetwarzać obiekt iterowalny. Jeśli kolekcja się wyczerpie, zostanie zwrócona domyślna wartość `None`.

Możemy również pobrać wersję funkcji `first` z PyPi. Odmianę implementacji tego pomysłu zawiera moduł `first`. Więcej informacji na ten temat można znaleźć pod adresem *https://pypi.python.org/pypi/first*.

Funkcję `first` można wykorzystać jako mechanizm pomocniczy przy próbie ustalenia, czy liczba jest liczbą pierwszą, czy nie. Oto funkcja, która sprawdza, czy liczba jest liczbą pierwszą:

```
import math
def isprimeh(x: int) -> bool:
    if x == 2: return True
    if x % 2 == 0: return False
```

```
    factor= first(
        lambda n: x%n==0,
        range(3, int(math.sqrt(x)+.5)+1, 2))
    return factor is None
```

Ta funkcja obsługuje kilka przypadków brzegowych — uwzględnia fakt, że 2 jest liczbą pierwszą, a każda inna liczba parzysta jest złożona. Następnie wykorzystuje zdefiniowaną wcześniej funkcję `first()` w celu znalezienia pierwszego czynnika w wybranej kolekcji.

Kiedy funkcja `first()` zwróci czynnik, wartość tego czynnika nie ma znaczenia. W tym konkretnym przykładzie istotne jest samo istnienie tego czynnika. Dlatego funkcja `isprimeh()` zwraca `True`, jeśli nie znaleziono żadnego czynnika.

Możemy zrobić coś podobnego do obsługi wyjątków danych. Poniżej znajduje się wersja funkcji `map()`, która również filtruje nieprawidłowe dane:

```
def map_not_none(func: Callable, source: Iterable) -> Iterator:
    for x in source:
        try:
            yield func(x)
        except Exception as e:
            pass # Aby pomóc w debugowaniu, użyj print(e)
```

Powyższa funkcja przetwarza elementy obiektu iterowalnego, przypisując każdy element do zmiennej `x`. Próbuje zastosować tę funkcję do elementu; jeśli nie zostanie zgłoszony żaden wyjątek, do wyniku dołączana jest uzyskana wartość. Jeśli zostanie zgłoszony wyjątek, nieprawidłowy element źródłowy będzie bezgłośnie odrzucony.

Może to być przydatne w przypadku danych zawierających wartości, które nie mają zastosowania lub są niedostępne. Zamiast przetwarzać złożone filtry w celu wykluczenia tych wartości, próbujemy przetworzyć same wartości i usunąć te, które nie są prawidłowe.

Do mapowania wartości różnych od `None` możemy użyć funkcji `map()`, jak pokazano poniżej:

```
data = map_not_none(int, some_source)
```

Do każdej wartości w `some_source` zastosujemy funkcję `int()`. Gdyby parametr `some_source` był iterowaną kolekcją ciągów znaków, to powyższy kod mógłby być wygodnym sposobem na odrzucenie ciągów, które nie reprezentują liczb.

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

instrukcji Pythona. Podczas tworzenia funkcji wyższego rzędu oprócz używania instrukcji możemy również zastosować statyczną konfigurację.

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 typing import Callable, Optional, Any

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

```
null_log_scale = NullAware(math.log)
```

Spowoduje to utworzenie nowej funkcji i przypisanie jej nazwy `null_log_scale()`. 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)
>>> list(scaled)
[2.302585092994046, 4.605170185988092, None, 3.912023005428146,
4.0943445622221]
```

Mniej popularnym podejściem jest stworzenie i używanie emitowanej funkcji w jednym wyrażeniu, jak pokazano poniżej:

```
>>> scaled = map(NullAware(math.log), some_data)
>>> list(scaled)
[2.302585092994046, 4.605170185988092, None, 3.912023005428146,
4.0943445622221]
```

W wyniku oceny wartości `NullAware(math.log)` powstała funkcja. Następnie wykorzystano tę anonimową funkcję w funkcji `map()` do przetwarzania obiektu iterowalnego `some_data`.

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. Podczas pracy z funkcjami skalarnymi należy wziąć pod uwagę kilka złożonych kwestii projektowych. Przy pracy z iterowalnymi kolekcjami musimy być nieco bardziej ostrożni.

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.

Powrócimy do pokazanej wcześniej złożonej funkcji `sum_filter_f()`. Oto wersja zbudowana na podstawie definicji klasy wywoływalnej:

```
from typing import Callable, Iterable

class Sum_Filter:
    __slots__ = ["filter", "function"]
    def __init__(self,
            filter: Callable[[Any], bool],
            func: Callable[[Any], float]) -> None:
    self.filter = filter
    self.function = func
def __call__(self, iterable: Iterable) -> float:
    return sum(
        self.function(x)
        for x in iterable
            if self.filter(x)
    )
```

Ta klasa ma dokładnie dwa atrybuty w każdym obiekcie. 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. Jego użycie jest nieco prostsze w porównaniu z poprzednim przykładem `sum_filter_f()`.

Funkcję `count_not_None()` możemy wykorzystać w następujący sposób:

```
N = count_not_none(data)
```

Można z niej skorzystać zamiast z funkcji `sum_filter_f()`:

```
N = sum_filter_f(valid, count_, data)
```

Funkcja `count_not_none()` bazująca na obiekcie wywoływalnym nie wymaga tak wielu argumentów jak funkcja konwencjonalna. To sprawia, że jest znacznie prostsza w użyciu. Może jednak być nieco bardziej niejasna, ponieważ szczegóły działania tej funkcji znajdują się w dwóch miejscach kodu źródłowego: tam, gdzie funkcja została utworzona jako egzemplarz klasy wywoływalnej, oraz tam, gdzie funkcji użyto.

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 także formatom wyrażeń lambda, których można użyć do uproszczenia funkcji wyższego rzędu. 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 (lub obiektem, który jest rodzajem kolekcji) 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 11. „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 powiązanemu z tym problemowi wykonywania redukcji z kolekcji do pojedynczych wartości.

Skorowidz

M

N

O

P

R

S

T

U

W

X

Z

Python: kod funkcyjny i funkcjonalny!

Zgodnie z paradygmatem programowania funkcyjnego największy nacisk należy kłaść na stałe i funkcje. Polega to na konstruowaniu funkcji oraz na obliczaniu wartości wyrażeń. W ten sposób otrzymuje się kod odporny na błędy. Python nie jest w pełni funkcyjnym językiem programowania, jednak pozwala na taki sposób pisania programów. Dzięki temu umożliwia tworzenie zwięzłego i eleganckiego kodu. Na przykład stosowanie wyrażeń generatorowych w Pythonie sprawia, że tworzone programy działają szybciej, ponieważ zużywają mniej zasobów. Niezależnie więc od stosowanego paradygmatu warto zapożyczyć pewne elementy programowania funkcyjnego i wykorzystać je do tworzenia ekspresyjnych i zwięzłych aplikacji w Pythonie.

To znakomity podręcznik dla programistów, którzy chcą wykorzystać techniki i wzorce projektowe z funkcyjnych języków programowania, aby tworzyć w Pythonie zwięzłe, eleganckie i ekspresyjne programy — z czytelnym i łatwym w utrzymaniu kodem. Zawiera ogólny przegląd koncepcji funkcyjnych oraz wyjaśnia tak istotne pojęcia jak funkcje pierwszej klasy, funkcje wyższego rzędu, funkcje czyste, leniwe wartościowanie i wiele innych. Wnikliwie omawia sposób korzystania z tych funkcji w Pythonie 3.6, a także techniki przygotowywania i eksploracji danych. Ponadto pokazuje, w jaki sposób standardowa biblioteka Pythona pasuje do funkcyjnego modelu programowania. Co ważne, w książce znalazło się kilka przykładów prezentujących w praktyce opisane koncepcje.

W tej książce między innymi:

- podstawy modelu programowania funkcyjnego
- działania na kolekcjach danych i przetwarzanie krotek
- projektowanie dekoratorów
- biblioteka PyMonad
- usługi sieciowe a programowanie funkcyjne

Steven F. Lott — ma blisko pięćdziesiąt lat doświadczenia w programowaniu — kiedy rozpoczynał przygodę z kodem, komputery były duże, drogie i rzadkie. Od ponad dziesięciu lat używa Pythona do rozwiązywania problemów biznesowych; napisał kilka cenionych książek o tym języku. Obecnie jest technomadą. Mieszka na wschodnim wybrzeżu USA.

ISBN 978-83-283-5069-4