

Python Comprehension

Jest to mechanizm służący do generowania kolekcji (lista, słownik, zbiór) na podstawie jednowierszowej definicji. Równoważne definicje zawsze można podać za pomocą pętli. Czasami zaś wystarczy przepisać na język Python definicję matematyczną zbioru.

Możliwa składnia

#Zamiast pisać w pętli

lista = []

for element in zakres:

if pewien_warunek_na(element):

lista.append(„Cos sie dzieje z:” + element)

#możemy zapisać w jednej linijce

lista = [„Cos sie dzieje z:” + element for element in zakres if pewien_warunek_na(element)]

Przykład pierwszy

$A = \{x^2: x \in \langle 0, 9 \rangle\}$

$B = \{1, 3, 9, 27, \dots, 3^5\}$

$C = \{x: x \in A \text{ i } x \text{ jest liczbą nieparzystą}\}$

W pythonie zapiszemy to:

#wersja z pętlą

a = []

for x *in* range(10):

a.append(x**2)

print(a)

b = []

for y *in* range(6):

b.append(3**y)

print(b)

c = []

for z *in* a:

if z % 2 == 1:

c.append(z)

```
print(c)

#wersja z python comprehension
a = [x**2 for x in range(10)]
b = [3**i for i in range(6)]
c = [x for x in a if x % 2 == 1]

print(a)
print(b)
print(c)
```

Przykład drugi

Chcemy uzyskać liczby parzyste z podanego zakresu

```
#wersja z pętlą
liczby = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
lista = []

for i in liczby:
    if i % 2 == 0:
        lista.append(i)

print("Liczby parzyste uzyskane z wykorzystaniem pętli")
print(lista)

print()
```

```
#wersja z python comprehension
lista2 = [i for i in liczby if i % 2 == 0]

print(lista2)
```

Przykład trzeci zagnieżdżenia

```
#wersja z zagnieżdżonymi pętlami
lista = []

for i in [1, 2, 3]:
    for j in [4, 5, 6]:
        lista.append((i,j))
```

```
print(lista)
```

#wersja z python comprehension

```
lista2 = [(i,j) for i in [1, 2, 3] for j in [4, 5, 6]]
```

```
print(lista2)
```

Przykład czwarty związany ze zamianą klucza z wartością w słowniku

#wersja z pętlą

```
skroty = {"PZU": "Państwowy zakład ubezpieczeń",  
          "ZUS": "Zakład ubezpieczeń społecznych",  
          "PKO": "Państwowa kasa oszczędności"}
```

```
odwrocone = {}
```

```
for key,value in skroty.items():
```

```
    odwrocone[value] = key
```

```
print(odwrocone)
```

#wersja z python comprehension

```
odwrocone2 = {value: key for key, value in skroty.items()}
```

```
print(odwrocone2)
```

Funkcje

W pythonie możemy definiować własne funkcje, które będziemy traktować jak podprogramy lub jak funkcje w matlabie.

Składnia

```
def nazwa_funkcji(arg_pozycyjny, arg_domyslny=wartosc, *arg_4, **arg_5):  
    instrukcje  
    return wartość
```

Definicja instrukcji to instrukcja która tworzy obiekt. Funkcje możemy wywoływać z argumentami lub bez ale zawsze musimy używać nawiasów (nawet jak nie ma argumentów). Funkcja może zwracać jedną lub wiele wartości, które będą zwrócone jako krotka

Przykład pierwszy

Chcemy zdefiniować funkcję, która będzie obliczać pierwiastki równania kwadratowego

```
def row_kwadratowe(a,b,c):  
    delta = b**2 - 4 * a * c  
    if delta < 0:  
        print("brak pierwiastków")  
        return -1  
    elif delta == 0:  
        print("jedne pierwiastek")  
        x = (-b) / (2 * a)  
        return x  
    else:  
        print("dwa pierwiastki")  
        x1 = (-b - math.sqrt(delta)) / (2 * a)  
        x2 = (-b + math.sqrt(delta)) / (2 * a)  
        return x1,x2  
print(row_kwadratowe(6,1,3))  
print(row_kwadratowe(1,2,1))  
print(row_kwadratowe(1,4,1))
```

Przykład drugi

Definiujemy funkcję z wartościami domyślnymi

```
import math  
  
def dlugosc_odcinka(x1 = 0, y1 = 0, x2 = 0, y2 = 0):  
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)  
  
#wywołujemy funkcje dla wartości domyślnych  
print(dlugosc_odcinka())
```

#wywołujemy funkcje dla własnych podanych wartości

#są to argumenty pozycyjne czyli ważna jest kolejność podania wartości

```
print(dlugosc_odcinka(1,2,3,4))
```

#wywołujemy funkcje podając mieszane wartości

#dwie pierwsze interpretowane są jako x1 i y1 jak podano w definicji funkcji

```
print(dlugosc_odcinka(2, 2, y2=2, x2=1))
```

#wywołujemy funkcje podając wartości nie w kolejności

```
print(dlugosc_odcinka(y2=5, x1=2, y1=2, x2=6))
```

#wywołujemy funkcje podając tylko dwa argumenty a reszta domyślne

```
print(dlugosc_odcinka(x2=5, y2=5))
```

Przykład trzeci

Symbol * oznacza dowolną ilość argumentów przechowywanych w krotce

```
def ciag(* liczby):
```

```
    # jeżeli nie ma argumentów to
```

```
    if len(liczby) == 0:
```

```
        return 0
```

```
    else:
```

```
        suma = 0
```

```
        #sumujemy elementy ciągu
```

```
        for i in liczby:
```

```
            suma += i
```

```
        #zwracamy wartość sumy
```

```
        return suma
```

#wywołanie gdy nie ma argumentów

print(ciaag())

#podajemy argumenty

print(ciaag(1, 2, 3.5, 4, 5, 6, 7, 8))

Przykład czwarty

****** dwie gwiazdki oznaczają że możemy użyć dowolną ilość argumentów z kluczem

*def to_lubie(** rzeczy):*

for cos in rzeczy:

print("To jest ")

print(cos)

print(" co lubie ")

print(rzeczy[cos])

to_lubie(slodycze="czekolada", rozrywka=['gry', 'filmy'])

Moduły i pakiety

Żeby użyć funkcji matematycznych potrzebowaliśmy zaimportować plik math.

Taki plik nazywa się modułem i są tam zapisane po prostu kody w języku Python. Jeśli takich plików będziemy mieć kilka to możemy utworzyć z nich pakiet.

Import modułów systemowych

Jeden import modułu powinien być w jednej linii np.

Import sys

Można również zapisać import modułu w postaci:

*from math import **

Import modułu zamieszczamy na początku pliku. Ewentualnie za komentarzami. Zaleca się następującą kolejność importów:

- Biblioteki standardowe
- Powiązane biblioteki zewnętrzne
- Lokalne aplikacje/biblioteki

Tworzenie swojego modułu

- Tworząc swój moduł piszemy funkcje i zapisujemy ją do pliku z rozszerzeniem .py
- Następnie dołączamy do nowego skryptu swój moduł używając instrukcji

Przykład

Zawartość pliku *litery*, który będzie naszym modułem

Plik *litery*

```
def wyswietl(a):
```

```
    print(a)
```

```
def dlugosc(a):
```

```
    return len(a)
```

Teraz możemy już wykorzystać funkcje z modułu *litery* (to będzie nowy skrypt)

```
import litery
```

```
a = "Ala ma kota"
```

```
litery.wyswietl(a)
```

```
print(litery.dlugosc(a))
```

```
#wyswietla wszystkie zmienne oraz nazwy modułów, które się w nim znajdują
```

```
print(dir(litery))
```

Tworzenie swojego pakietu

Pakiet składa się z kilku modułów i najczęściej zapisywany w określonym folderze, gdzie nazwa folderu oznacza nazwę pakietu. Jeżeli chcemy stworzyć pakiet musimy utworzyć katalog dodać tam moduły a następnie dorzucić pliku o nazwie `__init__.py`, w którym powinien się znaleźć sposób importu plików. Dla stylu **import pakiet.moduł** plik zostaje pusty dla stylu **from pakiet import *** w pliku zapisujemy zmienną `__all__` która zawiera wszystkie moduły, które mogą być zaimportowane.

Przykład

Tworzymy jeszcze jeden moduł

#piosenka.py

```
def spiew():
```

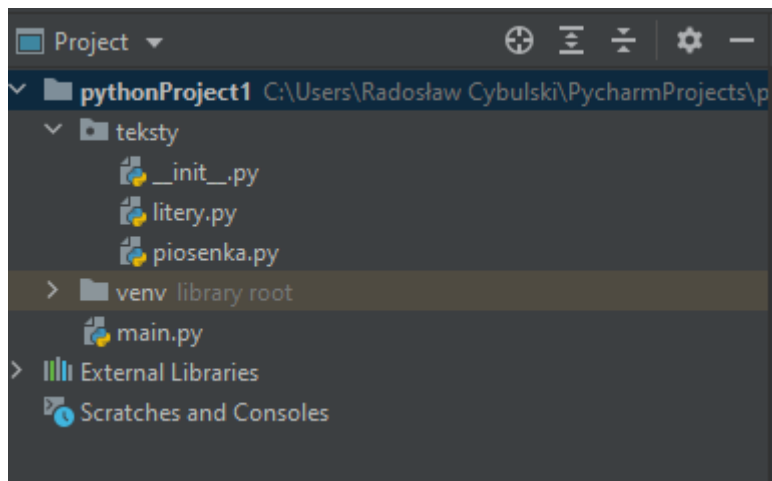
```
    print("La la la la la")
```

```
def zespol():
```

```
    print("Boysband")
```

```
    print("Girl'n'dance")
```

Tworzymy teraz katalog teksty i wrzucamy tam nasze moduły oraz edytujemy plik `__init__.py`



Zawartość pliku `__init__.py`

```
__all__ = ["literary", "piosenka"]
```

Operacje na plikach

Operacje na plikach można podzielić na trzy etapy:

1. Otwarcie pliku
2. Działanie na pliku (odczyt lub zapis)
3. Zamknięcie pliku

```
plik = open(nazwa, tryb, bufor)
```

gdzie:

plik – nazwa obiektu, którą sami nadajemy

nazwa – nazwa pliku na dysku, jaka jest

tryb – tryb otwarcia pliku (np. do odczytu, do zapisu itd.)

bufor – obszar pamięci przechowujący dane w oczekiwaniu na zapis i odczyt

Wybrane tryby otwarcia plików:

r – tylko do odczytu. Plik musi istnieć:

w – tylko do zapisu. Jeżeli pliku nie ma to zostanie utworzony a jeżeli jest to jego zawartość zostanie zastąpiona nową.

a – do dopisywania. Dane dopisują się na końcu pliku. Jeżeli plik nie istnieje to zostanie utworzony

r+ - do odczytu i zapisu. Plik musi istnieć.

w+ - do odczytu i zapisu. Jeżeli plik nie istnieje zostanie utworzony.

a+ - do odczytu i zapisu. Jeżeli plik nie istnieje zostanie utworzony.

Do odczytania danych z pliku można użyć komend:

- read(rozmiar) – odczytuje dane o rozmiarze, jeśli podany
- readline(rozmiar) – odczytuje wiersze lub ilości znaków jeśli podano rozmiar
- readlines() – odczytuje wiersze z pliku

Przykład pierwszy

```
plik = open("tekst.txt", "r")
```

```
#odczyt 10 znaków
```

```
znaki = plik.read(10)
```

```
#odczyt jednej lini z pliku
```

```
linia = plik.readline()
```

```
#odczyt wierszy z pliku
```

```
wiersze = plik.readlines()
```

```
#zamknięcie pliku
```

```
plik.close()
```

```
#drukujemy 10 znaków
```

```
print(znaki)
```

```
print("\n")
```

```
#drukujemy linie
```

```
print(linia)
```

```
print("\n")
```

```
#drukujemy wiersze
```

```
print(wiersze)
```

Uwaga 1

Jeśli otwieramy plik i odczytujemy z niego dane jak wyżej to wskaźnik aktualnej pozycji w pliku się przemieszcza. Dlatego w wyniku najpierw otrzymamy pierwsze 10 znaków, potem następne znaki z pozostałej linijki a na koniec resztę linijek tekstu z pliku.

Uwaga 2

Po zakończeniu działania skryptu wszystkie otwarte pliki zamykane są automatycznie.

Do zapisywania danych do pliku możemy użyć:

- `write(łańcuch)` – zapisuje dane ze zmiennej łańcuch
- `writelines(lista)` – zapisuje dane z listy

Przykład drugi

```
import sys

print("Podaj kierunek studiów, rok i specjalność")

#odczyt danych ze standardowego wyjścia
dane = sys.stdin.readline()

#otwarcie pliku
plik = open("dane.txt", "w+")

#zapisanie do pliku
plik.write(dane)

#zamykamy plik
plik.close()

#tworzymy liste
lista = []

for x in range(6):
    lista += [x]

#otwarcie pliku do dopisania
plik = open("dane.txt", "a+")

#zapisujemy
plik.writelines(str(lista))

#zamknięcie pliku
plik.close()
```

Przykład trzeci

Plik możemy otwierać do zapisu i odczytu za pomocą komendy `with`, wówczas nie musimy martwić się o zamknięcie pliku. Pętla `for` pozwala na wyświetlenie pliku linijka po linijce `with open("tekst.txt", "r") as plik:`

```
for linia in plik:
```

```
    print(linia,end="")
```