wolniej lub szybciej. Zamiast spekulować, pomożemy zrozumieć, jak profilować aplikację, aby wiedzieć, która część programu spowalnia jego działanie i gdzie znajdują się wąskie gardła.

Struktury danych

Większość problemów programistycznych można rozwiązać w elegancki i prosty sposób przy użyciu odpowiednich struktur danych. Python oferuje wiele różnych struktur danych i umiejętność właściwego ich wykorzystywania pozwala na uzyskiwanie bardziej czytelnego i stabilnego rozwiązania niż programowanie własnych struktur danych.

Na przykład wszyscy używają `dict`, ale ile razy widziałeś kod, który próbuje uzyskać dostęp do słownika, przechwytując wyjątek `KeyError`, jak poniżej:

```
def get_fruits(basket, fruit):
    try:
        return basket[fruit]
    except KeyError:
        return None
```

lub sprawdzając najpierw, czy klucz istnieje:

```
def get_fruits(basket, fruit):
    if fruit in basket:
        return basket[fruit]
```

Jeśli użyjemy metody `get()` dostępnej w klasie `dict`, to unikamy konieczności przechwytywania wyjątku bądź wstępnego sprawdzania, czy klucz istnieje:

```
def get_fruits(basket, fruit):
    return basket.get(fruit)
```

Metoda `dict.get()` może również zwracać domyślną wartość zamiast `None` – wystarczy dodać do jej wywołania drugi argument:

```
def get_fruits(basket, fruit):
    # Zwraca fruit lub Banana, jeśli nie znaleziono obiektu fruit.
    return basket.get(fruit, Banana())
```

Wielu programistów ma na sumieniu stosowanie podstawowych struktur danych Pythona bez poznania wszystkich oferowanych przez nie metod. Odnosi się to również do zbiorów: dostępne w nich metody mogą rozwiązywać wiele problemów, które w przeciwnym razie wymagałyby napisania zagnieżdżonych bloków `for`/`if`. Na przykład programiści często używają

pętli `for/if` do sprawdzania, czy istnieją elementy nienależące do listy, jak w tym przykładzie:

```
def has_invalid_fields(fields):
    for field in fields:
        if field not in ['foo', 'bar']:
            return True
    return False
```

Pętla przechodzi po wszystkich elementach listy i sprawdza, czy wszystkie z nich są równe `foo` albo `bar`. Jednak można osiągnąć ten cel w bardziej efektywny sposób, bez stosowania pętli:

```
def has_invalid_fields(fields):
    return bool(set(fields) - set(['foo', 'bar']))
```

Ten zmieniony kod przekształca `fields` na zbiór i wyznacza resztę zbioru, odejmując od niego `set(['foo', 'bar'])`. Następnie przekształca zbiór na wartość logiczną, aby sprawdzić, czy pozostały jakieś elementy, które nie są równe ani `foo`, ani `bar`. Zastosowanie zbiorów eliminuje konieczność iteracji po liście i sprawdzania pojedynczych elementów. Prosta operacja na dwóch zbiorach, przeprowadzona wewnętrznie przez Pythona, jest szybsza.

Python oferuje również bardziej zaawansowane struktury danych, które mogą znacznie ułatwiać utrzymywanie kodu. Przeanalizujmy przykład przedstawiony na listingu 10.1.

```
def add_animal_in_family(species, animal, family):
    if family not in species:
        species[family] = set()
    species[family].add(animal)

species = {}
add_animal_in_family(species, 'cat', 'felidea')
```

Listing 10.1. Dodawanie wpisu do słownika zbiorów

Ten kod jest całkiem poprawny, ale ile razy będziemy potrzebować w naszym programie odmiany kodu z listingu 10.1? Dziesiątki? Setki?

Python oferuje strukturę `collections.defaultdict`, która w elegancki sposób rozwiązuje ten problem:

```
import collections

def add_animal_in_family(species, animal, family):
    species[family].add(animal)

species = collections.defaultdict(set)
add_animal_in_family(species, 'cat', 'felidea')
```