

# Kurs rozszerzony języka Python

## Lista 4.

Na poniższej liście są dwa rodzaje zadań. Proszę wybrać po jednym zadaniu z każdej grupy i je zaprogramować. Każde z tych zadań jest warte 4 punkty.

### Listy

Poniżej są zadania polegające na implementacji funkcji zwracających listy liczb naturalnych spełniających odpowiednie warunki. Każde z zadań należy wykonać w trzech wersjach: w wersji imperatywnej, w wersji z listą składaną i wersję funkcyjną:

- w wersji imperatywnej korzystamy z instrukcji `while`, `for in` etc. i uzupełniając listę wynikową metodą `append`;
- Wersja z *listą składaną* powinna być w postaci jednej listy składanej bądź zagnieżdżonych list składanych. W przypadku zagnieżdżenia można wydzielić podlisty np. tak:

```
def zadana_funkcja(n):
    lista_tymcz = [ lista skladana ]
    return [ lista_skladana_zawierajaca lista_tymcz ]
```

- *Implementacja funkcyjna* powinna korzystać z funkcji dedykowanych do operacji na listach (lub na generatorach list): `filter`, `range`, `sum` czy `reduce`. Tu zaznaczam, że funkcja ma finalnie zwrócić listę, nie generator.

Wykorzystując moduł `timeit` zbadaj dla różnych danych, jaki jest czas działania poszczególnych funkcji. Pomiary czasu sformatuj w postaci czytelnej tabelki w rodzaju

n	skladana	imperatywna
10:	0.018	0.008
20:	0.042	0.016
30:	0.074	0.024
40:	0.111	0.032
50:	0.155	0.040
60:	0.204	0.048
70:	0.261	0.057
80:	0.326	0.065
90:	0.394	0.073

### Zadanie 1.

Zaprogramuj jednoargumentowe funkcje `pierwsze_imperatywna(n)`, `pierwsze_skladana(n)` i `pierwsze_funkcyjna(n)`, które zwracają listę liczb pierwszych nie większych niż  $n$ , na przykład

```
>>> pierwsze(20)
[2, 3, 5, 7, 11, 13, 17, 19]
```

### Zadanie 2.

Zaprogramuj jednoargumentowe funkcje `doskonale_imperatywna(n)`, `doskonale_skladana(n)` i `doskonale_funkcyjna(n)`, które zwracają listę liczb doskonałych nie większych niż  $n$ , na przykład

```
>>> doskonale(10000)
[6, 28, 496, 8128]
```

### Zadanie 3.

Zaprogramuj jednoargumentowe funkcję `rozklad_imperatywna(n)`, `rozklad_skladana(n)` i `rozklad_funkcyjna(n)` które obliczają rozkład liczby  $n$  na czynniki pierwsze i zwracają jako wynik listę par  $[(p_1, w_1), (p_2, w_2), \dots, (p_k, w_k)]$  taką, że  $n = p_1^{w_1} * p_2^{w_2} * \dots * p_k^{w_k}$  oraz  $p_1, \dots, p_k$  są różnymi liczbami pierwszymi. Na przykład

```
>>> rozklad(756)
[(2, 2), (3, 3), (7, 1)]
```

Ponieważ w tym zadaniu może być potrzebna lista liczb pierwszych, można zaimplementować pomocniczą funkcję sprawdzającą pierwszość liczby bądź zwracającą listę liczb pierwszych. W przypadku tej funkcji pomocniczej implementacja może być dowolna.

### Zadanie 4.

Zaprogramuj jednoargumentowe funkcje `zaprzyjaznione_imperatywna(n)`, `zaprzyjaznione_skladana(n)` i `zaprzyjaznione_funkcyjna(n)`, które zwracają listę par liczb zaprzyjaznionych nie większych niż  $n$ , na przykład

```
>>> zaprzyjaznione(1300)
[(220, 284), (1184, 1210)]
```

Odpowiednie definicje można znaleźć np. w polskiej Wikipedii. Wybierz dwa z podanych zadań. Każde zadanie jest warte 4 pkt.

### Rozwiązanie zadań

Zaprogramuj wyszukiwanie rozwiązań oparte na sprawdzaniu wszystkich potencjalnych rozwiązań (*brute force*). Funkcja rozwiązująca zadanie powinna zwracać iterator, tak aby można było wypisać wszystkie znalezione rozwiązania wykorzystując instrukcję `for-in`:

```
for rozwiazanie in rozwiazywanie_zadania(dane_wejsciowe):
    print(rozwiazanie)
```

Zadbaj o to, aby nie generować niepotrzebnie list (np. z permutacjami).

Można założyć, że dane wejściowe są zawsze poprawne i nie trzeba ich dodatkowo sprawdzać.

### Zadanie 5.

















*Kryptarytm* to zadanie, w którym litery należy zastąpić cyframi tak, aby powstało poprawne działanie. Przykładem takiego kryptarytmu jest

```
  KIOTO
+ OSAKA
-----
  TOKIO
```

Napisz program rozwiązujący takie kryptarytmy. Przyjmij, że dane wejściowe zawierają trzy słowa i operator.

### Zadanie 6.

Poniższe zadanie polega na rekonstrukcji dwuwymiarowego obrazu na podstawie rzucanego cienia. Zakładamy, że obraz jest prostokątem czarno-białych pikseli. Cień to dwa wektory, opisujące ile jest zaczerwionych pikseli w wierszu bądź kolumnie. Poniżej przykład obrazu rozmiaru  $4 \times 4$ :

2	1	3	1	
				1
				3
				1
				2

którego cień opisują dwa wektory:  $H = (2, 1, 3, 1)$ ,  $V = (1, 3, 1, 2)$ . Dla danego cienia może istnieć wiele różnych obrazów.

### Zadanie 7.

W popularnej łamigłówce *sudoku* zadanie polega na wypełnieniu diagramu  $9 \times 9$  cyframi od 1 do 9 tak, aby w każdym wierszu i każdej kolumnie żadna cyfra się nie powtarzała. Dodatkowo, w każdym podkwadracie  $3 \times 3$  nie może powtarzać się żadna cyfra. Poniżej jest przykład prawidłowo wypełnionego diagramu:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Zaprogramuj funkcję `rozwiązanie_sudoku(s)` która dla częściowo wypełnionego diagramu  $s$  zwraca poprawne jego wypełnienie (bądź **None** gdy nie ma rozwiązania). Reprezentacja diagramu jest dowolna.

Zaprogramuj również funkcję, która wyświetli czytelnie diagram.

*Marcin Młotkowski*