Kolekcje i funkcje

Kolekcje

Python nie jest oczywiście pierwszym językiem, w którym są kolekcje – np. biblioteki standardowe C/C++ (C – 1972, C++ – 1983) mają po kilka rodzajów kolekcji, w postaci szablonów klas. Python jest językiem zorientowanym na przetwarzanie kolekcji, jest to spuścizna języków SETL i ABC, z których się wywodzi:

- kolekcje dzielą się na sekwencje (indeksowane) i zbiory; indeksowanie [ ] działa dla wszystkich sekwencji, w tym łańcuchów
- cztery rodzaje kolekcji są wbudowane w składnię języka i można je utworzyć używając nawiasów: [ ], ( ) oraz { }, są to lista (list), krotka (tuple), słownik (dictionary) oraz zbiór (set); zamiast nawiasów można użyć funkcji: list(), tuple(), dict(), set()
- niektóre funkcje wbudowane, np. print, all, filter, len, range, operują na kolekcjach lub są z nimi powiązane
- instrukcja iteracyjna for operuje na kolekcjach

Oprócz 4 kolekcji podstawowych jest jeszcze wiele innych, np. Array, Queue, Counter, OrderedDict, ChainMap, …

Podstawowe cechy kolekcji

Podstawowe cechy kolekcji:

- mutable – czy można zmieniać wartości elementów kolekcji
- ordered – czy elementy kolekcji zachowują kolejność
- unique – czy elementy mogą się powtarzać w kolekcji

Są też specjalne wersje kolekcji przeznaczone do programowania współbieżnego (wielowątkowego), "thread save"

Różnice między kolekcjami mają wpływ np. na indeksowanie: w słowniku można dodać element przez indeksowanie, w liście nie; w liście indeksowanie umożliwia modyfikację elementu, w krotce nie. Największa jest różnica pomiędzy zbiorami (set i frozenset) a kolekcjami – dla zbiorów nie jest zdefiniowany operator indeksowania, są też dostępne inne operatory (binarne zamiast arytmetycznych) oraz metody (np. add zamiast append)

Lista (list)

Cechy: mutable, ordered, not unique

Najczyściej wykorzystywana kolekcja Pythona. Jest najbliższym odpowiednikiem tablic innych języków, jednak może zawierać elementy różnych typów – również inne kolekcje, tzn. kolekcje mogą być zagnieżdżone (nested). Lista jest indeksowana – indeksami są liczby całkowite, a pierwszym indeksem jest 0 (jak w tablicach języków C-podobnych)

```
l1 = [ [], 1, 2.0, "trzy", ['a', 'b', 'c', 'd'] ]
l1[1] = 13

s1 = { 'a', 'b', 'c' }
l2 = list(s1)          # ['a','b','c']

l3 = list(range(3))    # [0,1,2]
```

```
l4 = l2 + l3 * 2       # ['a','b','c', 0,1,2, 0,1,2]
```

Metody list:
– append – dodaje element do listy
– clear – usuwa wszystkie elementy
– count – zlicza elementy o wskazanej wartości
– index – pozycja elementu na liście
– remove – usuwa element z listy, zwraca None
– pop – usuwa element z listy, ze wskazanej pozycji, i zwraca go
– insert – wstawia element na wskazaną pozycję

```
l = [ 'a', 'b', 'c', 'a' ]

l.count('a')   # 2
l.index('b')   # 1
l.append('x')  # l = [ 'a', 'b', 'c', 'a', 'x']
l.remove('c')  # l = [ 'a', 'b', 'a', 'x']
e = l.pop(1)   # 'b', l = [ 'a', 'a', 'x']
```

Krotka (tuple)

Cechy: immutable, ordered, not unique

Krotka to niezmienna lista (rekord danych). Ma wszystkie cechy listy, oprócz możliwości modyfikacji elementów: indeksowanie (również slicing) służy tylko do odczytu wartości elementów krotki, natomiast nie ma metod append, insert, clear. Jeżeli element krotki jest modyfikowalną kolekcją, to nie można go podmienić (zastąpić czymś innym), ale można zmieniać jego zawartość.

Krotkę tworzą przecinki, a nie nawiasy np. (7, 13) – krotka, podobnie 7, 13 – krotka, ale przy wywołaniu funkcji przecinki służą do rozdzielenia argumentów – wtedy krotkę tworzą i nawiasy, i przecinki: f(7, 13) – dwa argumenty funkcji, f((7, 13)) – jeden argument, krotka. Nawiasy ( ) są również stosowane w wyrażeniach – to znaczenie nawiasów ma pierwszeństwo: (7) – liczba, (7,) – krotka; Wyjątkiem są puste nawiasy: () – pusta krotka

Metody krotek:
– count – zlicza elementy o wskazanej wartości
– index – pozycja elementu na liście

```
t1 = ('jeden', 'dwa', 'trzy')
t2 = ('cztery', ) # ważne: bez przecinka powstaje łańcuch
t3 = t1 + t2*3

print(t1[1])        # 'dwa'
t1[1] = 'DWA'       # błąd!

t3.count('cztery')  # 3
t3.index('jeden')   # 0
```

Słownik (dictionary)

Cechy: mutable , unordered, unique

Słownik języka Python to tablica asocjacyjna. Elementami słownika są pary klucz- wartość:

- kluczem może być tylko element "niemodyfikowalny", liczba (int, float lub complex), łańcuch znaków lub krotka
- wartość może być dowolnego typu

Kolejność elementów w słowniku może być różna od tej, w jakiej elementy są dodawane (zależy to od wartości innych zmiennych). Przez indeksowanie można zmieniać wartość elementu przypisanego do klucza albo dodać klucz z przypisana wartością

```
d = { 'a': 1 }
d['a'] = 13     # modyfikacja elementu, d = {'a':13}
d['b'] = 77     # dodanie elementu, d = {'a':13, 'b':77}
print(d['x'])   # błąd!
```

Metody słownika:
– fromkeys – tworzy słownik z kolekcji, biorąc elementy kolekcji jako klucze i przypisując wszystkim wartość None
– get – zwraca wartość podanego klucza lub None gdy klucza nie ma
– pop – usuwa ze słownika wskazany klucz i zwraca wartość, która była przypisana do tego klucza
– update – dodaje do słownika elementy ze wskazanej kolekcji; kolekcja ta musi być słownikiem albo kolekcją składającą się wyłącznie z dwuelementowych kolekcji (np. listą krotek)

```
d1 = { 'dwa':2, 'trzy':3 }
d1.get('cztery')     # None; d1['cztery'] rzuci wyjątek
d1.pop('dwa')        # 2; d1 = {'trzy':3 }
d1.update(('trzy','three'))   # błąd!
d1.update([('trzy','three')]) # zastąpi istniejący klucz

d2 = dict((('trzy', 'three'), ('pięć', 'five')))
```

Zbiór (set)

Cechy: mutable , unordered, unique

Zbiór w języku Python reprezentuje teoriomnogościowy zbiór matematyczny, jest to modyfikowalny zbiór niemodyfikowalnych elementów. Elementem zbioru może być zatem tylko element "niemodyfikowalny", liczba (int, float lub complex), łańcuch znaków lub krotka. Ze względu na użycie nawiasów klamrowych "{: oraz "}" również dla słowników, pusty zbiór można utworzyć tylko funkcją set() – puste nawiasy klamrowe to pusty słownik.

Zbiory nie są indeksowane, jak to zbiory, za to mają specyficzne operatory:

```
|  suma
&  przecięcie (część wspólna)
-  różnica
^  dopełnienie (różnica symetryczna)
```

Znaczenie i sposób działania poszczególnych operatorów pokazują przykłady:

```
A = { 1, 4 }; B = { 4, 7 };
A | B    # { 1, 4, 7 }          A & B    # { 4 }
A - B    # { 1 }                B - A    # { 7 }
A ^ B    # { 1, 7 }
```

Metody zbiorów:
– add – dodaje wskazany element
– remove – usuwa wskazany element (rzuca wyjątek, gdy go brak)
– discard – usuwa wskazany element, o ile taki istnieje
– clear – usuwa wszystkie elementy
– update – dodaje do zbioru elementy z innych kolekcji
– issubset – sprawdza czy zbiór jest podzbiorem wskazanego zbioru
– isdisjoint – sprawdza czy zbiór jest rozdzielny ze wskazanym zbiorem

```
A = { 3, 7, 13 }
B = set()
D = { 1: 'jeden' }

B.add(8)         # B = { 8 }
A.isdisjoint(B)  # True
A.update(D)      # A = { 1, 3, 13, 7 }
```

Funkcje wbudowane i kolekcje

Spora część funkcji wbudowanych operuje na kolekcjach, m.in.:

- all – True, jeżeli wszystkie elementy kolekcji są True
- any – True, jeżeli choć jeden element kolekcji jest True
- min – najmniejsza wartość z kolekcji
- max – największa wartość z kolekcji
- sorted – lista posortowana, utworzona z kolekcji
- sum – suma elementów kolekcji
- len – długość kolekcji

Większość tych funkcje przyjmuje jako argument obiekt "iterable" (iterowalny, tj. taki, dla którego zdefiniowany jest iterator); Iterable jest pojęciem szerszym, niż kolekcja, obejmuje również wycinanie (slicing), wyrażenia listowe, wyrażenia generatorów, funkcje generujące oraz klasy ze zdefiniowanym iteratorem

Slicing

Wycinanie (slicing) jest rozszerzeniem indeksowania, umożliwia wybieranie wielu elementów kolekcji. Działa na kolekcjach uporządkowanych – lista, krotka, łańcuch znaków. Notacja:

```
<kolekcja> [ <start> : <stop> : <step> ]
```

przy czym domyślnie start = 0, stop = rozmiar kolekcji, który można odczytać funkcją len() , natomiast step = 1; Dowolne z tych elementów można pominąć, jednak aby ich interpretacja była jednoznaczna, pozostawia się dwukropki, np. `lista[::2]`
Parametr <stop> jest specyficznie rozumiany: wynik zawiera elementy o indeksach od <start> do <stop>-1 (trochę jak w pętli for w C#). Można stosować wartości ujemne: <start> i <stop>, co oznacza indeksy liczone od końca (ostatni element: -1, przedostatni: -2, …), natomiast nie oznacza, że elementy są zwracane w odwrotnej kolejności – o tym decyduje wartość <step>. Ujemny <step> oznacza zmniejszanie indeksów elementów kolekcji, zamiast zwiększania.

```
ss = '0123456789'
ss[::2]      # '02468'
ss[1:8:3]    # '147'
ss[1:6]      # '12345'
ss[6:1:-1]   # '65432'
```

```
ss[-4:-1]    # '678'
ss[::-1]     # '9876543210'
```

Konwersje kolekcji

Kolekcje można konwertować na inne kolekcje, używając wbudowanych funkcji

- Funkcje list(), tuple(), set()
  Każda z tych funkcji otrzymuje jako argument kolekcję i tworzy na jej podstawie nową kolekcję, typu zgodnego z nazwą funkcji. Ich działanie zależy od typu kolekcji docelowej, np. set() usunie duplikaty. Jeżeli argumentem jest słownik, to do nowej kolekcji zostaną wzięte tylko klucze, a nie wartości.
- Funkcja dict()
  Tworzy słownik tylko z kolekcji (listy lub krotki) zawierającej dwuelementowe listy lub krotki, np. d = dict([[1,'1'], [2,'2']]); Funkcja enumerate() zamienia listę lub krotkę na listę par indeks-wartość, co umożliwia przekształcenie listy w słownik

```
ll = list('abc')            # ['a', 'b', 'c']
tt = tuple(ll)              # ('a', 'b', 'c')
dd = dict(enumerate(ll))    # {0: 'a', 1: 'b', 2: 'c'}
ll = list(dd)               # [0, 1, 2]
```

Wyrażenia listowe i wyrażenia generatorów

Wyrażenia listowe

Wyrażenia listowe (inaczej rozwinięcia list) to konstrukcja składniowa umożliwiająca tworzenie list na podstawie kodu, a nie wartości; rodzaj pętli for użytej inline.

Składnia:

```
<lista> = [<wyrażenie> for <zmienna> in <kolekcja> if <warunek>]
```

Pętla for pobiera do <zmiennej> elementy <kolekcji>, następnie <wyrażenie> wykorzystując <zmienną> oraz inne dostępne w tym miejscu programu obiekty oblicza wartość, która jest dodawana do listy, o ile spełniony jest <warunek>; Zamiast kolekcji można użyć generatora, jak range() albo enumerate()

```
lista = [ i**2 for i in range(10)]  # 0, 1, 4, 9, …

krotka = ('zero', 'jeden', 'dwa')
lista = [ word for word in krotka if len(word)>3]
```

Wyrażenia listowe można użyć również do tworzenia innych kolekcji, które są modyfikowalne – np. słowników i zbiorów, z ograniczeniami tych kolekcji (duplikaty zostaną usunięte)

```
krotka = ('zero', 'jeden', 'dwa')
zbiór = { word for word in krotka }
słownik = { len(word):word for word in krotka }
```

Wyrażenia generatorów

Wyrażenia generatorów to konstrukcja składniowa podobna, jak wyrażeń listowych, jednak działanie jest zupełnie inne: tworzony jest generator – czyli kod generujący kolejne wartości kolekcji, a nie gotową listą wartości. Utworzony w ten sposób generator można użyć zamiast kolekcji we wszystkich wyrażeniach i instrukcjach Pythona, np. w pętli for

```
lista = [ i**2 for i in range(1,5) ]
print(lista) # [1, 4, 9, 16, 25]

gen = ( i**2 for i in range(1,10) )
print (gen) # <generator object <genexpr> at 0x02B77A00>

for n in gen:
    print(n)
```

Generatory wykorzystuje się przede wszystkim po to, aby oszczędzić miejsce w pamięci operacyjnej. Generator jest tylko małym fragmentem kodu, podczas gdy lista wartości może zająć spory obszar RAM:

```
ośCzasu = (0.001*t for t in range(100000))
```

Funkcje

Python wspiera metodykę proceduralną, zatem funkcje nie muszą być deklarowane wyłącznie jako elementy składowe klas. Do definiowania funkcji służy słowo kluczowe (w Pythonie def) – jest to charakterystyczne dla języków skryptowych, ze względu na typowanie dynamiczne. Do zwracania rezultatu służy instrukcja return, jednak nie jest ona obowiązkowa (domyślnie funkcja zwraca wartość None).

Przykład:

```
def funkcja(a, /, b=2, *, c=3):
    return a + b + c

x1 = funkcja(1)
x2 = funkcja(1, 7)
x3 = funkcja(1, c=13, b=7)
```

Argumenty

Python ma rozbudowany system definiowania argumentów – pozycyjnych, nazwanych, domyślnych oraz argumentów zmiennych, nazwanych i nienazwanych:

- Argumenty, które mają wartość domyślną, można pominąć przy wywołaniu
- Argumenty nazwane można podawać w dowolnej kolejności
- Jeżeli nie jest to jawnie wskazane przez znaki "/" i "*", przy wywołaniu można stosować argumenty pozycyjne i nazwane (ale po pierwszym nazwanym tylko nazwane)
- Symbole specjalne, których można użyć w liście argumentów:
  * – miejsce, do którego można stosować tylko argumenty pozycyjne
  / – miejsce, od którego można stosować tylko argumenty nazwane

Argumenty zmienne

Argumenty zmienne funkcji (ang. arbitrary arguments, nazwa polska jest nieco myląca) to mechanizm umożliwiający przekazanie do funkcji dowolnej liczby argumentów.
Argumenty zmienne mogą być nienazwane lub nazwane:

- *args – nienazwane, zamieniane przy wywołaniu w krotkę
- **kwargs – nazwane, zamieniane przy wywołaniu w słownik par nazwa-wartość

Argumenty zmienne można łączyć ze zwykłymi, podobnie jak zmienne nienazwane i nazwane, o ile nienazwane będą zawsze przez nazwanymi, a zwykłe przed zmiennymi

```
def f1(*a, **b):
    pass

f1(1, 2, 3, x=7, y=13)

def f2(a, *b, c=7, **d):
    pass

f2(1, 2,3,4,5, x=7,  y=13, c=77)
```

Ciekawy i zarazem funkcjonalny przykład kombinacji argumentów zmiennych nienazwanych oraz zwykłych nazwanych, z wartościami domyślnymi, dostarcza wbudowana funkcja print:

```
print(*objects, end='\n', sep=' ')
```

Dodatkowe informacje

- Funkcje nie mogą być przeciążone – ze względu na dynamiczne typowanie
- Funkcje mogą być rekurencyjne
- Funkcje są w Pythonie obiektami – można je zapisać w zmiennej lub przekazać jako argument do innej funkcji; Zmienną (argument funkcji) zawierającą funkcję można wywołać tak samo jak oryginalna funkcję

```
def silnia(n):
    return 1 if n<2 else n*silnia(n-1)

fs = silnia

fs(5)      # 120
print(fs)  # <function silnia at 0x02B37A90>
```

Wyrażenia Lambda

Wyrażenia lambda są prostymi funkcjami anonimowymi. Do definiowania wyrażeń lambda służy słowo kluczowe (lambda), z tych samych względów co słowo kluczowe def w przypadku funkcji. Wyrażenie lambda może mieć dowolną liczbę argumentów, ale może zawierać tylko jedno wyrażenie (nawet nie instrukcję, jak if lub for). Wykorzystywane są najczęściej przy wywoływaniu funkcji, które otrzymują funkcję jako argument

```
kwadrat = lambda x: x**2
potęga = lambda p,w: p**w

def całka(f, a=0.0, b=1.0, n=100):
    # tu można użyć funkcji f
    return …

c1 = całka(kwadrat)
c2 = całka(lambda x: x**4)
```

Generatory

Funkcja generująca to funkcja, która używa yield zamiast return. Instrukcja yield może być użyte w funkcji wielokrotnie (również w pętli): każde yield zwraca wynik, ale nie przerywa wykonania funkcji. Kiedy generator pobiera kolejną wartość, funkcja jest wznawiana aż do kolejnego użycia yield

Funkcja generująca może być użyta jako część wyrażenia listowego lub generującego albo w pętli for

```
def simpleRange(n):
    while n>0:
        yield n
        n -= 1

lista = [k**2 for k in simpleRange(3)] # [9, 4, 1]

for i in simpleRange(5):
    print(i**3)              # 125, 48, 27, 8, 1
```

Funkcje wbudowane dla kolekcji – uzupełnienie

Jak już wspomniano, spora część funkcji wbudowanych operują na kolekcjach, m.in. all, any, min, max, sum – i inne. Większość tych funkcje przyjmuje jako argument obiekt "iterable" (iterowalny, tj. taki, dla którego zdefiniowany jest iterator), przy czym Iterable jest pojęciem szerszym, niż kolekcja, obejmuje również wycinanie (slicing), wyrażenia listowe, wyrażenia generatorów, funkcje generujące oraz klasy ze zdefiniowanym iteratorem.

Niektóre z tych funkcji przyjmują dodatkowy argument w postaci funkcji, która z kolei otrzymuje jako argument pojedynczy element kolekcji i zwraca jego "metrykę"; Owa metryka np. przy sortowaniu służy jako kryterium sortowania. Dzięki takiej konstrukcji, funkcje wbudowane kolekcji dają programiście elastyczne i uniwersalne narzędzia przetwarzania kolekcji, co zostanie pokazane na kilku przykładach.

- Lista jest konwertowana na wyrażenie generatora; można dowolnie określić sposób konwersji, zaś funkcja all sprawdza czy wynik konwersji zawsze jest True

  ```
  lista = [ 'zero', 'jeden', 'dwa' ]
  nieKrótkie = all( len(x)>3 for x in lista )  # False
  ```

- Funkcja min porównuje wartości dostarczane przez wyrażenie generatora, czyli drugie elementy krotek; zwraca najmniejszą wartość, chociaż nie wiadomo z której krotki

  ```
  liczby = [ ('siedem', 7), ('cztery', 4) , ('pięć', 5) ]
  minVal = min(  krotka[1] for krotka in liczby )  # 4
  ```

- Funkcja min porównuje metryki zwracane przez wyrażenie lambda; zwraca całą krotkę, dla której metryka jest najmniejsza

  ```
  liczby = [ ('siedem', 7), ('cztery', 4) , ('pięć', 5) ]
  minTup = min(liczby, key = lambda krotka: krotka[1])
  ```

- Przykład podobny do poprzedniego, ale operujący na słowniku. Funkcja min porównuje metryki i zwraca klucz słownika, pod którym wartość jest najmniejsza:

  ```
  liczby = { 'siedem':7, 'dwa':2, 'cztery':4, 'trzy':3 }
  minDic = min( liczby, key = lambda k: liczby[k] )  #'dwa'
  ```

- Podobnie można użyć funkcji sorted – zależnie od konstrukcji metryki uzyskuje się różne kryteria sortowania dowolnej kolekcji, ale zawsze przekonwertowanej na listę – tu sorted dostaje krotkę krotek, a zwraca listę krotek:

```
liczby=(('cztery',4),('jeden',1),('trzy',3),('sto',100))

# według wartości liczb (1, 3, 4, 100)
l1 = sorted(liczby, key = lambda krotka: krotka[1])

# alfabetycznie według opisów (4, 1, 100, 3)
l2 = sorted(liczby, key = lambda krotka: krotka[0])

# według długości opisów (100, 3, 1, 4)
l3 = sorted(liczby, key = lambda krotka: len(krotka[0]))
```

Pomiar czasu:

Sposób pomiaru czasu wykonywania fragmentu programu

```
from datetime import datetime


s = datetime.now()

# działania, których czas wykonania jest mierzony

d = datetime.now() - s;
print(f'Czas obliczeń: {d}')
```

Zadania

Proszę napisać program, który…

1. Wyświetla w zgrabnej tabelce wartości funkcji sinus i cosinus, dla kątów w zakresie od 0 do 90 stopni, co 10 stopni. Funkcje trygonometryczne posługują się kątami wyrażonymi w radianach, należy zatem przeliczyć stopnie na radiany. Warianty:
   a. Bezpośrednio, przeliczenie stopnie-radiany w wywołaniu funkcji
   b. Z użyciem własnej funkcji stopnie-radiany, math.sin(Deg2Rad(fi))
   c. Z użyciem własnej funkcji przyjmującej kąt w stopniach, SinFmDeg(fi)
2. Oblicza przy pomocy funkcji i wyświetla pierwiastki równania kwadratowego. Funkcja powinna zwracać krotkę z wartościami pierwiastków – dwoma, jednym lub pustą: print(pierwiastki(2, -10, 12))
3. Wykonuje zadania z punktu 2, ale sprawdza długość krotki i drukuje odpowiedni komunikat oprócz (albo zamiast) wartości pierwiastków. Wskazówka: długość kolekcji podaje wbudowana funkcja len
4. Wyświetla wartości silni dla liczb z zakresu podanego przez użytkownika. Do obliczenia silni należy zdefiniować funkcję
5. Wykonuje zadania z punktu 4, ale funkcja silnia jest rekurencyjna
6. Znajduje n-ty element ciągu Fibonacciego. Wskazówka: można wykorzystać kod z ćwiczenia z pętli, trzeba go tylko przerobić na funkcję
7. Wykonuje zadania z punktu 6, ale funkcja Fibonacci jest rekurencyjna
8. Znajduje NWD dwóch liczb.
9. Wykonuje zadania z punktu 8, ale funkcja NWD jest rekurencyjna
10. Znajduje parę liczb względnie pierwszych (NWD=1) z zakresu liczb podanych przez użytkownika – wariant stochastyczny (obie liczby są losowane, do skutku). Do wyznaczania NWD należy użyć funkcji z punktu 5. Wskazówka: losową liczbę całkowitą wyznacza funkcja randint z modułu random
11. Oblicza przybliżoną wartość całki, obliczaną metodą prostokątów. Należy zdefiniować funkcję obliczającą przybliżenie całki, przyjmującą 4 argumenty, np. całka(f, a, b, n) Argumentami powinny być: funkcja całkowana f, granice przedziału całkowania a i b oraz liczba prostokątów n. Warianty:
    a. Funkcja podcałkowa definiowana jako tradycyjna funkcja
    b. Funkcja podcałkowa w postaci wyrażenia lambda
12. Zmienić definicję funkcji z punktu 11 tak, aby n miało wartość domyślną, np. 1000
13. Zmienić definicję funkcji z punktu 11 tak, aby n trzeba było podawać jako argument nazwany
14. Oblicza przybliżoną wartość pierwiastka równania nieliniowego metodą bisekcji. Należy zdefiniować funkcję obliczającą przybliżenie pierwiastka, np. zero(f, a, b, e), której argumentami powinny być: funkcja nieliniowa f, początkowe granice przedziału a i b oraz dokładność e. Warianty: jak w punkcie 11

15. Zmienić definicję funkcji z punktu 14 tak, aby e miało wartość domyślną, np. 1e-6 i trzeba było podawać e jako argument nazwany
16. Zmienić definicję funkcji z punktu 15 tak, żeby sama znajdowała granice (a, b), np. ±1, ±10, ±100, …).
17. Wyświetla wartości x w zakresie od 0 do 2 włącznie, z krokiem co 0.1. Należy zdefiniować własną funkcję generatora floatRangeH(a, b, h), gdzie a i b to granice przedziału, zaś h to krok. W definicji funkcji należy użyć pętli while oraz instrukcji yield
18. Zmienić definicję funkcji z punktu 17 tak, aby można ją było wywołać również z dwoma argumentami, floatRangeH(b, h) – wówczas funkcja powinna przyjąć wartość a=0. Wskazówka: należy oczywiście użyć argumentu domyślnego, ale w nietypowy sposób, ponieważ normalnie wartość domyślna dotyczy ostatniego, a nie pierwszego argumentu. Można np. wykorzystać jako wartość domyślną None, trzeba jej tylko sprytnie użyć. Wszystkie argumenty powinny być pozycyjne, a nie nazwane.
19. Zmienić definicję funkcji z punktu 18 tak, aby można ją było wywołać również z jednym argumentem, floatRangeH(b) – wówczas funkcja powinna przyjąć wartość a=0, h=b/10
20. Wyświetla wartości x w zakresie od 0 do 2 w 10 krokach, (bez 2, tj. 0, 0,2, 0,4, …, 1,8). Należy zdefiniować własną funkcję generatora floatRangeN(a, b, n), gdzie a i b to granice przedziału, zaś n to liczba kroków. W funkcji należy użyć pętli for oraz instrukcji yield
21. Zmienić definicję funkcji z punktu 20 tak, aby dodatkowy, nazwany argument decydował, czy wygenerowane wartości mają zawierać górną granicę przedziału (a, b), czy nie, np. floatRangeN(a, b, n, upperLimit=False)
22. Wyświetla w zgrabnej tabelce wartości wybranej funkcji, np. exp(-x), dla x w zakresie od 0 do 2 co 0.1. Należy użyć jednej z wersji funkcji generatora z punktu 20 lub 21
23. Wyświetla liczby pierwsze ze wskazanego przedziału. Należy zdefiniować własną funkcję generatora Primes(a, b), gdzie a i b to granice przedziału. W funkcji należy użyć pętli for oraz kombinacji instrukcji warunkowej i yield. Zastosować najprostszy test pierwszości, tj. policzenie wszystkich naturalnych podzielników i sprawdzenie, czy są dokładnie 2
24. Zmodyfikować funkcję z punktu 23 tak, aby przyjmowała dodatkowy argument, tj. liczbę licz pierwszych, po której osiągnięciu funkcja zaprzestaje szukania: Primes(a, b, nmax). Argument ten powinien mieć wartość domyślną, a w przypadku jego pominięcia funkcja powinna zwrócić wszystkie liczby pierwsze z przedziału (a, b)
25. Wyświetla liczbę liczb pierwszych ze wskazanego przedziału, np. (10000, 20000). Zmierzyć czas szukania liczb pierwszych, a następnie opracować alternatywną funkcję generatora z punktu 23, aby działała szybciej – przez sprawdzanie podzielników od 2 do pierwiastka ze sprawdzanej liczby i tylko do pierwszego znalezionego podzielnika.
26. Jak w punkcie 25, ale z użyciem sita Eratostenesa. Wskazówka: korzystając z funkcji z punktu 24 lub 25 wygenerować listę liczb pierwszych z przedziału (2, pierwiastek(b)). Następnie postępować jak w punkcie 25, ale sprawdzać jako potencjalne podzielniki tylko liczby pierwsze z wygenerowanej w kroku poprzednim listy. Trzeba oczywiście zmienić konstrukcję pętli do sprawdzania podzielników;