Poniższy rozdział pochodzi z książki:

Charles R. Severance - "Python dla wszystkich: Odkrywanie danych z Python 3"

Pełna wersja podręcznika znajduje się na stronie https://py4e.pl/book

Rozdział 9

Słowniki

Słownik jest podobny do listy, ale bardziej ogólny. W liście indeksy muszą być liczbami całkowitymi; w słowniku indeksy mogą być (prawie) dowolnego typu.

Możesz myśleć o słowniku jako o mapowaniu pomiędzy zestawem indeksów (które są nazywane *kluczami*) a zestawem wartości. Każdy klucz mapuje się do jakiejś wartości. Skojarzenie klucza i wartości nazywane jest parą *klucz-wartość* lub czasem *elementem*.

Jako przykład zbudujemy słownik, który mapuje słowa angielskie na hiszpańskie, więc klucze i wartości będą napisami.

Funkcja `dict()` tworzy nowy słownik bez żadnych elementów. Ponieważ `dict()` jest nazwą wbudowanej funkcji, powinieneś unikać używania jej jako nazwy zmiennej.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

Nawiasy klamrowe, `{}`, reprezentują pusty słownik. Aby dodać elementy do słownika, możesz użyć nawiasów kwadratowych:

```
>>> eng2sp['one'] = 'uno'
```

Powyższa linia tworzy element, który mapuje klucz `'one'` do wartości `'uno'`. Jeśli ponownie wyświetlimy słownik, to zobaczymy parę klucz-wartość z dwukropkiem pomiędzy kluczem a wartością:

```
>>> print(eng2sp)
{'one': 'uno'}
```

Widoczny powyżej format wyjściowy jest również formatem wejściowym. Na przykład możesz utworzyć nowy słownik z trzema elementami:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp)
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

W Pythonie od wersji 3.6 kolejność par klucz-wartość jest zgodna kolejnością ich wstawienia do słownika (w poprzednich wersjach Pythona kolejność pozycji była nieprzewidywalna).

Pomimo tej nowej własności, tradycyjnie do wyszukiwania elementów słownika używamy kluczy:

```
>>> print(eng2sp['two'])
'dos'
```

Klucz `'two'` zawsze mapuje się do wartości `'dos'`.

Jeśli klucza nie ma w słowniku, otrzymasz wyjątek:

```
>>> print(eng2sp['four'])
KeyError: 'four'
```

Funkcja `len()` działa również na słownikach i zwraca liczbę par klucz-wartość:

```
>>> len(eng2sp)
3
```

Operator `in` też działa na słownikach i informuje Cię, czy coś pojawia się w słowniku jako *klucz* (wystąpienie jako wartość nie jest wystarczające).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

Aby sprawdzić, czy coś pojawia się w słowniku jako wartość, możesz użyć metody `values()`, zwracającą wartości jako typ, który może być skonwertowany na listę, a następnie użyć operatora `in`:

```
>>> vals = list(eng2sp.values())
>>> 'uno' in vals
True
```

Operator `in` używa różnych algorytmów dla list i słowników. W przypadku list, używa algorytmu wyszukiwania liniowego. Gdy lista staje się dłuższa, czas wyszukiwania wydłuża się wprost proporcjonalnie do jej długości. W przypadku słowników Python używa algorytmu zwanego *tablicą mieszającą* lub *tablicą z haszowaniem*, który ma niezwykłą właściwość: operator `in` zajmuje mniej więcej tyle samo czasu, niezależnie od tego, ile pozycji jest w słowniku. Nie będę tłumaczył, dlaczego funkcje haszujące są tak magiczne, ale możesz przeczytać o tym więcej na stronie https://pl.wikipedia.org/wiki/Tablica_mieszaj%C4%85ca.

Ćwiczenie 1. Pobierz kopię pliku https://py4e.pl/code3/words.txt

Napisz program, który odczytuje słowa z `words.txt` i przechowuje je jako klucze w słowniku. Nie ma znaczenia, jakie będą wartości w słowniku. Następnie możesz użyć operatora `in` jako szybkiego sposobu na sprawdzenie, czy dany wyraz znajduje się w słowniku.

9.1. Słownik jako zbiór liczników

Załóżmy, że otrzymałeś napis zawierający angielski wyraz i chcesz policzyć, ile razy każda litera się w nim pojawiła. Możesz to zrobić na kilka sposobów:

1. Mógłbyś utworzyć 26 zmiennych, po jednej dla każdej litery alfabetu. Następnie mógłbyś przejść po napisie i dla każdego znaku zwiększyć odpowiedni licznik, prawdopodobnie używając połączonych wyrażeń warunkowych.
2. Mógłbyś utworzyć listę z 26 elementami. Następnie mógłbyś przekonwertować każdy znak na liczbę (używając wbudowanej funkcji `ord()`), użyć liczby jako indeksu do listy i zwiększyć odpowiedni licznik.
3. Mógłbyś utworzyć słownik, w którym znaki byłyby kluczami, a liczniki odpowiednimi wartościami. Za pierwszym razem, gdy natrafisz na znak, dodasz element do słownika. Następnie zwiększałbyś wartość istniejącej wartości.

Każda z tych opcji wykonuje to samo obliczenie, ale każda z nich implementuje to w inny sposób.

Implementacja to sposób przeprowadzenia obliczeń; niektóre implementacje są lepsze od innych. Na przykład zaletą implementacji poprzez słownik jest to, że nie musimy z góry wiedzieć, które litery pojawiają się w napisie, a ponadto musimy tylko zrobić miejsce na te, które faktycznie się pojawiają.

Oto jak może wyglądać kod:

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print(d)
```

W praktyce obliczamy *histogram*, który jest statystycznym terminem określającym zestaw liczników (lub częstości).

Pętla `for` przechodzi po napisie. Za każdym razem gdy przechodzimy przez pętlę, jeśli znak zawarty w `c` nie występuje w słowniku, tworzymy nową pozycję z kluczem `c` i wartością początkową 1 (ponieważ widzimy tę literę pierwszy raz). Jeśli `c` znajduje się już w słowniku, zwiększamy wartość `d[c]`.

Oto wynik programu:

```
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
```

Histogram wskazuje, że np. litery "a" i "b" pojawiają się raz; "o" pojawia się dwa razy itd.

Słowniki posiadają metodę `get()`, która przyjmuje klucz i domyślną wartość. Jeśli klucz pojawia się w słowniku, `get()` zwraca odpowiednią wartość; w przeciwnym razie zwraca wartość domyślną. Na przykład:

```
>>> counts = { 'chuck' : 1, 'annie' : 42, 'jan' : 100 }
>>> print(counts.get('jan', 0))
100
>>> print(counts.get('tim', 0))
0
```

Możemy użyć `get()` do bardziej zwięzłego napisania naszej pętli z histogramem. Ponieważ metoda `get()` automatycznie zajmuje się przypadkiem, gdy danego klucza nie ma w słowniku, możemy zredukować cztery linie do jednej i wyeliminować instrukcję `if`.

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c, 0) + 1
print(d)
```

Użycie metody `get()` do uproszczenia tej pętli zliczania kończy się bardzo często używanym "idiomem" w Pythonie i będziemy go używać wiele razy w pozostałej części książki. Powinieneś poświęcić chwilę na porównanie pętli przy użyciu instrukcji `if` i operatora `in` z pętlą przy użyciu metody `get()`. Robią one dokładnie to samo, ale drugi sposób jest bardziej zwięzły.

9.2. Słowniki i pliki

Jednym z częstych zastosowań słownika jest zliczanie występowania słów w pliku zawierającym jakiś tekst. Zacznijmy od bardzo prostego pliku zawierającego słowa wzięte z dramatu *Romeo i Julia* (użyjemy tekstu w wersji angielskiej).

Do pierwszego zestawu przykładów wykorzystamy skróconą i uproszczoną wersję tekstu bez interpunkcji. Później będziemy pracować z tekstem zawierającym interpunkcję.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Napiszemy w Pythonie program, który odczyta wiersze pliku, rozbijemy każdy wiersz na listę słów, a następnie przejdziemy w pętli przez każde słowo zawarte w linii i za pomocą słownika policzymy wystąpienia każdego słowa.

Za chwilę zobaczysz, że mamy dwie pętle `for`. Pętla zewnętrzna odczytuje wiersze pliku, a pętla wewnętrzna iteruje przez każde ze słów w danym wierszu. Jest to przykład schematu zwanego *pętlą zagnieżdżoną*, ponieważ jedna z pętli jest pętlą *zewnętrzną*, a druga – pętlą *wewnętrzną*.

Pętla wewnętrzna wykonuje wszystkie swoje iteracje za każdym razem, gdy pętla zewnętrzna wykonuje jedną iterację. W związku z tym myślimy o pętli wewnętrznej jako iterującej "szybciej", a o pętli zewnętrznej jako iterującej wolniej.

Połączenie dwóch zagnieżdżonych pętli zapewnia, że będziemy zliczać każde słowo w każdym wierszu pliku wejściowego.

```
fname = input('Podaj nazwę pliku: ')
try:
    fhand = open(fname)
except:
    print('Nie można otworzyć pliku:', fname)
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Kod źródłowy: https://py4e.pl/code3/count1.py
```

W naszej instrukcji `else` używamy zwięzłej alternatywy dla inkrementacji zmiennej. `counts[word] += 1` jest odpowiednikiem `counts[word] = counts[word] + 1`. Każdej z tych metod można użyć do zmiany wartości zmiennej o dowolną pożądaną wielkość. Podobne alternatywy istnieją dla `-=`, `*=` i `/=`.

Kiedy uruchomimy program, zobaczymy surowy zrzut wszystkich zliczeń w tablicy haszującej w kolejności wstawiania nowych słów. Poniżej widzimy uruchomienie programu na pliku `romeo.txt`, który jest dostępny pod adresem https://py4e.pl/code3/romeo.txt.

```
Podaj nazwę pliku: romeo.txt
{'But': 1, 'soft': 1, 'what': 1, 'light': 1, 'through': 1,
 'yonder': 1, 'window': 1, 'breaks': 1, 'It': 1, 'is': 3,
 'the': 3, 'east': 1, 'and': 3, 'Juliet': 1, 'sun': 2,
 'Arise': 1, 'fair': 1, 'kill': 1, 'envious': 1, 'moon': 1,
 'Who': 1, 'already': 1, 'sick': 1, 'pale': 1, 'with': 1,
 'grief': 1}
```

Przeglądanie słownika w celu znalezienia najczęściej używanych słów i ich liczby wystąpień jest dość niewygodne. Aby to poprawić, będziemy musieli dodać trochę więcej kodu Pythona.

9.3. Pętle i słowniki

Jeśli używasz w instrukcji `for` słownika jako sekwencji, pętla przechodzi wtedy przez klucze słownika. Poniższa pętla wyświetla każdy klucz i odpowiadającą mu wartość:

```
counts = { 'chuck' : 1, 'annie' : 42, 'jan' : 100 }
for key in counts:
```

```
    print(key, counts[key])
```

Oto jak wygląda wynik działania programu:

```
chuck 1
annie 42
jan 100
```

Możemy użyć tego schematu do zaimplementowania różnych idiomów pętli, które opisaliśmy wcześniej. Na przykład, jeśli chcielibyśmy znaleźć wszystkie wpisy w słowniku o wartości powyżej dziesięciu, moglibyśmy napisać następujący kod:

```
counts = { 'chuck' : 1, 'annie' : 42, 'jan' : 100 }
for key in counts:
    if counts[key] > 10 :
        print(key, counts[key])
```

Pętla `for` iteruje przez *klucze* słownika, więc dla każdego klucza musimy użyć operatora indeksu w celu pobrania odpowiadającej mu *wartości*. Poniżej mamy wynik działania programu:

```
annie 42
jan 100
```

Widzimy teraz tylko te elementy, które mają wartość powyżej 10.

Jeżeli chcesz wyświetlić klucze w porządku alfabetycznym, to – korzystając z metody `keys()` dostępnej w obiektach słownikowych – sporządzasz listę kluczy występujących w słowniku, a następnie sortujesz ją i przechodzisz w pętli po tej posortowanej liście, przeglądając każdy klucz i wyświetlając pary klucz-wartość w posortowanej kolejności, tak jak pokazano poniżej:

```
counts = { 'chuck' : 1, 'annie' : 42, 'jan' : 100 }
lst = list(counts.keys())
print(lst)
lst.sort()
for key in lst:
    print(key, counts[key])
```

Oto jak wygląda wynik programu:

```
['chuck', 'annie', 'jan']
annie 42
chuck 1
jan 100
```

Najpierw widzisz listę kluczy w nieposortowanej kolejności, które otrzymujemy z metody `keys()`. Następnie widzimy uporządkowane pary klucz-wartość, wyświetlane w pętli `for`.

9.4. Zaawansowane parsowanie tekstu

W powyższym przykładzie, używając pliku `romeo.txt`, uprościliśmy go tak bardzo, jak to tylko możliwe, usuwając ręcznie całą interpunkcję. Rzeczywisty tekst ma dużo interpunkcji, tak jak to pokazano poniżej.

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
```

Ponieważ funkcja `split()` szuka spacji i traktuje słowa jako tokeny oddzielone spacjami, traktowalibyśmy słowa "soft!" i "soft" jako *różne* słowa i dla każdego z nich tworzylibyśmy osobny wpis w słowniku.

Również ze względu na to, że plik ma tekst pisany dużymi literami, traktowalibyśmy "who" i "Who" jako różne słowa o różnej liczności.

Oba te problemy możemy rozwiązać za pomocą metod związanych z tekstowym typem danych `str`, tj. `lower()`, `maketrans()` i `translate()`. Metoda `translate()` jest najbardziej wyrafinowaną z tych metod. Oto dokumentacja dla tej metody:

`line.translate(str.maketrans(fromstr, tostr, deletestr))`

Zmienia znaki w `fromstr` na znak na tej samej pozycji w `tostr` i usuwa wszystkie znaki, które są w `deletestr`. Znaki z `fromstr` i `tostr` mogą być pustymi napisami, a parametr `deletestr` może zostać pominięty.

Moduł `string` dostarcza gotowe sekwencje znaków, które reprezentują np. liczby lub litery. Przykładowo, w `string.digits` znajdziemy napis zwierający liczby:

```
>>> import string
>>> string.digits
'0123456789'
```

Metoda `maketrans()` z pustymi napisami `fromstr` i `tostr` utworzy nam słownik, w którym będziemy mieli informację o tym na co ma być zamieniony dany znak (a w zasadzie jego reprezentacja liczbowa). Możemy zauważyć, że za każdym razem chcemy dokonać zamiany na `None`, co oznacza, że chcemy usunąć dany znak:

```
>>> str.maketrans('', '', string.digits)
{48: None, 49: None, 50: None, 51: None, 52: None,
 53: None, 54: None, 55: None, 56: None, 57: None}
```

Taki utworzony słownik możemy potem wykorzystać w metodzie `translate()` do efektywnego usunięcia znaków.

Wracając do naszego programu, nie będziemy określać parametru `fromstr` i `tostr` (będą pustymi napisami), ale użyjemy parametru `deletestr` do usunięcia wszystkich znaków interpunkcyjnych. Pozwolimy nawet Pythonowi wskazać nam sekwencję znaków, które uważa za "interpunkcję":

```
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Dokonujemy następujących zmian w naszym programie:

```
import string

fname = input('Podaj nazwę pliku: ')
try:
    fhand = open(fname)
except:
    print('Nie można otworzyć pliku:', fname)
    exit()

counts = dict()
for line in fhand:
    line = line.translate(str.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Kod źródłowy: https://py4e.pl/code3/count2.py
```

Częścią uczenia się "Sztuki Pythona" lub "Myślenia po pythonowemu" jest uświadomienie sobie, że Python często ma wbudowane rozwiązania dla wielu prostych problemów związanych z analizą danych. Z czasem zobaczysz wystarczająco dużo kodu przykładowego i przeczytasz wystarczająco dużo dokumentacji, tak aby wiedzieć, gdzie szukać, by sprawdzić, czy ktoś już nie napisał czegoś, co znacznie ułatwi Ci pracę.

Poniżej znajduje się skrócony wynik działania programu na pliku `romeo-full.txt`, który jest dostępny pod adresem https://py4e.pl/code3/romeo-full.txt.

```
Podaj nazwę pliku: romeo-full.txt
{'romeo': 40, 'and': 42, 'juliet': 32, 'act': 1, '2': 2,
 'scene': 2, 'ii': 1, 'capulets': 1, 'orchard': 2,
 'enter': 1, 'he': 5, 'jests': 1, 'at': 9, 'scars': 1,
 'that': 30, 'never': 2, 'felt': 1, 'a': 24, ...}
```

Przeglądanie powyższego wyniku nadal jest niewygodne i choć możemy użyć Pythona, by dał nam dokładnie to, czego chcemy, to żeby to zrobić, musimy najpierw poznać *krotki*. Wrócimy do tego przykładu w kolejnym rozdziale.

9.5. Debugowanie

Podczas pracy z większymi zbiorami danych debugowanie poprzez wyświetlanie i ręczne sprawdzanie danych może okazać się niewygodne. Oto kilka sugestii dotyczących debugowania dużych zbiorów danych:

Stopniowe redukowanie danych wejściowych Jeśli to możliwe, zmniejsz rozmiar zbioru danych. Na przykład, jeśli program odczytuje plik tekstowy, zacznij od pierwszych 10 linii lub od najmniejszego możliwego fragmentu. Możesz albo edytować same pliki, albo (lepiej) zmodyfikować program tak, by czytał tylko pierwsze `n` linii.

Jeśli jest jakiś błąd, możesz zredukować `n` do najmniejszej wartości, która generuje błąd, a następnie zwiększać tę wartość stopniowo, w międzyczasie znajdując i poprawiając błędy.

Sprawdź podsumowania i typy Zamiast wyświetlać i sprawdzać cały zbiór danych, spróbuj wyświetlić podsumowanie danych: na przykład liczbę pozycji w słowniku lub sumę liczb.

Często błędy czasu wykonania powoduje wartość, która nie jest właściwego typu. Do debugowania tego typu błędów często wystarczy wypisać typ wartości.

Napisz mechanizm do samokontroli Czasami warto napisać kod do automatycznego sprawdzania błędów. Na przykład, jeśli obliczasz średnią z listy liczb, możesz sprawdzić, czy wynik nie jest większy od największego elementu na liście lub mniejszy od najmniejszego. Nazywa się to "sprawdzaniem poczytalności" (ang. *sanity check*), ponieważ wykrywa wyniki, które są "zupełnie nielogiczne".

Inny rodzaj sprawdzenia porównuje wyniki dwóch różnych obliczeń, tak aby sprawdzić czy są one spójne. Nazywa się to "sprawdzaniem spójności" (ang. *consistency check*).

Ładne wyświetlenie wyniku Czytelne sformatowanie wyników debugowania może ułatwić wykrycie błędu.

Pamiętaj, że czas spędzony na budowaniu dodatkowych zabezpieczeń w Twoim programie może skrócić czas, który spędzisz na debugowaniu.

9.6. Słowniczek

element słownika Inna nazwa dla pary klucz-wartość.
funkcja haszująca Funkcja używana przez tablice mieszające do obliczania lokalizacji dla danego klucza.
histogram Zbiór liczników.
implementacja Sposób przeprowadzenia obliczeń.
klucz Obiekt, który pojawia się w słowniku jako pierwsza część pary klucz-wartość.
para klucz-wartość Reprezentacja mapowania z klucza do wartości.

pętle zagnieżdżone Kiedy istnieje jedna lub więcej pętli "wewnątrz" innej pętli. Pętla wewnętrzna wykonuje się do końca za każdym razem, gdy pętla zewnętrzna iteruje się raz.
słownik Mapowanie z zestawu kluczy do odpowiadających im wartości.
tablica mieszająca Algorytm używany do implementacji słowników Pythona; inaczej tablica z haszowaniem.
wartość Obiekt, który pojawia się w słowniku jako druga część pary klucz-wartość. Jest to bardziej szczegółowe określenie niż nasze poprzednie użycie słowa "wartość".
wyszukiwanie Operacja słownikowa, która dla podanego klucza znajduje odpowiadającą mu wartość.

9.7. Ćwiczenia

Ćwiczenie 2. Napisz program, który każdą wiadomość mailową (zapisaną w pliku w formacie Mbox) skategoryzowałby według dni tygodnia. Aby to zrobić, poszukaj wierszy rozpoczynających się od "From", a następnie poszukaj trzeciego słowa i zachowaj bieżące zliczenia dla każdego dnia. Na koniec programu wyświetl zawartość swojego słownika (kolejność nie ma znaczenia).

Przykładowa linia:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

Przykładowe uruchomienie:

```
Podaj nazwę pliku: mbox-short.txt
{'Sat': 1, 'Fri': 20, 'Thu': 6}
```

Ćwiczenie 3. Napisz program, który odczytuje dane z pliku w formacie Mbox i przy pomocy słownika tworzy histogram. Chcemy zliczyć, ile wiadomości przyszło z każdego adresu e-mail. Jako wynik działania programu ma zostać wyświetlony utworzony słownik.

```
Podaj nazwę pliku: mbox-short.txt
{'stephen.marquard@uct.ac.za': 2, 'louis@media.berkeley.edu': 3,
 'zqian@umich.edu': 4, 'rjlowe@iupui.edu': 2, 'cwen@iupui.edu': 5,
 'gsilver@umich.edu': 3, 'wagnermr@iupui.edu': 1,
 'antranig@caret.cam.ac.uk': 1, 'gopal.ramasammycook@gmail.com': 1,
 'david.horwitz@uct.ac.za': 4, 'ray@media.berkeley.edu': 1}
```

Ćwiczenie 4. Do powyższego programu dodaj kod, tak by dowiedzieć się, kto wysłał najwięcej e-maili. Po przeczytaniu wszystkich danych i utworzeniu słownika, przeszukaj tej osoby za pomocą pętli wyszukującej największą wartość (patrz: rozdział 5, sekcja "Pętle typu maksimum i minimum"). Na koniec programu wyświetl informację dotyczącą adresu mailowego i liczbie wysłanych wiadomości.

```
Podaj nazwę pliku: mbox-short.txt
cwen@iupui.edu 5

Podaj nazwę pliku: mbox.txt
zqian@umich.edu 195
```

Ćwiczenie 5. Napisz program zapamiętujący nazwę domeny, z której została wysłana wiadomość (zamiast informacji, od kogo pochodziła wiadomość, tzn. całego adresu e-mail). Na koniec programu wyświetl zawartość słownika.

```
Podaj nazwę pliku: mbox-short.txt
{'uct.ac.za': 6, 'media.berkeley.edu': 4, 'umich.edu': 7,
 'iupui.edu': 8, 'caret.cam.ac.uk': 1, 'gmail.com': 1}
```