Ponadto użycie dwóch prostych instrukcji — złączenie (linijka oznaczona *) i przecięcie (linijka oznaczona **) ukazuje wyniki, do których na pewno nie sposób szybko dojść, po prostu patrząc na surowe serie danych!

Program wyświetli wynik (wyróżnienia są wprowadzone sztucznie):

```
Wszystkie wyniki serii 1.: ['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']
Wszystkie wyniki serii 2.: ['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']
Wszystkie wyniki serii 3.: ['400', '400', '401', '401', '401', '400', '400', '425',
↪'400']
Znormalizowana lista wyników (bez duplikatów)
  Seria 1.: {'431', '408', '401', '410', '414', '400', '413', '425', '429'}
  Seria 2.: {'431', '415', '408', '422', '401', '410', '400', '425', '429'}
  Seria 3.: {'400', '401', '425'}
Lista wszystkich wykrytych błędów:
  {'414', '413', '410', '422', '400', '425', '408', '415', '401', '431', '429'}
Te same kody błędów wykryte w każdej z serii:
  {'400', '401', '425'}
```

Mam nadzieję, że podane w tym rozdziale informacje zachęcą Cię do używania tych struktur danych, gdyż pozwalają one wyjątkowo łatwo rozwiązywać niektóre problemy algorytmiczne, i to na dodatek bez pisania setek linii kodu!

Zbiory tworzone na podstawie wyrażeń

Podobnie jak to jest możliwe dla list, Python oferuje też inicjalizację zbioru na podstawie wyrażenia zawartego w nawiasie. W literaturze anglojęzycznej ten mechanizm określany jest terminem *set comprehension*. Popatrz na prosty przykład kodu, który ilustruje ten mechanizm:

```
x1 = [-4, -2, -1, 0, 1, 2, 4]
x2 = { n*n for n in x1 } # Wynik: x2 = {16, 1, 4, 0}
```

W pierwszym wierszu deklarujemy listę x1 zawierającą siedem wartości, w drugim tworzymy nowy *zbiór*, złożony z wartości będących kwadratami wartości pobranych z listy x1. Ponieważ tworzymy zbiór, Python nie pozwoli na tworzenie duplikatów, stąd mniejszy rozmiar tego zbioru wyjściowego (tylko cztery elementy).

Słowniki

Omówmy teraz interesującą strukturę danych zwaną słownikiem, która... nie ma nic wspólnego ze swoim odpowiednikiem ze świata realnego. W księgarni słownik jest zazwyczaj dość opasłą publikacją zawierającą zbiór haseł oraz odpowiadających im definicji. Hasła są często posortowane alfabetycznie, a definicje przybierają niekiedy dość wyrafinowane formy (opis główny, odsyłacze, hasła podrzędne).

W Pythonie jest nieco prościej, słownikiem jest lista par *<klucz>:<wartość>* (tak, elementy takiej pary są rozdzielone dwukropkiem) i nie ma możliwości uzyskania tak eleganckich form, jak np. hasła ze *Słownika mitów i tradycji kultury* Władysława Kopalińskiego czy z klasycznego słownika języka obcego.

Przyjrzyj się przykładowej deklaracji słownika w Pythonie [*slowniki0.py* (fragment)]:

```
doctorsWho={
      "Pierwszy": "William Hartnell",
      "2":        "Patrick Troughton",
      "3":        "Jon Pertwee",
      "4":        "Tom Baker",
      "5":        "Peter Davison",
      "6":        "Colin Baker",
      "7":        "Sylvester McCoy",
      "8":        "Paul McGann",
      "9":        "Christopher Eccleston",
       10:        "David Tennant",
      "11":       "Matt Smith",
      "12":       "Peter Capaldi",
      "Trzynasta": "Jodie Whittaker"}
print ("Kowalski" in doctorsWho)     # False (niestety!)
for key in doctorsWho:
    print(f" {key}  -   {doctorsWho[key]}")
```

Pierwsze, co się rzuca w oczy, to brak jakichkolwiek ograniczeń dotyczących wartości lub formatu klucza. Może to być napis (np. „Pierwszy"), a nawet liczba (klucz 10.)!

Elementy słownika (wartość przypisaną do klucza) wyłuskujemy, używając operatora dostępu [] (czyli nawiasów klamrowych), np.:

```
print(doctorsWho[10])              # Wypisze: David Tennant
print(doctorsWho["Trzynasta"])     # Wypisze: Jodie Whittaker
```

co jest pewną nowością w porównaniu z listami, gdzie znaczenie miał numer porządkowy (indeks).

Jeśli uruchomisz skrypt *slowniki0.py*, to pokazany wyżej fragment kodu wyświetli następujący wynik:

```
Pierwszy  -   William Hartnell
 2  -   Patrick Troughton
 3  -   Jon Pertwee
 4  -   Tom Baker
 5  -   Peter Davison
 6  -   Colin Baker
 7  -   Sylvester McCoy
 8  -   Paul McGann
 9  -   Christopher Eccleston
 10  -   David Tennant
 11  -   Matt Smith
 12  -   Peter Capaldi
Trzynasta  -   Jodie Whittaker
```

Zapoznaj się z tabelą 8.6, podsumowującą cechy wyróżniające słownikowy typ danych w Pythonie.

TABELA 8.6. Słowniki w Pythonie — cechy charakterystyczne

CECHA	KOMENTARZ
Uporządkowanie	Dane w słowniku w najnowszych odsłonach Pythona są uporządkowane w tym sensie, że nowe wpisy są dokładane na koniec, ale nie oznacza to, że można do nich sięgnąć przez indeks znany z list. Jedyną metodą umożliwiającą pobranie elementu jest zaadresowanie go za pomocą tzw. klucza (patrz niżej).
Duplikaty kluczy są zabronione	Klucze muszą być unikatowe, **ale możemy wpisać kilka takich samych wartości dla różnych kluczy**. Uwaga: Python pozwoli jednak na zadeklarowanie słownika zawierającego pozornie zduplikowane klucze, np.: `slownik={"k1":5, "k2":6, "k2":66, "k3":7}.` Okazuje się jednak, że końcowa zawartość takiego słownika będzie wyglądała tak: `{'k1':5, 'k2':66, 'k3':7}` — klucz k2 pozostał unikatowy, ale pierwotna wartość *6* została nadpisana przez *66*.
Modyfikowalność	Do słownika można swobodnie dokładać nowe elementy oraz modyfikować istniejące wpisy.
Dowolna zawartość	Słowniki zawierają pary `<klucz>:<wartość>` rozdzielone znakiem dwukropka. *Klucze* muszą umożliwiać efektywne adresowanie, co *de facto* ogranicza ich typ do liczb lub napisów. *Wartości* mogą być dość dowolne: liczby, napisy **oraz typy złożone** (np. listy).
Symbol rozpoznawczy	Nawiasy klamrowe: {} oraz : (dwukropek) rozdzielający pary `<klucz>:<wartość>`.

Jakie operacje są wykonywane na słownikach? W tabeli 8.7 podsumowałem dostępne operacje.

Ponieważ operator dostępu [] wymaga podania poprawnego klucza, to jeśli chcesz uniknąć obsługi wyjątku `KeyError`, musisz badać wcześniej lub być pewny obecności klucza w słowniku. Można też zastosować specjalną metodę `get()` badającą obecność klucza:

```
test = [ {'400': 'Nieprawidłowe zapytanie'},
         {'405': 'Niedozwolona metoda'}, {'500': 'Wewnętrzny błąd serwera'}]
print(test[0]['400']) # Wypisze: Nieprawidłowe zapytanie
print(test[0]['401']) # Zwróci wyjątek: KeyError!

# Sprawdźmy inaczej, czy w tabeli test znajduje się rekord o kluczu '401':

klucz='401'
```

TABELA 8.7. Podstawowe operacje wykonywane na słownikach w Pythonie

CECHA	KOMENTARZ
`[]` Operator dostępu	Operator dostępu `[klucz]` pozwala uzyskać dostęp wartości wskazywanej przez `klucz`. Element musi jednak istnieć, w przeciwnym razie Python zgłosi wyjątek `KeyError`.
`len(x)`	Znana, klasyczna funkcja zwracająca długość kolekcji (tutaj: listę kluczy).
`in`	Słowo kluczowe `in` pozwala na sprawdzenie istnienia klucza w słowniku. Przykład operujący znanym nam już słownikiem `doctorsWho`: `print ("Kowalski" in doctorsWho)` wypisze `False`.
`del`	Polecenie wykasowania wpisu ze słownika. Przykład: `del doctorsWho["Pierwszy"]` Element musi jednak istnieć, w przeciwnym razie Python zgłosi wyjątek `KeyError`.

```
for x in test:
    if x.get(klucz)!=None:
        print(f"Znalazłem rekord dla klucza o wartości {klucz}'")
    else:
        print(f"Brak danych dla klucza o wartości {klucz}")
```

Lista metod dostępnych dla słowników w Pythonie jest dość krótka (tabela 8.8).

Słowniki pozwalają na efektywne implementowanie małych baz danych lub swego rodzaju map symboli, których można używać w programach.

Pamiętasz przykład zawarty w pliku *analizaWWW.py*? Okazuje się, że można go łatwo wzbogacić o dodatkowy słownik tłumaczący kody błędów i prezentujący wyniki w sposób znacznie bardziej przyjazny dla odbiorcy (*analizaWWW2.py*):

```
# Skrypt niemal identyczny z analizaWWW.py, ale wzbogacony m.in. o:
słownikBłędów={
    '400' : 'Bad Request'
    # '401': 'Unauthorized', # Świadomie pominięty!
    '408' : 'Request Timeout',
    '410' : 'Gone',
    '413' : 'Request Entity Too Large',
    '415' : 'Unsupported Media Type',
    '414' : 'Request-URI Too Long',
    '422' : 'Unprocessable entity',
    '425' : 'Too Early',
    '429' : 'Too Many Requests',
    '431' : 'Request Header Fields Too Large'}
# Analiza danych (w tym miejscu wprowadzono drobne zmiany w porównaniu z pierwotnym kodem
# znanym z analizaWWW.py):
print("Lista wszystkich wykrytych błędów:")
wszystkie=wyniki_zbiory[0] |  wyniki_zbiory[1] | wyniki_zbiory[2]
```

TABELA 8.8. Wybrane metody dostępne dla słowników w Pythonie

CECHA	KOMENTARZ
`update( {x:y} )`	Dodaje parę `x:y` do słownika lub aktualizuje wartość dla klucza x, jeśli wcześniej już istniał taki klucz. Przykład dla podanego wcześniej słownika `doctorsWho`: `doctorsWho.update( { 15:"Jan Kowalski" } )` *# Nowy wpis* `doctorsWho.update( {"7":"Sylwek McCoy" } )` *# Aktualizacja* Przykład: `(lub: doctorsWho["7"]=" Sylwek McCoy"`
`clear()`	Kasuje (czyści) zawartość słownika.
`get(x)` Opcjonalny parametr: `default` Uwaga: jest to odpowiednik operatora dostępu []	Zwraca *wartość* skojarzaną z kluczem x. Jeśli dla klucza x wartość nie istnieje, to zwróci `None` lub ewentualnie pewien jawnie podany parametr `default`. Przykład: `print(doctorsWho.get(10))` *# Lub: print(doctorsWho[10])* Poniższa instrukcja wypisze: „Nie ma takiej serii Doktor Who": `print(doctorsWho.get(15, "Nie ma takiej serii Doktor Who"))`
`pop(x)`	Usuwa ze słownika wpis dla klucza x. Przykład: `doctorsWho.pop("Pierwszy")` *# Lub: del doctorsWho["Pierwszy"]* Element musi jednak istnieć, w przeciwnym razie Python zgłosi wyjątek `KeyError`.
`keys()`	Zwraca listę *kluczy* zapisanych w słowniku.
`values()`	Zwraca listę *wartości* zapisanych w słowniku.

```
for kod in wszystkie:
    print(kod, " ", slownikBledow.get(kod, "Nieznany kod błędu!"))
print("Te same kody błędów wykryte w każdej z serii:")

wspolne=wyniki_zbiory[0] &  wyniki_zbiory[1] & wyniki_zbiory[2]

for kod in wspolne:
    print(kod, " ", slownikBledow.get(kod, "Nieznany kod błędu!")) #(*)
```

Spójrz, jak przyjaźnie dla odbiorcy prezentują się teraz wyniki analizy wykonań serii testów:

```
Lista wszystkich wykrytych błędów:
415   Unsupported Media Type
410   Gone
425   Too Early
401   Nieznany kod błędu!
429   Too Many Requests
431   Request Header Fields Too Large
414   Request-URI Too Long
422   Unprocessable entity
408   Request Timeout
413   Request Entity Too Large
400   Bad Request
Te same kody błędów wykryte w każdej z serii:
401   Nieznany kod błędu!
400   Bad Request
425   Too Early
```


Zwróć uwagę na wyróżniony wiersz — jest to efekt użycia w linii (*), w metodzie `get()` parametru domyślnego, którym jest napis „Nieznany kod błędu" zwracany w przypadku brakującego wpisu w słowniku.

Słowniki po prostu są bardzo przydatne i wzbogacają możliwości pisania eleganckiego kodu!

Na koniec proponuję przeanalizowanie i samodzielne przetestowanie prostego, interaktywnego programu opartego na strukturze słownikowej służącej do zapamiętywania listy telefonów pracowników pewnego przedsiębiorstwa. Jest to oczywiście niezbyt profesjonalna baza danych (informacje nie są przecież zapisywane w sposób permanentny), ale możesz na tej podstawie poznać techniki tworzenia i modyfikowania słowników w Pythonie (*slowniki.py*).

```
pracownicy = {
    "Jan Kowalski": 668168555,
    "Anna Zwinna": 605123001,
    "Marek Ekspercki": 721003050,
    "Jan Bęcki": 672000455}
for imie, numer in pracownicy.items():      # Wypisanie zawartości słownika
    print(f"Pracownik: {imie}, telefon:\t{numer}")
print(pracownicy) # Tak też można wypisać zawartość słownika
nazwisko = input("Podaj nazwisko osoby: ") # Sprawdzanie obecności klucza w słowniku
if nazwisko in pracownicy:
    print(f"Znalazłem {nazwisko} w bazie danych!")
else:
    print(f"Nie znalazłem {nazwisko} w bazie danych!")
# Odczyt danych ze słownika. Przypadek: istnieje para klucz-wartość
print("klucz: Jan Bęcki, wartość:", pracownicy.get("Jan Bęcki"))
# Odczyt danych ze słownika. Przypadek: NIE istnieje para klucz-wartość
print("klucz: Janek Bęcki, wartość:", pracownicy.get("Janek Bęcki"))
# Zwraca: None. Co z tym fantem robić? W podany niżej sposób możesz uniknąć zwracania pustej
# wartości (None) i wypisać np. „Nie znaleziono wpisu!":
print("klucz: Janek Bęcki, wartość:", pracownicy.get("Janek Bęcki", "Nie znaleziono
➥wpisu!"))
# Dodanie lub modyfikowanie wpisów w słowniku
print("Rozszerzanie słownika o nowe wpisy (Piotr Wróblewski, tel. 668999550):")
pracownicy["Piotr Wróblewski"]=668999550
print(pracownicy)
print("Aktualizacja zawartości słownika dla wcześniej użytego klucza (Piotr
➥Wróblewski, nowy tel. 668888550 ")
pracownicy["Piotr Wróblewski"]=668888550
print(pracownicy)
# Usunięcie elementu ze słownika
nazwisko = input("Podaj nazwisko osoby do usunięcia: ")
if nazwisko in pracownicy:
    print(f"Znalazłem \'{nazwisko}\' w słowniku, usuwam!!!")
    pracownicy.pop(nazwisko)
else:
    print(f"Nie znalazłem \'{nazwisko}\' w słowniku")
print ("Finalna zawartość listy pracowników:")
```