Programowanie funkcyjne

Listy Składane

Listy składane (ang. list comprehension):

- pozwalają na tworzenie nowych list na podstawie już istniejących w formie zwięzłego wyrażenia.

Listy Składane

Przykład 1:

```
>>> liczby=range(-20,20)
>>> liczby
range(-20, 20)
>>> liczby=list(liczby)
>>> liczby
[-20, -19, -18, -17, -16, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5, -4,
-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
]
>>> liczby_i_kwadraty=[(x,x**2) for x in liczby]
>>> liczby_i_kwadraty
[(-20, 400), (-19, 361), (-18, 324), (-17, 289), (-16, 256), (-15, 225), (-14, 1
96), (-13, 169), (-12, 144), (-11, 121), (-10, 100), (-9, 81), (-8, 64), (-7, 49
), (-6, 36), (-5, 25), (-4, 16), (-3, 9), (-2, 4), (-1, 1), (0, 0), (1, 1), (2,
4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100), (11
, 121), (12, 144), (13, 169), (14, 196), (15, 225), (16, 256), (17, 289), (18, 3
24), (19, 361)]
```

Listy Składane

Przykład 1 (cd.):

```
>>>
>>> kwadraty_parzyste=[x**2 for x in liczby if x%2==0]
>>> kwadraty_parzyste
[400, 324, 256, 196, 144, 100, 64, 36, 16, 4, 0, 4, 16, 36, 64, 100, 144, 196, 2
56, 324]
>>> liczby_dodatnie=[x for x in liczby if x > 0]
>>> liczby_dodatnie
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>>
```

Wyrażenia Lambda

Wyrażenia lambda:

- to rodzaj krótkich, jednolinijkowych, anonimowych (bez nazwy) funkcji,
- można je przypisywać do zmiennych i wywoływać jak zwykłe funkcje,
- wykorzystuje się je w miejscach, gdzie wymagany jest obiekt funkcyjny, ale pisanie w tym celu klasycznej funkcji (z pomocą słowa kluczowego ‘def’) byłoby nieopłacalne.

Przykład 2:

```
# funkcja lambda jednoparametrowa:
>>>
>>> kwadrat = lambda x:x*x
>>> a=kwadrat(13)
>>> a
169
>>>
```

Wyrażenia Lambda

Przykład 2 (cd.):

```
>>> import datetime

# funkcja bezparametrowa:

>>> aktualna_godzina = lambda: datetime.datetime.now().hour
>>> aktualna_godzina
<function <lambda> at 0x00000000029AC400>    #tak nie można, to jest funkcja
>>> aktualna_godzina()
16
>>> print(aktualna_godzina())
16

#funkcja dwuparametrowa

>>> suma=lambda a,b: a+b
>>> suma(2,3)
5
>>>
```

Funkcje wyższego rzędu

Funkcje wyższego rzędu:

- w Pythonie wszystko jest obiektem – funkcje również; funkcje mogą być przekazywane jako parametry, mogą być zwracane jak wartość,
- funkcje wyższego rzędu to funkcje, które przyjmują i/lub zwracają obiekty w postaci funkcji.

Przykład 3:

```
#funkcja zwracająca inną funkcję:
 >>> def gen_inc(n):
...       def fun(x):
...           return n+x
...       return fun
...
>>> inc5=gen_inc(5)
>>> inc5(10)
15
>>> print(inc5(10))
15
>>>
```

Funkcje wyższego rzędu

Przykład 3:

```
#definicja funkcji 'map' pobierającej parametry: funkcja i lista; zwracająca listę:

>>> def map(fun, list):
...     return [fun(item) for item in list]
...
>>> def add100(x):
...     return x+100
...
>>> lista_0_9=list(range(10))
>>> lista_0_9
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Funkcje wyższego rzędu, cd.

Przykład 3, cd:

```
#przykład  wywołania funcji 'map':

>>> map(add100, lista_0_9)
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]
>>>

#przykład  wywołania funkcji 'map' – zamiast funkcji - bezpośrednia definicja
wyrażenia lambda, zamiast listy – funkcja generująca zakres 'range()':

>>> map(lambda x: x+100, range(10))
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]
>>>
```

Generatory

Generatory:

- dają możliwość generowania sekwencji obiektów, po których w wygodny sposób można iterować, **bez konieczności przechowywania ich w pamięci** (tak się dzieje, jeśli korzystamy z listy),
- charakteryzują się one tzw. „leniwą ewaluacją” — generują i wydają ('yield') kolejne elementy sekwencji jedynie w momencie, gdy są one potrzebne **(oszczędność pamięci)**,
- obiekty takie mogą być wkomponowane w pętle, dla których 'dają' kolejne elementy sekwencji,
- kiedy zaczyna się iteracja po elementach z wykorzystaniem np. „for”, za każdym razem uruchamiany jest generator, który generuje kolejny element sekwencji (wyrażenie 'yield'); generator zapamiętuje jaki element ostatnio 'wydał' , 'wie' jak utworzyć kolejny element i 'wie' jak zasygnalizować koniec sekwencji (brak kolejnych elementów),
- należy rozróżnić istnienie funkcji generatora (obiekt klasy 'function') od generatora (obiekt klasy 'generator'), choć mianem generatora określa się potocznie obydwie klasy obiektów,
- funkcje generatora 'tylko' zwracają obiekty klasy 'generator', natomiast np. w pętli są wykorzystywane bezpośrednio obiekty klasy 'generator'.

Generatory

Przykład 4: Funkcja generatora

```
# definicja funkcji generatora – zwraca kolejno liczby o ‘n’ do 1:

>>> def gen_fun(n):
...     while n:
...         print('zwracam %d z generatora.'  %n)
...         yield n
...         n-=1
...
```

Generatory

Przykład 4 (cd.):

```
# wykorzystanie funkcji generatora jako źródła sekwencji do pętli:
>>>import time
>>>for x in generator(10):
...     time.sleep(0.5)
...     print('Wypisuję %d w pętli.' %x)
...     time.sleep(0.5)
...
zwracam 10 z generatora.
Wypisuję 10 w pętli.
zwracam 9 z generatora.
Wypisuję 9 w pętli.
…
zwracam 3 z generatora.
Wypisuję 3 w pętli.
zwracam 2 z generatora.
Wypisuję 2 w pętli.
zwracam 1 z generatora.
Wypisuję 1 w pętli.
```

Generatory

Przykład 5:

```
#bezpośrednia definicja generatora (tzw. wyrażenie generatorowe):
>>>
>>> generator=((x,2*x) for x in range(10))
>>> for a,b in generator:
...         print(a,b)
...
0 0
1 2
2 4
3 6
4 8
5 10
6 12
7 14
8 16
9 18
>>>
```

Generatory

Przykład 5 (cd.):

```
#powyższy wynik można osiągnąć z wykorzystaniem listy:

>>> list=[(x,2*x) for x in range(10)]
>>> for a,b in list:
...     print(a,b)
...
0 0
1 2
2 4
3 6
4 8
5 10
6 12
7 14
8 16
9 18
>>>
```

Funkcja generatora a generator

Przykład 6:

```
#definicja funkcji generatora:
>>> def gen(n):
...     while n:
...         yield n
...         n-=1
...
>>> type(gen)
<class 'function'>
#teraz funkcja generatora zwraca obiekt generatora:
>>> gen_obj=gen(5)
>>> type(gen_obj)
<class 'generator'>
>>>
#bezpośrednie utworzenie obiektu generatora  - wyrażenie generatorowe:
>>> gen_obj2=((x,2*x) for x in range(10))
>>> type(gen_obj2)
<class 'generator'>
>>>
```

Argumenty funkcji

Argumenty funkcji:

- pozycyjne i nazwane,
- można przekazywać argumenty z domyślnymi wartościami (nazwane),
- ważna jest kolejność – argumenty nazwane muszą znajdować się za argumentami pozycyjnymi,
- do funkcji można przekazywać także zmienną liczbę argumentów z wykorzystaniem składni: *args, **kwargs (tu także ważna jest kolejność).

Przykład 7 – argumenty funkcji (nazwane):

```
def fun(x,y):
    return x+y
print(fun(1,2))

def fun(x,y,z=5):
    return x+y+z
print(fun(1,2,4))
```

```
#output:
3
7
```

Zmienna liczba argumentów funkcji

Specjalne argumenty funkcji *args, **kwargs:

- pozwalają na przekazywanie zmiennej liczby argumentów z wykorzystaniem mechanizmów 'pakowania' i 'odpakowywania',
- realizuje się to poprzez dodanie * lub ** do nazwy zmiennej parametru funkcji, np. *args umożliwia przekazywanie zmiennej liczby argumentów pozycyjnych (pakowanie do krotki), **kwargs umożliwia przekazywanie zmiennej liczby argumentów nazwanych (pakowanie do słownika).
- nazwy args i kwargs nie są obowiązkowe.
- ważna jest poprawna kolejność argumentów, np.: fun(a,b,*args,c=5,**kwargs)

Przykład 8 - zmienna liczba argumentów pozycyjnych funkcji:

```
def test_args(*args):    #'args' jest traktowana w funkcji jak krotka
   for x in args:
      print(x)

test_args('a1', 'a2')
test_args()
test_args('b1', 'b2','b3')
```

```
#output:
a1
a2
b1
b2
b3
```

Zmienna liczba argumentów funkcji

Przykład 9 - zmienna liczba argumentów nazwanych funkcji:

```
def test_kwargs(**kwargs):      #kwargs to po prostu słownik
    for key,value in kwargs.items():
        print("%s = %s" %(key, value))

test_kwargs(k1='a1', k2='a2')
test_kwargs()
test_kwargs(k1='b1', k2='b2',k3='b3')
```

```
#output:
k1 = a1
k2 = a2
k1 = b1
k2 = b2
k3 = b3
```

Zadanie 1

Napisz funkcję, która przyjmuje napis — tekst w języku naturalnym. Funkcja ma zwrócić listę krotek: słowo i jego długość dla wszystkich słów znajdujących się w napisie. Użyj list składanych.

Zadanie 2

Korzystając z list składanych wygeneruj listę ‚n’ pierwszych elementów ciągu Fibonacciego do listy. Liczba 'n' podawana jest w konsoli (funkcja input()). Wykorzystaj funkcję, która generuje pojedynczy element ciągu Fibonacciego oraz funkcję generatora range().

Zadanie 3

Zaimplementuj samodzielnie funkcję *‘filter’*, która przyjmuje parametry: funkcję logiczną (np. funkcję, która zwraca prawdę, jeśli liczba jest podzielna przez 3 i 7) oraz listę (np. listę liczb). Funkcja ma zwrócić listę elementów podanej listy, które spełniają warunek określony za pomocą przekazanej, jako pierwszy parametr, funkcji.

Zadanie 4

Napisz funkcję, która przyjmuje listę punktów na płaszczyźnie (w postaci dwuelementowych krotek i z wartościami typu float) i zwraca listę tych samych punktów posortowaną według odległości od początku układu współrzędnych. Użyj metody ‚sort' obiektu ‚list' (lub wbudowanej funkcji ‚sorted' — czym one się różnią?) oraz wyrażenia lambda jako jej argumentu 'key'.

Zadanie 5

Napisz generator, który będzie zwracał nazwy kolejnych plików z bieżącego katalogu, których rozszerzenie to ‘.py’. Skorzystaj z funkcji ‘os.listdir’ oraz metody ‘endswith’ klasy ‘str’.

Zadanie 6

Napisz funkcję, która przyjmuje argument - data w postaci krotki, np.: (rok, miesiąc, dzień) i zwraca informację o dniu tygodnia dla tej daty (dla lat 1-9999 ne).

Zadanie 7

Zmodyfikuj zadanie 6 tak, aby funkcja akceptowała różną liczbę argumentów pozycyjnych i nazwanych (w nawiasach krotki), np.:

dni_tygodnia((2020,3,10),(2021,3,11),data1=(2023,3,10),data2=(2024,3,11))

Funkcja powinna zwracać słownik zawierający pary {(data) : 'dzień tygodnia'}.

Sugestia – zdefiniuj funkcje wewnętrzną, która znajduje dzień tygodnia dla jednej daty.