Rozdział 8

Listy

8.1. Lista jest sekwencją

Podobnie jak ciąg znaków, *lista* jest sekwencją wartości. W ciągu znaku wartości są znakami, natomiast w liście mogą być dowolnego typu. Wartości występujące w liście są nazywane *elementami*, z rzadka *pozycjami*.

Istnieje kilka sposobów na stworzenie nowej listy; najprostszym jest umieszczenie elementów w nawiasach kwadratowych (`[` i `]`):

```
[10, 20, 30, 40]
['chrupiąca żabka', 'pęcherz barana', 'paw skowronka']
```

Pierwszym przykładem jest lista składająca się z czterech liczb całkowitych. Drugi przykład to lista trzech ciągów znaków. Elementy listy nie muszą być tego samego typu. Poniższa lista zawiera ciąg znaków, liczbę zmiennoprzecinkową, liczbę całkowitą i (uwaga!) inną listę:

```
['spam', 2.0, 5, [10, 20]]
```

Gdy jedna lista znajduje się w innej liście, to mówimy, że jest *zagnieżdżona*.

Lista, która nie zawiera żadnych elementów, nazywana jest listą pustą. Możesz utworzyć taką listę, używając pustych nawiasów kwadratowych, tj. `[]`.

Jak pewnie się spodziewałeś, możesz przypisać wartości listy do zmiennych:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

8.2. Listy są zmienne

Składnia dostępu do elementów listy jest taka sama, jak w przypadku dostępu do znaków w napisach: używa się operatora w postaci nawiasów kwadratowych. Wyrażenie wewnątrz nawiasów określa indeks. Pamiętaj, że indeksy zaczynają się od 0:

```
>>> print(cheeses[0])
Cheddar
```

W odróżnieniu od ciągów znaków, listy są zmienne, ponieważ możesz zmienić kolejność elementów na liście lub ponownie przypisać jakiś element do listy. Kiedy nawias kwadratowy pojawi się po lewej stronie przypisania, identyfikuje on pozycję listy, do której zostanie coś przypisane.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

Pierwszym elementem `numbers`, który kiedyś wynosił 123, jest teraz 5.

Możesz myśleć o liście jako o relacji między indeksami i elementami. Ta relacja nazywana jest *mapowaniem*; każdy indeks „mapuje" do jednego z elementów.

Indeksy listy działają tak samo jak indeksy ciągów znaków:

- Każde wyrażenie będące liczbą całkowitą może być użyte jako indeks.
- Jeśli spróbujesz odczytać lub zapisać pozycję, która nie istnieje, otrzymasz `IndexError`.
- Jeśli indeks ma wartość ujemną, to oblicza się go wstecz od końca listy.

Operator `in` działa również na listach.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

8.3. Poruszanie się po listach

Najczęstszym sposobem na przejście po elementach listy jest pętla `for`. Składnia jest taka sama jak dla ciągów znaków:

```
for cheese in cheeses:
    print(cheese)
```

Jest to dobre rozwiązanie, jeśli musisz tylko odczytać elementy z listy. Ale jeśli chcesz coś w niej dopisać lub zaktualizować, to potrzebujesz indeksów. W takiej sytuacji najczęściej wykorzystuje się połączenie funkcji `range()` i `len()`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

Powyższa pętla przechodzi przez listę i aktualizuje każdy element. Funkcja `len()` zwraca liczbę elementów na liście. Natomiast funkcja `range()` zwraca listę indeksów od 0 do $n-1$, gdzie n jest długością listy. Przy każdej iteracji pętli `i` kolejno uzyskuje indeks elementu. Instrukcja przypisania w ciele pętli używa `i` do odczytania starej wartości elementu i przypisania nowej wartości.

Pętla `for` przechodząca przez pustą listę nigdy nie wykonuje ciała pętli:

```
empty = []
for x in empty:
    print('Nikt tego nigdy nie zobaczy.')
```

Chociaż lista może zawierać inną listę, to zagnieżdżona lista nadal liczy się jako pojedynczy element. Długość poniższej listy wynosi cztery:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

8.4. Operacje na listach

Operator + konkatenuje (łączy) listy:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Z kolei operator * powtarza listę określoną liczbę razy:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

W pierwszym przykładzie lista jest powtórzona cztery razy. Drugi przykład powtarza listę trzy razy.

8.5. Wycinki list

Operator wycinania działa również na listach:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Jeśli pominiesz pierwszy indeks, wycinek zaczyna się od początku listy. Jeśli pominiesz drugi indeks, wycinek idzie aż do końca. Zatem jeśli pominiesz oba, wycinek jest kopią całej listy.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Ponieważ listy są zmienne, często przydatne jest zrobienie kopii przed wykonaniem innych operacji, które namieszają i zmasakrują ich elementy.

Operator wycinania zastosowany po lewej stronie instrukcji przypisania może zmienić wiele elementów:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

8.6. Metody obiektów będących listami

Python udostępnia metody, które działają na listach. Na przykład metoda `append()` dodaje nowy element na końcu listy:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

Metoda `extend()` działa podobnie, z tym że przyjmuje jako argument listę elementów do dopisania:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

W powyższym przykładzie lista `t2` pozostaje niezmieniona.

Metoda `sort()` układa elementy listy od najmniejszego do największego:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Większość metod listowych nie zwraca żadnej konkretnej wartości; modyfikują one listę i zwracają `None`. Tego typu modyfikacja nazywana jest *modyfikacją w miejscu*. Jeśli przypadkowo napiszesz `t = t.sort()`, zapewne będziesz zawiedziony zwróconym wynikiem.

8.7. Usuwanie elementów

Istnieje kilka sposobów na usunięcie elementów z listy. Jeśli znasz indeks elementu, który chcesz usunąć, możesz użyć metody `pop()`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

Metoda `pop()` modyfikuje listę i zwraca element, który został usunięty. Jeśli nie podasz indeksu, to usunie on i zwróci ostatni element listy.

Jeśli nie potrzebujesz usuniętej wartości, możesz użyć operatora `del`:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

Jeśli znasz element, który chcesz usunąć (ale nie jego indeks), możesz użyć metody `remove()`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

Wartością zwracaną przez `remove()` jest `None`.

Aby usunąć więcej niż jeden element, możesz użyć `del` z indeksami podanymi w postaci wycinka:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

Jak zwykle wycinek wybiera wszystkie elementy aż do drugiego indeksu, ale bez ostatniej pozycji.

8.8. Listy i funkcje

Istnieje szereg wbudowanych funkcji, które mogą być używane na listach, pozwalających na szybkie przeglądanie listy bez konieczności pisania własnych pętli:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums) / len(nums))
25.666666666666668
```

Funkcja `sum()` działa tylko wtedy, gdy elementy listy są liczbami. Pozostałe funkcje (`max()`, `len()` itp.) działają z listami ciągów znaków i innych typów, które mogą być porównywane.

Moglibyśmy przepisać nasz wcześniejszy program, który teraz obliczyłby średnią z liczb podanych przez użytkownika za pomocą listy.

Poniżej mamy program, który liczy średnią bez listy:

```
total = 0
count = 0
while True:
    inp = input('Wprowadź liczbę: ')
    if inp == 'gotowe': break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print('Średnia:', average)

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

W powyższym programie mamy zmienne `count` i `total`, aby podczas kolejnych próśb o wprowadzenie danych pamiętać, ile już wprowadzono liczb, oraz by przechowywać sumę bieżącą wprowadzonych liczb.

W alternatywnym podejściu możemy po prostu zapamiętać każdą wprowadzoną liczbę i pod koniec użyć wbudowanych funkcji do obliczenia sumy i zliczenia elementów.

```
numlist = list()
while True:
    inp = input('Wprowadź liczbę: ')
    if inp == 'gotowe': break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('Średnia:', average)

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

Przed rozpoczęciem pętli tworzymy pustą listę, w tym przypadku przy pomocy funkcji `list()` (pustą listę moglibyśmy też utworzyć przy pomocy `[]`). Następnie za każdym razem, gdy otrzymamy nową liczbę, dołączamy ją do listy. Na końcu programu po prostu obliczamy sumę liczb występujących na liście i dzielimy ją przez liczbę elementów listy, tak aby uzyskać średnią.

8.9. Listy i ciągi znaków

Ciąg znaków jest sekwencją znaków, a lista jest sekwencją wartości, ale lista znaków i ciąg znaków to nie to samo. Aby przekonwertować ciąg znaków na listę znaków, możesz użyć funkcji `list()`:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Ponieważ `list()` jest nazwą wbudowanej funkcji, powinieneś unikać używania jej jako nazwy zmiennej. Unikam również litery „l", ponieważ w wyglądzie jest zbyt podobna do liczby „1". Właśnie dlatego używam „t".

Funkcja `list()` rozbija ciąg znaków na pojedyncze litery. Jeśli chcesz rozbić ciąg znaków na słowa, możesz użyć metody `split()`:

```
>>> s = 'usycham z tęsknoty za fiordami'
>>> t = s.split()
>>> print(t)
['usycham', 'z', 'tęsknoty', 'za', 'fiordami']
>>> print(t[2])
tęsknoty
```

Gdy już użyjesz `split()`, aby rozbić ciąg znaków na listę słów, możesz użyć operatora indeksu (nawias kwadratowy), żeby przyjrzeć się konkretnemu słowu na liście.

Możesz wywołać `split()` z opcjonalnym argumentem zwanym *separatorem*, który określa, jakie znaki mają być użyte do określania granic słów. Poniższy przykład używa myślnika jako separatora:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

Metoda `join()` jest odwrotnością `split()`. Przyjmuje ona listę ciągów znaków i konkatenuje jej elementy. `join()` jest metodą obiektów będących ciągami znaków, więc musisz wywołać ją na separatorze i przekazać listę jako argument:

```
>>> t = ['usycham', 'z', 'tęsknoty', 'za', 'fiordami']
>>> delimiter = ' '
>>> delimiter.join(t)
'usycham z tęsknoty za fiordami'
```

W tym przypadku separator jest znakiem spacji, więc `join()` umieszcza spację między słowami. Aby połączyć ciągi znaków bez spacji, możesz użyć pustego ciągu znaków (`''` lub `""`) jako separatora.

8.10. Parsowanie linii

Zwykle gdy czytamy z pliku, chcemy zrobić coś innego, niż tylko wyświetlić całą linię. Często chcemy znaleźć „interesujące linie", a następnie *rozdzielić* linię, aby później znaleźć jakąś interesującą *część* tej linii. Co by było, gdybyśmy chcieli wyświetlić dzień tygodnia z tych linii, które zaczynają się od „From"?

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

Metoda `split()` jest bardzo skuteczna, gdy ma do czynienia z tego rodzaju problemem. Możemy napisać mały program, który szuka linii zaczynających się od „From", rozdzielić te linie przy pomocy `split()`, a następnie wyświetlić trzecie słowo występujące w danej linii:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From '): continue
    words = line.split()
    print(words[2])

# Kod źródłowy: https://py4e.pl/code3/search5.py
```

Program generuje następujące wyjście:

```
Sat
Fri
Fri
Fri
...
```

Później poznamy coraz bardziej wyrafinowane techniki wybierania linii do dalszego przetwarzania oraz dowiemy się, jak je przeanalizować, tak aby znaleźć dokładnie ten fragment informacji, którego szukamy.

8.11. Obiekty i wartości

Jeśli wykonamy poniższe instrukcje przypisania:

```
a = 'banan'
b = 'banan'
```

to wiemy, że zarówno `a`, jak i `b` odnoszą się do ciągu znaków, ale nie wiemy, czy odnoszą się one do *tego samego* ciągu znaków. Istnieją dwa możliwe stany:

W jednym przypadku `a` i `b` odnoszą się do dwóch różnych obiektów, które mają tę samą wartość. W drugim przypadku odnoszą się one do tego samego obiektu.

Aby sprawdzić, czy dwie zmienne odnoszą się do tego samego obiektu, możesz użyć operatora `is`.

Rysunek 8.1. Zmienne i obiekty

```
>>> a = 'banan'
>>> b = 'banan'
>>> a is b
True
```

W powyższym przykładzie Python stworzył tylko jeden obiekt ciągu znaków, a obie zmienne a i b odnoszą się właśnie do tego obiektu.

Ale kiedy tworzysz dwie listy, otrzymujesz dwa obiekty:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

W tym przypadku powiedzielibyśmy, że te dwie listy są *równoważne*, ponieważ mają te same elementy, ale nie są *identyczne*, ponieważ nie są tym samym obiektem. Jeżeli dwa obiekty są identyczne, to są one również równoważne, ale jeżeli są równoważne, to niekoniecznie są identyczne.

Do tej pory używaliśmy zamiennie słów „obiekt" i „wartość", ale powinniśmy mówić, że obiekt ma wartość. Jeśli wykonasz `a = [1,2,3]`, to a odnosi się do obiektu listy, którego wartością jest konkretna sekwencja elementów. Jeśli inna lista ma te same elementy, to powiedzielibyśmy, że ma taką samą wartość.

8.12. Aliasy

Jeśli a odnosi się do obiektu, a Ty przypisujesz `b = a`, to obie zmienne odnoszą się do tego samego obiektu:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

Powiązanie zmiennej z obiektem nazywane jest *referencją*. W powyższym przykładzie istnieją dwie referencje do tego samego obiektu.

Obiekt z więcej niż jedną referencją ma więcej niż jedną nazwę, więc mówimy, że obiekt jest *aliasowany*.

Jeżeli aliasowany obiekt jest zmienny, to zmiany dokonane przy użyciu jednego aliasu wpływają na drugi:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Chociaż takie zachowanie może być użyteczne, ma ono skłonność do generowania błędów. Ogólnie rzecz biorąc, jeśli pracujesz z obiektami zmiennymi, to bezpieczniej jest unikać tworzenia ich aliasów.

W przypadku obiektów niezmiennych, takich jak ciągi znaków, aliasowanie nie jest aż tak dużym problemem. W poniższym przykładzie:

```
a = 'banan'
b = 'banan'
```

prawie nigdy nie ma różnicy, czy a i b odnoszą się do tego samego ciągu znaków, czy też nie.

8.13. Argumenty będące listami

Kiedy przekazujesz listę do funkcji, funkcja ta otrzymuje referencję do listy. Jeżeli funkcja modyfikuje parametr będący listą, to z poziomu wywoływania funkcji zauważymy tę zmianę. Na przykład `delete_head()` usuwa pierwszy element z listy:

```
def delete_head(t):
    del t[0]
```

Oto jak możemy jej użyć:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

Parametr `t` i zmienna `letters` są aliasami dla tego samego obiektu.

Ważne jest, by odróżniać operacje, które modyfikują listy, od operacji, które tworzą nowe listy. Na przykład, metoda `append()` modyfikuje listę, ale operator + tworzy nową listę:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None

>>> t1 = [1, 2]
>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t1 is t3
False
```

Ta różnica jest istotna, gdy piszesz definicję funkcji modyfikującej listy. Na przykład poniższa funkcja *nie usuwa* nagłówka listy:

```
def bad_delete_head(t):
    t = t[1:]               # ŹLE!
```

Operator wycinania tworzy nową listę, a przypisanie sprawia, że zmienna `t` odnosi się teraz do tej nowej listy, ale żadna z tych operacji nie ma żadnego wpływu na listę, która została przekazana do funkcji jako argument.

Zamiast tego można napisać funkcję, która tworzy i zwraca nową listę. Na przykład `tail()` zwraca wszystko oprócz pierwszego elementu listy:

```
def tail(t):
    return t[1:]
```

Funkcja ta pozostawia pierwotną listę bez zmian. Poniżej pokazano, jak można użyć tej funkcji:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

Ćwiczenie 1. Napisz funkcję o nazwie `chop()`, która przyjmuje listę i modyfikuje ją, usuwając z niej pierwszy i ostatni element; funkcja ma zwracać `None`. Następnie napisz funkcję o nazwie `middle()`, która przyjmuje listę i zwraca nową listę, która zawiera wszystkie elementy wejściowej listy za wyjątkiem pierwszego i ostatniego elementu.

8.14. Debugowanie

Nieostrożne korzystanie z list (i innych zmiennych obiektów) może prowadzić do długich godzin debugowania. Oto kilka typowych pułapek wraz ze sposobami na ich uniknięcie:

1. Nie zapominaj, że większość metod listowych modyfikuje argument i zwraca `None`. Jest to zachowanie odmienne w stosunku do tego, co widzieliśmy w metodach obiektów będących ciągami znaków, które zwracają nowy ciąg znaków, a oryginał zostawiają w spokoju.

   Jeśli jesteś przyzwyczajony do pisania takiego kodu dla ciągów znaków jak ten:

   ```
   word = word.strip()
   ```

   to kuszące jest pisanie kodu dla list w poniższy sposób:

   ```
   t = t.sort()            # ŹLE!
   ```

   Metoda `sort()` zwraca `None`, więc następna operacja, którą wykonasz z `t`, może się nie udać.

   Przed użyciem metod i operatorów dla list powinieneś dokładnie przeczytać dokumentację, a następnie przetestować ją w trybie interaktywnym. Metody i operatory, które listy współdzielą z innymi sekwencjami (np. ciągami znaków), są udokumentowane pod adresem:

   https://docs.python.org/library/stdtypes.html#common-sequence-operations

   Metody i operatory, które stosuje się tylko do zmiennych sekwencji, są udokumentowane na stronie internetowej:

   https://docs.python.org/library/stdtypes.html#mutable-sequence-types

2. Wybierz idiom i trzymaj się go.

   Częścią problemu z listami jest to, że jest zbyt wiele sposobów na wykonanie jakiejś rzeczy. Na przykład, aby usunąć element z listy, możesz użyć `pop()`, `remove()`, `del`, a nawet użyć przypisania poprzez wycinanie.

   Aby dodać element, możesz użyć metody `append()` lub operatora `+`. Ale nie zapomnij, że oba poniższe sposoby są prawidłowe:

   ```
   t.append(x)
   t = t + [x]
   ```

   Natomiast te są niepoprawne:

   ```
   t.append([x])           # ŹLE!
   t = t.append(x)         # ŹLE!
   t + [x]                 # ŹLE!
   t = t + x               # ŹLE!
   ```

   Wypróbuj każdy z tych przykładów w trybie interaktywnym, tak aby upewnić się, że rozumiesz, co one robią. Zauważ, że tylko ostatni z nich powoduje błąd w czasie wykonania; pozostałe trzy są dozwolone, ale robią coś niepoprawnego.

3. Rób kopie, by unikać aliasowania.

   Jeśli chcesz użyć metody takiej jak `sort()`, która modyfikuje argument, ale musisz także zachować oryginalną listę, to możesz uprzednio zrobić jej kopię.

   ```
   orig = t[:]
   t.sort()
   ```

W powyższym przykładzie możesz również użyć wbudowanej funkcji `sorted()`, która zwraca nową, posortowaną listę, a oryginał pozostawia nienaruszony. Jednak w tym przypadku powinieneś unikać używania `sorted` jako nazwy dla zmiennej!

4. Listy, `split()` i pliki

   Kiedy odczytujemy i analizujemy pliki, pojawia się wiele okazji, by trafić na takie dane wejściowe, które mogą wysypać nasz program, więc dobrym pomysłem jest powrót do wzorca *strażnika*, gdy piszemy programy, które czytają z pliku i szukają „igły w stogu siana".

   Wróćmy do naszego programu, który szuka dnia tygodnia w wierszach pliku:

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

   Ponieważ rozbijamy linijkę na słowa, możemy zrezygnować z użycia `startswith()` i po prostu spojrzeć na pierwsze słowo linii, aby określić, czy w ogóle jesteśmy zainteresowani tym wierszem. Możemy użyć `continue` do pominięcia linii, które zaczynają się od słowa „From", tak jak w przykładzie poniżej:

   ```
   fhand = open('mbox-short.txt')
   for line in fhand:
       words = line.split()
       if words[0] != 'From' : continue
       print(words[2])
   ```

   Wygląda to znacznie prościej i nawet nie musimy używać `rstrip()` do usunięcia znaku końca linii. Ale czy to jest faktycznie lepsze rozwiązanie? Spójrzmy na wynik uruchomienia skryptu:

   ```
   Sat
   Traceback (most recent call last):
   File "search8.py", line 4, in <module>
       if words[0] != 'From' : continue
   IndexError: list index out of range
   ```

   Niby działa i widzimy dzień tygodnia dla pierwszej linii (Sat), ale potem program się wysypuje i widzimy błąd razem ze zrzutem z mechanizmu *traceback*. Co poszło nie tak? Jakie pomyłki w danych spowodowały, że nasz elegancki, pomysłowy i bardzo pythonowy program zawiódł?

   Mógłbyś się na ten kod długo gapić i zachodzić w głowę lub poprosić kogoś o pomoc, ale szybsze i mądrzejsze podejście to dodanie funkcji `print()`. Najlepszym miejscem na dodanie wyświetlania danych jest tuż przed wierszem, w którym program się wysypał i pokazanie tych danych, które wydają się być przyczyną niepowodzenia.

   Dzięki takiemu podejściu do problemu możemy wygenerować wiele linii na wyjściu, ale przynajmniej od razu będziemy mieli jakąś wskazówkę co do problemu. Tak więc dodajemy wyświetlenie zmiennej `words` tuż przed piątym wierszem. Dodajemy nawet przedrostek „Debug:" do linii, dzięki czemu możemy oddzielić nasze zwykłe wyjście od wyjścia związanego z debugowaniem.

   ```
   fhand = open('mbox-short.txt')
   for line in fhand:
       words = line.split()
       print('Debug:', words)
       if words[0] != 'From' : continue
       print(words[2])
   ```

   Kiedy uruchomimy program, przez ekran przewinie się duża ilość danych, ale na końcu zobaczymy nasze debugowane dane wyjściowe i zrzut z mechanizmu *traceback*, więc wiemy, co się wydarzyło tuż przed błędem.

   ```
   (...)
   Debug: ['X-DSPAM-Confidence:', '0.8475']
   Debug: ['X-DSPAM-Probability:', '0.0000']
   Debug: []
   ```

```
Traceback (most recent call last):
File "search9.py", line 5, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

Dla każdego wiersza wypisujemy listę słów, które otrzymujemy z metody `split()` podczas dzielenia linii na słowa. Gdy program się wysypie, lista słów jest pusta `[]`. Jeśli otworzymy plik w edytorze tekstu i spojrzymy na niego, to zobaczymy następujący widok:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

Błąd pojawia się wtedy, gdy nasz program natknie się na pustą linię! Oczywiście w pustym wierszu znajduje się „zero słów". Dlaczego nie pomyśleliśmy o tym, gdy pisaliśmy kod? Kiedy nasz kod szuka pierwszego słowa (`word[0]`), by sprawdzić, czy pasuje do „From", otrzymujemy błąd „index out of range".

Jest to oczywiście idealne miejsce na dodanie kodu ze *strażnikiem*, tak aby uniknąć sprawdzania pierwszego słowa, jeśli go nie ma. Istnieje wiele sposobów na ochronę takiego kodu; zdecydujemy się sprawdzić liczbę słów, które mamy, zanim spojrzymy na pierwsze słowo:

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    # print('Debug:', words)
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print(words[2])
```

Najpierw zakomentowaliśmy instrukcję debugowania zamiast ją od razu usuwać, na wypadek gdyby nasza modyfikacja zawiodła i trzeba było debugować kod ponownie. Następnie dodaliśmy instrukcję strażnika, który sprawdza, czy mamy zero słów, a jeśli tak, to używamy `continue` by przejść do następnej linii w pliku.

Możemy myśleć o tych dwóch instrukcjach `continue` jako o pomocy w filtrowaniu zestawu linii, które są dla nas „interesujące" i które chcemy jeszcze trochę przetworzyć. Linia, która nie ma słów, jest dla nas „nieciekawa", więc przechodzimy do następnej linii. Linia, która nie ma „From" jako pierwszego słowa, jest dla nas nieciekawa, więc ją pomijamy.

Zmodyfikowany program działa, więc być może jest poprawny. Nasza instrukcja strażnika zapewnia, że `word[0]` nigdy nie spowoduje błędu, ale może to nie być wystarczające. Podczas programowania musimy zawsze myśleć: „Co może się nie udać?".

Ćwiczenie 2. Sprawdź, która linia powyższego programu nie jest jeszcze właściwie zabezpieczona. Skonstruuj taki plik tekstowy, który wysypie program, a następnie zmodyfikuj program tak, by był on odpowiednio zabezpieczony. Przetestuj poprawiony program, by upewnić się, że obsługuje Twój nowy plik tekstowy.

Ćwiczenie 3. Napisz ponownie kod z powyższego przykładu, ale bez dwóch instrukcji `if`. Zamiast tego użyj pojedynczej instrukcji `if` ze złożonym wyrażeniem logicznym używającym operatora logicznego (zastosuj wzorzec strażnika).

8.15. Słowniczek

aliasowanie Okoliczność, w której dwie lub więcej zmiennych odnosi się do tego samego obiektu.

element Jedna z wartości na liście (lub innej sekwencji); nazywana także pozycją.
identyczny Będący tym samym obiektem (co oznacza, że jest też równoważny).
indeks Wartość będąca liczbą całkowitą, która wskazuje element na liście.
lista Sekwencja wartości.
lista zagnieżdżona Lista, która jest elementem innej listy.
obiekt Coś, do czego może odnosić się zmienna. Obiekt ma typ i wartość.
przejście po liście Sekwencyjny dostęp do każdego elementu listy.
referencja Związek między zmienną a jej wartością.
równoważny Posiadający tę samą wartość.
separator Znak lub ciąg znaków używany do wskazania, gdzie ciąg znaków powinien zostać podzielony.

8.16. Ćwiczenia

Ćwiczenie 4. Pobierz kopię pliku https://py4e.pl/code3/romeo.txt. Napisz program, który będzie przechowywał listę wyrazów występujących w pliku. Na początku otwórz plik `romeo.txt` i czytaj go linia po linii. Każdą z nich podziel na listę słów za pomocą metody `split()`. Dla każdego słowa sprawdź, czy znajduje się ono już na liście. Jeśli go nie ma, to dodaj je do listy. Gdy program zakończy pracę, posortuj i wypisz wynikowe słowa w kolejności alfabetycznej.

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

Ćwiczenie 5. Napisz program, który czyta dane mailowe z pliku w formacie Mbox, a gdy znajdzie wiersz zaczynający się od „From", niech podzieli go na słowa, korzystając z metody `split()`. Interesuje nas, kto wysłał wiadomość – informacja ta jest drugim słowem w wierszu zawierającym „From".

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

Musisz przeparsować linię zawierającą „From" i wyświetlić drugie słowo dla każdej takiej linii. Następnie musisz zliczyć liczbę wierszy „From" (nie „From:") i na końcu wyświetlić tę liczbę. Poniżej znajduje się przykładowe poprawne wyjście (część linii wyjścia została usunięta):

```
Podaj nazwę pliku: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...część linii wyjścia usunięta...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
Mamy 27 linii, w których From jest pierwszym wyrazem
```

Ćwiczenie 6. Napisz ponownie program, który poprosi użytkownika o listę liczb i wypisze na końcu największą i najmniejszą z nich, gdy użytkownik wpisze „gotowe". Napisz program w ten sposób, że zapisze on wprowadzone przez użytkownika liczby na liście i użyje funkcji `max()` oraz `min()` do znalezienia największej i najmniejszej liczby po zakończeniu pętli.

```
Wprowadź liczbę: 6
Wprowadź liczbę: 2
Wprowadź liczbę: 9
Wprowadź liczbę: 3
Wprowadź liczbę: 5
Wprowadź liczbę: gotowe
Największa: 9.0
Najmniejsza: 2.0
```