TABELA 8.4. Zbiory w Pythonie — cechy charakterystyczne

CECHA	KOMENTARZ
Nieuporządkowanie	Zbiory zawierają elementy nieuporządkowane i nie można sięgać do ich elementów poprzez indeksowanie, znane chociażby z takich typów jak listy lub tuple.
Brak duplikatów	Duplikaty są zabronione (każdy element zbioru jest unikatowy).
Zakaz modyfikacji elementów	Raz wprowadzone do zbioru dane są w nim niemodyfikowalne.
Rozszerzalność	Po wstępnej inicjalizacji zbioru można do niego dokładać nowe elementy. Liczbę elementów zawartych w zbiorze możesz odczytać, stosując funkcję `len()`.
Dowolna zawartość	Zbiory mogą zawierać listę wartości dowolnych typów pod warunkiem, że są one niemodyfikowalne: mogą to być znaki, liczby, napisy, tuple, wartości logiczne `True` lub `False`, ale listy i słowniki już nie.
Symbol rozpoznawczy	Nawiasy klamrowe: {}.

```
print ("Budowanie zbioru na podstawie listy:", lista1)
print ("Lista 'lista1'=", lista1)
# Alternatywna metoda budowania zbiorów z użyciem konstruktora 'set' załadowanego listą elementów
zbior2=set(lista1)
print ("Zbiór 'zbior2' utworzony z 'lista1'=", zbior2)

print ("Zbiory zawierające elementy różnych typów:")
kodHEX={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'A', 'B', 'C', 'D', 'E', 'F'}
print ("Dozwolone znaki kodu szesnastkowego:\n", kodHEX)
```

Jeśli uruchomisz skrypt, to pokazany wyżej fragment kodu wyświetli następujący wynik:

```
Zbiór:  {'komar', 'insekt', 'żaba'}
Spróbujmy dodać do zbiornika kolejną żabę...
Aktualny stan zbiornika:
Zbiór:  {'komar', 'insekt', 'żaba'}
Budowanie zbioru na podstawie listy:
Lista 'lista1'= [2, 3, 4, 5, 6, 6, 6, 9, 9, 0]
Zbiór 'zbior2'= {0, 2, 3, 4, 5, 6, 9}
Zbiory zawierające elementy różnych typów:
Dozwolone znaki kodu szesnastkowego:
 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'D', 'E', 'B', 'F', 'C', 'A'}
```

Liczebność zbioru możesz sprawdzić, aplikując na nim znaną Ci już funkcję `len()`. Ale to nie koniec dostępnych możliwości. Python oferuje szereg metod pozwalających operować typem set (dodawanie, usuwanie elementów, wyszukiwanie itp.) — najpopularniejsze znajdziesz w tabeli 8.5.

(Rozszerzone fragmenty przykładowego kodu podanego w tej tabeli znajdziesz w pliku *zbiory.py*).

TABELA 8.5. Metody dostępne dla zbiorów w Pythonie

CECHA	KOMENTARZ
`add(x)`	Dodaje element x do zbioru.
`pop()`	Usuwa losowy element ze zbioru i zwraca go jako wynik wywołania tej metody. Jeśli zbiór okaże się pusty, próba użycia `pop()` zgłosi wyjątek `KeyError`. Przykład: `zb={5, 7, 8, 7, 7}` *# Po inicjacji kodu ten zbiór będzie zawierał* *# tylko 8, 5 i 7* `print (zb)` *# Wypisze: {8, 5, 7}* `symbol=zb.pop()` `print(symbol)` *# Wypisze: 3*
`Remove(x)`	Usuwa element x ze zbioru. Przykład: `x={5, 7, 8, 7, 7}` *# Zbiór będzie zawierał tylko 8, 5 i 7* `zb.remove(7)` Jeśli zbiór nie zawiera x, to próba użycia `remove(x)` zgłosi wyjątek `KeyError`.
`clear()`	Kasuje (czyści) zawartość zbioru.

`union(...)` lub operator |	**Suma** zbiorów (podczas sumowania powtórzone elementy są uwzględniane tylko raz). Przykład:
	`zb1={5, 6, 7}`
	`zb2={6, 7, 8, 9}`
	`zb3=zb1.union(zb2)`
	`print(zb1|zb2)` *# Wypisze: {5, 6, 7, 8, 9}*
	`print(zb3)` *# Wypisze: {5, 6, 7, 8, 9} – jest to analogiczna operacja*
	Jako parametry metody `union` możesz podać więcej zbiorów, oddzielając je przecinkami (podobnie możesz napisać: `zbior1|zbior2|zbior3` itp.). W takim przypadku Python będzie wykonywał działanie od lewej do prawej.

`intersection (...)` lub operator &	**Przecięcie** zbiorów (zbiór wynikowy będzie zawierał elementy obecne we wszystkich zbiorach jednocześnie). Przykład:
	`zb1={2, 4, 6, 8}`
	`zb2={1, 4, 6, 10}`
	`zb3={0, 4, 6, 11}`
	`print(zb1&zb2&zb3)` *# Wypisze: {4, 6}*
	`print(zb1.intersection(zb2, zb3))` *# Wypisze: {4, 6} – jest to*
	# analogiczna operacja

`difference (...)` lub operator -	**Różnica** zbiorów (zbiór wynikowy będzie zawierał elementy obecne w danym zbiorze i nieobecne w pozostałych). Przykład:
	`zb1={2, 4, 6, 8}`
	`zb2={1, 4, 6, 10}`
	`print(zb1-zb2)` *# Wypisze: {8, 2}*
	`print(zb1.difference(zb2))` *# Wypisze: {8, 2} – jest to analogiczna operacja*
	Jako parametry metody `difference` możesz podać więcej zbiorów, oddzielając je przecinkami (podobnie możesz napisać: `zbior1-zbior2-zbior3` itp.). W takim przypadku Python będzie wykonywał działanie od lewej do prawej.

TABELA 8.5. Metody dostępne dla zbiorów w Pythonie — *ciąg dalszy*

CECHA	KOMENTARZ

`symmetric_difference (...)` lub operator ^	**Różnica** symetryczna zbiorów `zb1` i `zb2` zwraca elementy obecne w `zb1` lub w `zb2`, ale nie w obu jednocześnie (nie w ich przecięciu)
	`zb1={2, 4, 6, 8}`
	`zb2={1, 4, 6, 10}`
	`print(zb1.symmetric_difference(zb2))` *# Wypisze: {1, 2, 8, 10}*
	`print(zb1^zb2)` *# Wypisze: {1, 2, 8, 10}*

`issubset()`	Czy dany zbiór jest podzbiorem drugiego?
	`zb1={2, 4}`
	`zb2={5, 4, 2, 10}`
	`print(zb1.issubset(zb2))` *# Wypisze: True*

`issuperset()`	Czy dany zbiór zawiera ten drugi (czy jest jego tzw. nadzbiorem)?
	`zb1={2, 4, 6, 8}`
	`zb2={4, 6, 8}`
	`print(zb1.issuperset(zb2))` *# Wypisze: True*

Zbiory nie są zbyt często używane przez programistów, co może wydawać się dziwne. Istnieje bowiem szereg problemów analizy danych, w których ten typ danych pasuje idealnie, a realizacje z użyciem list prowadziłyby do dość karkołomnych i nieczytelnych skryptów.

Przykład

Wyobraź sobie, że porównujesz wyniki serii testów przeprowadzonych na pewnym systemie opartym na komunikacji HTTP. Jak wiadomo, podczas komunikacji klient-serwer mogą wystąpić liczne błędy i nasze oprogramowanie może logować zwracane kody. Przykłady kilku takich kodów:[2]

- 400 Bad Request (pol. *nieprawidłowe zapytanie*),
- 401 Unauthorized (pol. *nieautoryzowany dostęp*),
- 404 Not Found (pol. *nie znaleziono* — serwer nie znalazł zasobu według podanego URL)
- i wiele innych.

Okazuje się, że jeśli zapiszesz kody błędów w zbiorach, a nie listach, to łatwo dokonasz korelacji wyników i sprawdzisz, czy pewne serie testów nie wykrywają podobnych problemów.

Poniżej znajdziesz szkielet kodu realizującego taką analizę, opartego na następujących założeniach:

[2] Pełną listę znajdziesz w specyfikacji RFC lub chociażby w artykule: *https://pl.wikipedia.org/wiki/Kod_odpowiedzi_HTTP*.

- Analizujemy wyniki z trzech serii testów.
- Dla każdej serii wczytamy wszystkie odebrane kody błędów do roboczych list Pythona, akceptując ewentualne duplikaty. Pochodzenie tych kodów na tym etapie ignoruję; mogły zostać pobrane z pliku, bazy danych lub mogły być rejestrowane na bieżąco jako odpowiedzi testowanej aplikacji.
- Następnie zawartość takich roboczych list, zawierających surowe dane, posłuży do utworzenia zbiorów kodów błędów — a wiemy, że zbiory nie akceptują duplikatów!
- Oczyszczone, dzięki usunięciu duplikatów, zbiory wynikowe poddamy porównaniom, wykorzystując właściwości zbiorów i dostępne dla nich metody.
- Popatrz, jak nieskomplikowany jest kod Pythona realizujący te założenia (*analizaWWW.py*):

```
# Tabela zawierająca otrzymane kody błędów z trzech serii testowania
wyniki_surowe=list()
wyniki_surowe.append(list())  # Seria 1.
wyniki_surowe.append(list())  # Seria 2.
wyniki_surowe.append(list())  # Seria 3.
wyniki_surowe[0]=["400", "401", "401", "401", "401", "410", "410", "425", "400",
                  "429",  "431", "431", "400", "431","413", "414", "425", "401",
                  "410", "410",  "401", "408", "408", "408", "400", "400", "400"]
wyniki_surowe[1]=["408", "408", "408", "400", "401", "401", "410", "425", "400",
                  "429", "431", "431", "415", "408","408", "400", "425", "401",
                  "410", "410","401", "408", "408", "408", "422", "400", "400"]
wyniki_surowe[2]=["400", "400", "401", "401", "401", "400", "400", "425", "400"]
# Sprawdźmy zawartość
print("Wszystkie wyniki serii 1.:", wyniki_surowe[0])
print("Wszystkie wyniki serii 2.:", wyniki_surowe[1])
print("Wszystkie wyniki serii 3.:", wyniki_surowe[2])

# Tabela zawierająca otrzymane kody błędów z trzech serii testowania przekształcone na zbiory
wyniki_zbiory=list()
wyniki_zbiory.append(set(wyniki_surowe[0])) # Seria 1. przekształcona do postaci zbioru
wyniki_zbiory.append(set(wyniki_surowe[1])) # Seria 2. przekształcona do postaci zbioru
wyniki_zbiory.append(set(wyniki_surowe[2])) # Seria 3. przekształcona do postaci zbioru

print("Znormalizowana lista wyników (bez duplikatów)")
print("  Seria 1.:", wyniki_zbiory[0])
print("  Seria 2.:", wyniki_zbiory[1])
print("  Seria 3.:", wyniki_zbiory[2])

# A teraz analiza danych:
print("Lista wszystkich wykrytych błędów:")
print(" ", wyniki_zbiory[0] |  wyniki_zbiory[1] | wyniki_zbiory[2]) #(*)
print("To same kody błędów wykryte w każdej z serii:")
print(" ", wyniki_zbiory[0] &  wyniki_zbiory[1] & wyniki_zbiory[2]) #(**)
```

W kodzie programu wpisałem serię wyników na tyle długą, aby nie dało się jej zbyt łatwo przeanalizować za pomocą popularnej inżynierskiej metody zwanej na oko... Dzięki temu możesz docenić, jak ogromne możliwości daje jej transformacja do postaci zbiorów zawierających wyłącznie unikatowe wartości