Laboratorium programistyczne — Python: Sumator z elementów logicznych

Projekt „Matematyka dla ciekawych świata”
Robert Paciorek

2019-03-07

1 Listy

Wspomnieliśmy wcześniej, że zapis `[0, 2, 3]` oznacza listę złożoną z 3 elementów: liczb 0, 2 i 3. Zastosowanie list jest dość szerszeokie i teraz poświęcimy im troche więcej uwagi.

Lista stanowi pewnego rodzaju kontener do przechowywania innych zmiennych, w którym elementy zorganizowane są na zasadzie określenia ich (względnej) kolejności. Lista może zawierać elementy różnych typów.

Na listach możemy wykonywać m.in. operacje modyfikowania, czy też usuwania jej elementów:

```
l = ["i", "C", 0, "M"]
l[0] = "I"
del l[2]
print(l)
```

```
['I', 'C', 'M']
```

W powyższym przykładzie widzimy:

- Modyfikację pierwszego elementu listy (`l[0] = "I"`), z użyciem odwołania poprzez numer elementu. Elementy list numerujemy od zera. Ujemne wartości oznaczają numerowanie od końca listy, czyli -1 jest ostatnim elementem listy, -2 przedostatnim, itd.
- Usunięcie trzeciego elementu listy (**del** `l[2]`). Powoduje to zmianę numeracji kolejnych elementów.

1.1 iterowanie po elementach listy — pętla for

Python oferuje pętlę umożliwiającą łatwe operowanie na elementach kolekcji takich jak listy. Służy do tego pętla **for**. Pozwala ona na wykonanie operacji na każdym elemencie listy.

Na przykład poniższa pętla wypisze wszystkie elementy listy podniesione do kwadratu:

```
for x in [1, 2, 3, 4]:
    print(x * x)
```

```
1
4
9
16
```

Zauważ, że wnętrze pętli jest wyznaczone w sposób analogiczny do wnętrza pętli **while**.

Zadanie 1.1.1

Napisz funkcję `bezwzgledne(lista)`, która dla danej listy liczb wypiszę listę wartości bezwzględnych tych liczb, tj. liczby ujemne zamieni na przeciwne, a liczby nieujemne pozostawi bez zmian. Poszczególne liczby powinny być oddzielone pojedynczymi spacjami. Przykładowe użycie funkcji powinno wyglądać następująco:

```
 > bezwzgledne([5, -10, 15, 0])
5 10 15 0
```

1.1.1 modyfikowanie elementów listy w trakcie iteracji

Jednak jeżeli chcemy modyfikować elementy listy iterując po niej, to konieczne jest iterowanie po indeksach (a nie jak dotychczas po wartościach):

```
for i in range(len(l)):
    print(l[i])
    l[i] = "q"
print(l)
```

```
I
C
M
['q', 'q', 'q']
```

Dzieje się tak gdyż przypisanie do zmiennej `x` jakiejś wartości w ramach konstrukcji **for** `x` **in** `lista:` modyfikuje tylko zmienną `x`, a nie element listy który został do niej pobrany.

1.2 Lista kolejnych liczb naturalnych

Często potrzebujemy, aby pętla przeszła po liście kilku kolejnych liczb naturalnych. W tym celu możemy oczywiście podać wprost kolejne elementy listy (tak jak w powyższym przykładzie), jednak istnieje wygodniejsze rozwiązanie, mianowicie polecenie **range**`()`:

```
for x in range(7):
    print(x, end = ', ')
```

```
0, 1, 2, 3, 4, 5, 6,
```

```
for x in range(5, 10):
    print(x, end = ', ')
```

```
5, 6, 7, 8, 9,
```

```
for x in range(10, 20, 3):
    print(x, end = ', ')
```

```
10, 13, 16, 19,
```

Na powyższych przykładach widzimy, że polecenie **range**`()` występuje w trzech wersjach:

- **range**`(kon)` generuje listę kolejnych liczb od 0 (**włącznie**) do `kon` (**wyłącznie**).
- **range**`(pocz, kon)` generuje listę kolejnych liczb od `pocz` (**włącznie**) do `kon` (**wyłącznie**).
- **range**`(pocz, kon, krok)` generuje listę liczb od `pocz` (**włącznie**) do `kon` (**wyłącznie**), przeskakując w każdym kroku o `krok`.

Do zapamiętania

Wszystkie przedziały w Pythonie są domknięte z lewej strony i otwarte z prawej strony, tzn. zawierają swój lewy koniec i nie zawierają swojego prawego końca.

Zadanie 1.2.1

Oblicz sumę $1^2 + 2^2 + 3^2 + \ldots + 99^2 + 100^2$.

Zadanie 1.2.2

Napisz pętlę, która wypisze wszystkie dwucyfrowe liczby podzielne przez 7. Kolejne liczby powinny być wypisane w jednym wierszu i porozdzielane pojedynczymi spacjami.

1.3 Wybór podlisty

Przy pomocy wartości podawanej w nawiasach kwadratowych po nazwie listy, możliwy jest nie tylko wybór pojedynczego elementu z listy, ale też wybór całej podlisty:

- `a[x:]` zwróci listę złożoną z elementów listy `a` od elementu x (włącznie) do końca.

- `a[x:y]` zwróci listę złożoną z elementów listy `a` od elementu x (włącznie) do elementu y (bez elementu y).
- `a[x:y:n]` zwróci listę złożoną z elementów listy `a` od elementu x (włącznie) do elementu y (bez elementu y) biorąc tylo co n-ty element.
- `a[x::n]` zwróci listę złożoną z elementów listy `a` od elementu x (włącznie) do końca biorąc tylo co n-ty element.

Jeżeli x wynosi zero (czyli chemy uzyskać pod listę od pierwszego elementu) to możemy go pominąć.

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8]
print(a[7:])
print(a[3:6])
print(a[3::2])
print(a[::-1])
```

```
[7, 8]
[3, 4, 5]
[3, 5, 7]
[8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Zwróć uwagę na podobieństwo do składni polecenia **range**().

1.4 Zadania dodatkowe

Zadanie 1.4.1

Używając dwóch pętli **for**, jedna wewnątrz drugiej, napisz program, który wypisze na ekranie *trójkąt z iksów*, taki jak poniżej:

```
X
XX
XXX
XXXX
XXXXX
XXXXXX
XXXXXXX
```

Zadanie 1.4.2

Zmodyfikuj rozwiązanie zadania 1.4.1 tak aby zamiast co najmniej jednej pętli **for** użyć pętli **while**.

Zadanie 1.4.3

Zmodyfikuj rozwiązanie zadania 1.4.1 tak aby zamiast co najmniej jednej z pętli użyć rekurencji. *Wskazówka: Funkcja rekurencyjna to funkcja, która wywołuje samą siebie (typowo ze zmodyfikowanymi argumentami), dopóki zachodzi jakiś ustalony warunek (typowo zależny od argumentów).*

2 Systemy liczbowe

Liczby mogą być zapisywane w różny sposób. Istnieją systemy addytywne (np. rzymski), w których istotna jest ilość powtórzeń danego elementu oraz systemy pozycyjne o różnych podstawach (np. system dziesiętny), w których istotne jest miejsce w którym znajduje się dany element.

W życiu codziennym najczęściej spotykamy się z zapisem dziesiętnym, funkcjonującym następująco: $5731 = 10^0 \cdot 1 + 10^1 \cdot 3 + 10^2 \cdot 7 + 10^3 \cdot 5$.

2.1 System dwójkowy

Ze względu na sposób budowy elektroniki cyfrowej i komputerów w informatyce dużo częściej spotykamy się z systemem dwójkowym (oraz systemami łatwo rozkładającymi się na dwójkowy, np. szesnastkowym).

Pojedynczą cyfrę systemu dwójkowego (przybierającą wartość 0 albo 1) określa się mianem *bit*u, liczby reprezentowane są jako ciągi takich cyfr. Terminem *bajt* określa się zazwyczaj ciąg o długości 8 bitów (ale w niektórych systemach ciąg o innej długości).

Podstawowym sposobem zapisy liczb całkowitych nie ujemnych w systemie dwójkowym jest *naturalny kod binarny* (*NKB*), w którym np. 4 bitowy ciąg $a_3a_2a_1a_0$ reprezentuje liczbę $2^0 \cdot a_0 + 2^1 \cdot a_1 + 2^2 \cdot a_2 + 2^3 \cdot a_3$. Zwróć uwagę na podobieństwo do systemu dziesiętnego.

Podstawowym sposobem zapisy liczb całkowitych (ze znakiem) jest *kod uzupełnień do dwóch* (*U2*) w którym n-bitowa liczba reprezentowana przez ciąg $a_{n-1}...a_3a_2a_1a_0$ będzie miała wartość $2^0 \cdot a_0 + 2^1 \cdot a_1 + 2^2 \cdot a_2 + ... + 2^{n-2} \cdot a_{n-2} - 2^{n-1} \cdot a_{n-1}$. Jako że najstarszy bit wchodzi z ujemną wagą, jego ustawienie na 1 oznacza liczbę ujemną (ale nie jest to kod znaku). Warto zauważyć kompatybilność z NKB.

Liczby zapisywane w tych kodowaniach systemu dwójkowego oznacza się często przy pomocy prefiksu "0b" albo sufiksu "b" (w Pythonie możemy stosować jedynie zapis z prefiksem), np. `0b101 = 101b` reprezentuje liczbę 5 w systemie dziesiętnym ($2^0 \cdot 1 + 2^1 \cdot 0 + 2^2 \cdot 1 = 5$).

2.2 Konwersje liczba – napis

Z punktu widzenia komputera liczba czy też element napisu, którym jest litera są pewną wartością numeryczną. Natomiast my do zapisu liczb używamy różnych systemów (np. dziesiętnego, czy też szesnastkowego). Domyślnie liczby wprowadzane do programu interpretowane są jako zapisane w systemie dziesiętnym, podobnie liczby uzyskiwane poprzez konwersję napisu przy pomocy funkcji **int()**. Możliwe jest jednak wprowadzanie liczb zapisanych w innych systemach liczbowych lub konwersja z napisu zawierającego liczbę — drugi, opcjonalny argument **int()** pozwala określić podstawę systemu z którego konwertujemy, zero oznacza automatyczne wykrycie w oparciu o prefix:

```
# szesnastkowo
h1, h2, h3 = 0x1F, int("0x1F", 0), int("1F", 16)
# oktalnie
o1, o2, o3 = 0o17, int("0o17", 0), int("17", 8)
# binarnie
b1, b2, b3 = 0b101, int("0b101", 0), int("101", 2)

print("",h1,o1,b1, "\n",h2,o2,b2, "\n",h3,o3,b3)
```

```
31 15 5
31 15 5
31 15 5
```

Możliwe jest także konwertowanie wartości liczbowej na napis w określonym systemie liczbowym:

```
a, b = 3, 13
c = (a + b) * b
s = "(" + bin(a) + " + " + oct(b) + ") * " + hex(b) + " = " + str(c)
print( s )
```

3 Sumator

Cyfrowe układy elektroniczne (także te składające się na nasze komputery) każdą liczbę traktują jako ciąg logicznych jedynek i zer (wartości True i False). Spróbujemy zasymulować takie działanie w Pythonie i zastanowić się nad tym jak można zrealizować dodawanie takich liczb.

Jeżeli znamy i pamiętamy jeszcze metodę dodawania „w słupku" („pod kreską") to wiemy, że dodawanie dwóch n cyfrowych liczb możemy potraktować jako ciąg n dodawań dwóch liczb jednocyfrowych z obsługą przeniesienia. Podobnie można postąpić przy sumowaniu liczb binarnych. Potrzebujemy zatem elementu, który będzie sumował 3 wartości logiczne (dwie pochodzące z sumowanych liczb, jedna z przeniesienia z poprzedniego elementu) i generował wynik sumy oraz wartość przeniesienia dla następnego elementu.

Zadanie 3.0.1

Zastanów się w jaki sposób, korzystają z poznanych funkcji logicznych (and, or, xor, not) można obliczyć wartość sumy dwóch cyfr binarnych a i b oraz wartość przeniesienia.
Wskazówka: zapisz tablicę prawdy dla tej operacji

Zadanie 3.0.2

Spróbuj rozszerzyć poprzednie rozwiązanie, tak aby uwzględniać w sumowaniu przeniesienie z poprzedniego elementu.
Wskazówka: rozszerz tablicę prawdy o jedną kolumnę wejściową

Zadanie 3.0.3

Napisz funkcję realizującą sumator. Funkcja powinna przyjmować 3 argumenty logiczne (wartości dwóch bitów do zsumowania oraz wartość przeniesienia) i zwracać wartość sumy i przeniesienia do następnego elementu.
Wskazówka: zwracanie dwóch elementów z funkcji najprościej zrealizować poprzez zwracanie dwu elementowej listy.

Liczba 6 posiada zapis binarny 0b110. Czyli jeżeli chcielibyśmy przedstawić ją w postaci listy wartości logicznych byłoby to [0, 1, 1]. Zwróć uwagę na zmianę kolejność bitów: wynika ona z tego że w zapisie list element o indeksie zero podajemy jako pierwszy a w zapisie binarnym liczby bit zerowy (wchodzący z wagą 2^0) jest jako ostatni. Dzięki takiemu zapisaniu liczb możemy użyć naszego sumatora do dodania wielo-bitowych liczb.

Zadanie 3.0.4

Napisz funkcję wykorzystującą sumator stworzony w zadaniu 3.0.3 do obliczania sumy dwóch liczb reprezentowanych jako listy wartości logicznych. Na przykład dla argumentów `[True, False, True]`, `[False, True, True]` (odpowiadających liczbom 5 i 6) wynikiem powinna być lista `[True, True, False, True]` (odpowiadająca liczbie 11).
Rozwiązanie powinno działać poprawnie dla liczb o dowolnej ilości bitów, także w przypadku gdy liczba bitów poszczególnych liczb jest różna. Rozwiązania działające tylko dla liczb o równej lub ustalonej liczbie bitów mogą otrzymać maksymalnie 2pkt.
Wskazówka: dodawanie elementu na koniec listy możliwe jest z użyciem metody `append`*, można też zedefiniować listę o zadanej ilości elementów np.* `5 * [False]` *i tylko modyfikować wybrane elementy.*

4 Głębiej w Pythona (cz. 2)

4.1 Określanie typu zmiennej

Do tej pory poznaliśmy kilka typów zmiennych w Pythonie: liczby, napisy oraz listy. Poznaliśmy także metody konwersji pomiędzy niektórymi z typów (np. instrukcje `str()`, `int()`). Jeżeli chcemy dowiedzieć się jakiego typu jest dana zmienna możemy skorzystać z funkcji `type()`:

```
a, b, c = 1, 3.14, "Python"
print(a, type(a))
print(b, type(b))
print(c, type(c))
c = (a == 1)
print(c, type(c))
```

```
1 <class 'int'>
3.14 <class 'float'>
Python <class 'str'>
True <class 'bool'>
```

Zauważ że inny typ związany jest z liczbami całkowitymi, inny z rzeczywistymi a inny z wartościami logicznymi (`True`/`False`). Zauważ także że zmienna może zmienić swój typ.

4.2 Obiektowość

Jak mogliśmy zauważyć przy sprawdzaniu typów zmiennych są one klasami. Związane z tym jest m.in. to iż posiadają one metody służące do operowania na nich. Opis danego typu wraz z dostępnymi metodami można obejrzeć przy pomocy polecenia **help**`()`, np. **help**`("list")`.

W przypadku list za pomocą metod tej klasy mamy możliwość wstawiania wartości na daną pozycję, sortowania i odwracania kolejności elementów:

```
l = ["i", "m"]
l.insert(1, "c")
print(l)
l.reverse()
print(l)
l.sort()
print(l)
```

```
['i', 'c', 'm']
['m', 'c', 'i']
['c', 'i', 'm']
```

Zwróć uwagę że sortowanie i odwracanie modyfikuje istniejącą listę a nie tworzy kopii.

Zadanie 4.2.1

Zapoznaj się z dokumentacją klasy odpowiedzialnej za napisy (**str**), zwróć szczególną uwagę na metody `split`, `find`, `replace`. Korzystając z metod klasy **str** napisz funkcję `parse` która dla napisu będącego jej argumentem wykona zamianę wszystkich ciągów "XY" na spację oraz dokona rozbicia napisu złożonego z pól rozdzielanych dwukropkiem na listę napisów odpowiadających poszczególnym polom. Funkcja powinna działać w następujący sposób:

```
 > l = parse("Ala:maXYkota:i inne:zwierzeta")
 > print(l)
['Ala', 'ma kota', 'i inne', 'zwierzeta']
```

Zadanie 4.2.2

Korzystając z metod klasy **list** i/lub funkcji **sorted**`()` napisz funkcję która sortuje podaną listę w kolejności malejącej.

Zadanie 4.2.3

Napisz funkcję `sortuj(lista)` która zwróci posortowaną listę. Funkcja nie może zmodyfikować oryginalnej listy.

4.3 Słowniki

Kolejnym użytecznym typem zmiennych w Pythonie są słowniki (zwane niekiedy *mapami* lub *tablicami asocjacyjnymi*). Podobnie jak listy służą do do przechowywania innych zmiennych. W odróżnieniu jednak od list w słownikach przechowywane są pary klucz - wartość, gdzie unikalny klucz służy do identyfikowania wartości.

```
slownik = { "bd" : "xx", 5: True, "a" : 11 }
for klucz in slownik:
    print (klucz, "=>", slownik[klucz])
```

```
a => 11
bd => xx
5 => True
```

Zauważ że zarówno klucz, jak i wartość mogą być dowolnego typu oraz że słownik nie zachowuje kolejności dodawania elementów.

Możliwe jest także sprawdzanie istnienia jakiegoś elementu w słowniku, usuwanie, dodawanie i zmienianie elementów słowniku, itd (zwróć także uwagę na inną metodę wypisywania słownika - poprzednio iterowaliśmy po kluczach, teraz po liście par klucz-wartość):

```
if "bd" in slownik:
    print ("jest element o kluczu 'bd'")
    del slownik['bd']
slownik[15] = True
slownik["a"] = "yy"
for k,v in m.items():
    print (k, "=>", v)
```

```
jest element o kluczu 'bd'
a => yy
15 => True
```

Zadanie 4.3.1

Napisz funkcję `zlicz` która dla podanej listy policzy powtórzenia jej elementów. Przykład użycia:

```
 > zlicz(["AX", "B", "AX"])
AX wystepuje 2 razy
B wystepuje 1 razy
```

Wskazówka: Użyj słownika, w którym element będzie stanowił klucz, a krotność jego wystąpień wartość. Możesz użyć metody `get()` do pobierania wartości z słownika, jeżeli w nim jest lub wartości domyślnej w przeciwnym wypadku - szczegóły zobacz w dokumentacji

Zadanie 4.3.2

Napisz funkcję która konwertuje listę napisów postaci `klucz=wartosc` na słownik. Funkcja musi dokonywać podziału napisów z listy w oparciu o pierwsze wystąpienie znaku równości przy pomocy metody `find()` typu przechowującego napisy (**str**). Funkcja musi dodawać kolejne napisy do słownika w taki sposób że część przed znakiem równości stanowi klucz, a część po znaku równości stanowi wartość.
Np. dla listy postaci: `["aa=13", "b=Ala=kot", "f=xyz"]` funkcja powinna zwrócić słownik:

```
{'b': 'Ala=kot', 'aa': '13', 'f': 'xyz'}
```

4.3.1 Sortowanie słownika

Jak już wspomnieliśmy słownik nie zachowuje porządku elementów. Jeżeli chcemy uzyskać posortowaną listę kluczy, wartości lub par klucz-wartość z słownika możemy skorzystać z funkcji **sorted**`()`. W przypadku par wywołanie będzie wyglądać następująco:

```
mapa = {'5': 3, 'bd': 20, 'a': 101}
lista = sorted( mapa.items() )
print(lista)
```

```
[('5', 3), ('a', 101), ('bd', 20)]
```

Zwróć uwagę, iż użyliśmy tej samej metody `items()`, z której korzystaliśmy do iterowania po parach klucz-wartość (dla listy samych kluczy lub wartości należy użyć w tym miejscu innej metody klasy **dict**). Zapewne zauważyłeś że sortowanie zostało przeprowadzone w oparciu o klucze, co jednak jeżeli chcielibyśmy posortować taką listę w oparciu o wartości? W takim przypadku możemy skorzystać z opcjonalnego argumentu funkcji **sorted()** o nazwie `key`, który przyjmuje funkcję mającą za zadanie na podstawie otrzymanego elementu listy (w tym wypadku pary klucz - wartość) zwrócić klucz sortowania:

```
mapa = {'5': 3, 'bd': 20, 'a': 101}
def k(x):
    return x[1]
lista = sorted( mapa.items(), key=k )
print(lista)
```

```
[('5', 3), ('bd', 20), ('a', 101)]
```

4.4 Funkcje jako argumenty funkcji

W powyższym przykładzie jednym za argumentów funkcji **sorted()** jest inna funkcja. Zauważ, że funkcja może być takim samym argumentem innej funkcji jak dowolna inna zmienna, może być też wynikiem zwracanym przez funkcję oraz może być przechowywana w zmiennej.

```
def dzialanie(operacja):
    if operacja == "dodaj":
        def f(a, b):
            return a+b
        return f
    elif operacja == "mnóż":
        def f(a, b):
            return a*b
        return f
def dwa(funkcja, argument):
    return funkcja(2, argument)

d = dzialanie("dodaj")
a = dwa(d, 11)
b = dzialanie("mnóż")(3,4)
print(a, b, d(3,4))
```

```
13 12 7
```

Zauważ że:

- wynikiem funkcji `dzialanie()` jest funkcja wykonująca wskazane działanie,
- funkcja `dwa()` jako argumenty przyjmuje funkcję realizującą działanie dwuargumentowe i jeden argument przekazywany do niej,
- zmienna d wskazuje na funkcję zwróconą przez funkcję `dzialanie()` i może być używana jako funkcja.

Zadanie 4.4.1

Zastanów się czy konstrukcję if/elif w funkcji `dzialanie()` można by zastąpić słownikiem, jak to ewentualnie zrobić i jakie mogłoby mieć to zalety bądź wady?

Zadanie 4.4.2

Napisz funkcję która przyjmuje dwa argumenty: listę oraz funkcję. Funkcja ma za zadanie wykonać przekazaną do niej funkcję na każdym elemencie listy. Przykład użycia:

```
>>> wykonaj([1,2,3], print)
1
2
3
```

4.5 Iteratory i generatory

Iterator jest obiektem pozwalającym na dostęp do kolejnych elementów jakiejś kolekcji (np. listy). Są one przydatne np. gdy chcemy uzyskiwać kolejne elementy kolekcji nie iterując po niej w ramach pętli **for**. Jego użycie wygląda następująco:

```
l = [6, 7, 8, 9]
i = iter(l)  # zmienna i jest tutaj iteratorem
print( next(i) )
print( next(i) )
```

Niekiedy zamiast tworzenia listy lepsze może być uzyskiwanie jej kolejnych elementów "na żywo". Funkcjonalność taką w pythonie zapewniają generatory. Są to funkcje które zwracają kolejne elementy danej kolekcji używając słowa kluczowego `yield`, zamiast **return**. Pamiętają one też swój stan wewnętrzny pomiędzy wywołaniami w ramach poszczególnych iteracji.

Generatory możemy używać np. do iterowania po nich w pętli **for**, możemy tez używać iteratorów do pobierania kolejnych wartości z generatora:

```
def f(l):
    a, b = 0, 1
    for i in range(l):
        r, a, b = a, b, a + b
        yield r

ii = iter( f(8) )
for i in f(16):
    print("i =", i)
    if i > 6:
        print("ii =", next(ii))
```

Można także tworzyć generatory nieskończone:

```
def ff():
    a, b = 0, 1
    while True:
        r, a, b = a, b, a + b
        yield r
```