KURS PROGRAMOWANIA W JĘZYKU PYTHON

TYDZIEŃ 2 – STRUKTURY DANYCH

1. Definiowanie zmiennych, typy zmiennych, zmienne globalne

Definiowanie zmiennych

W Pythonie zmienna jest nazwanym miejscem w pamięci komputera, które przechowuje dane. Definiujemy zmienne poprzez przypisanie wartości do nazwy zmiennej, na przykład: x = 5 przypisuje wartość 5 zmiennej x. Wartości mogą być liczbami, łańcuchami znaków, listami, krotkami, słownikami itp.

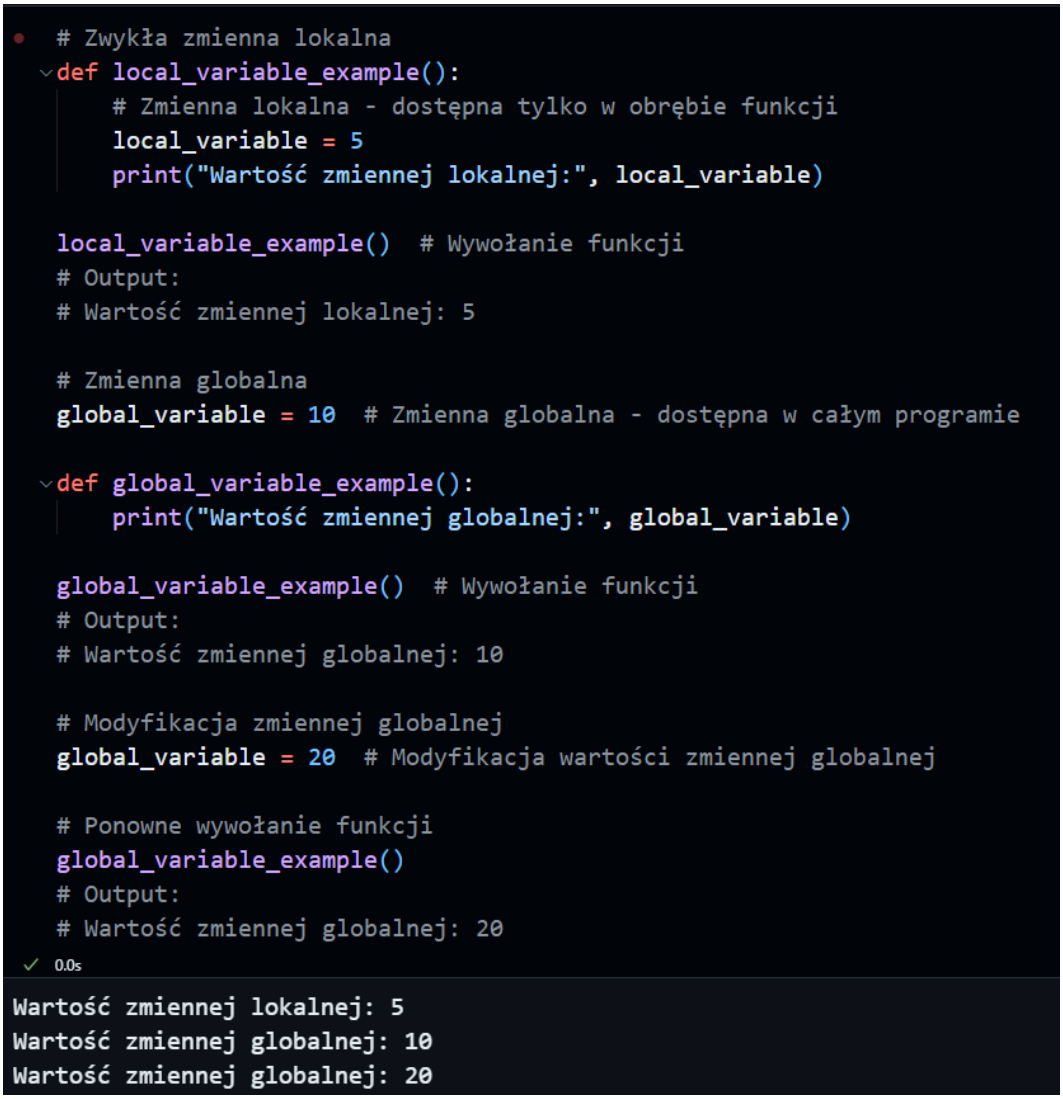
Typy danych

Typ danych określa rodzaj informacji przechowywanej w zmiennej. W Pythonie mamy wiele wbudowanych typów danych, np.:

- **int** - liczba całkowita (np. x = 5)
- **float** - liczba zmiennoprzecinkowa (np. y = 3.14)
- **str** - łańcuch znaków (np. name = "John")
- **list** - lista (np. numbers = [1, 2, 3, 4, 5])
- **tuple** - krotka (np. point = (10, 20))
- **dict** - słownik (np. person = {'name': 'John', 'age': 30})

Zmienne globalne

Zmienne globalne są zmiennymi, które są dostępne w całym programie. Mogą być one zdefiniowane poza jakąkolwiek funkcją i są dostępne dla wszystkich funkcji w programie. Zmienne globalne mogą być zmieniane i odczytywane z różnych części kodu, co może być zarówno zaletą, jak i wadą. Ich zmiana może prowadzić do niezamierzonych efektów ubocznych, dlatego należy ostrożnie nimi zarządzać.



```python
# Zwykła zmienna lokalna
def local_variable_example():
    # Zmienna lokalna - dostępna tylko w obrębie funkcji
    local_variable = 5
    print("Wartość zmiennej lokalnej:", local_variable)

local_variable_example()  # Wywołanie funkcji
# Output:
# Wartość zmiennej lokalnej: 5

# Zmienna globalna
global_variable = 10  # Zmienna globalna - dostępna w całym programie

def global_variable_example():
    print("Wartość zmiennej globalnej:", global_variable)

global_variable_example()  # Wywołanie funkcji
# Output:
# Wartość zmiennej globalnej: 10

# Modyfikacja zmiennej globalnej
global_variable = 20  # Modyfikacja wartości zmiennej globalnej

# Ponowne wywołanie funkcji
global_variable_example()
# Output:
# Wartość zmiennej globalnej: 20
```

0.0s

```
Wartość zmiennej lokalnej: 5
Wartość zmiennej globalnej: 10
Wartość zmiennej globalnej: 20
```

Wiem, że nie mieliście funkcji, ale to tylko takie krótkie zobrazowanie, przyda wam się w dalszej części kursu.

2. Listy, krotki, słowniki, tuple, macierze, generatory, iteratory

Listy (lists)

Dynamiczne struktury danych w Pythonie, które mogą przechowywać wiele elementów różnych typów danych. Elementy w liście są uporządkowane i indeksowane od zera. Listy są mutowalne, co oznacza, że możemy je zmieniać po ich utworzeniu.

```python
# Definicja listy
numbers = [1, 2, 3, 4, 5]
words = ['apple', 'banana', 'orange']
mixed = [1, 'apple', True]

# Dodawanie elementu do listy
numbers.append(6)
print(numbers)  # Output: [1, 2, 3, 4, 5, 6]

# Dostęp do elementów listy
print(words[0])  # Output: 'apple'
```

✓ 0.0s

```
[1, 2, 3, 4, 5, 6]
apple
```

Krotki (tuples)

Krotki są podobne do list, ale są niezmienne (immutable). Oznacza to, że po utworzeniu krotki nie można zmieniać jej zawartości. Krotki są często używane do przechowywania zestawów danych, które nie powinny być modyfikowane.

```python
# Definicja krotki
point = (10, 20)
dimensions = (100, 200, 300)

# Dostęp do elementów krotki
print(point[0])  # Output: 10

# Krotki są niezmienne
# point[0] = 20  # To spowoduje błąd
```

✓ 0.0s

```
10
```

Słowniki (dictionaries)

kolekcje par klucz-wartość, gdzie każdy klucz jest unikalny i mapuje się na wartość. Słowniki są mutowalne i pozwalają na szybkie wyszukiwanie wartości za pomocą klucza.

```
# Definicja słownika
person = {'name': 'John', 'age': 30, 'city': 'New York'}

# Dostęp do wartości za pomocą klucza
print(person['name'])  # Output: 'John'

# Dodawanie nowej pary klucz-wartość
person['occupation'] = 'Engineer'
print(person)  # Output: {'name': 'John', 'age': 30, 'city': 'New York', 'occupation': 'Engineer'}
```

✓ 0.0s

```
John
{'name': 'John', 'age': 30, 'city': 'New York', 'occupation': 'Engineer'}
```

Zbiory (sets)

przechowują unikalne elementy w dowolnym porządku. Elementy w zbiorze nie są indeksowane, co oznacza, że nie można odwoływać się do nich za pomocą indeksów jak w przypadku list czy krotek. Główną cechą zbioru jest to, że każdy element występuje w nim tylko raz, nawet jeśli próbujemy dodać go wielokrotnie. Zbiory w Pythonie są przydatne w sytuacjach, gdy chcemy przechować jedynie unikalne wartości oraz wykonywać operacje zbiorowe, takie jak przecięcie, różnica czy suma.

```
my_set = {1, 2, 3, 4, 5}
print("Zbiór:", my_set)
```

✓ 0.0s

```
Zbiór: {1, 2, 3, 4, 5}
```

Macierze (arrays)

W kontekście Pythona, macierze mogą być reprezentowane za pomocą list lub pakietu numpy, który dostarcza wydajne metody operacji na macierzach numerycznych.

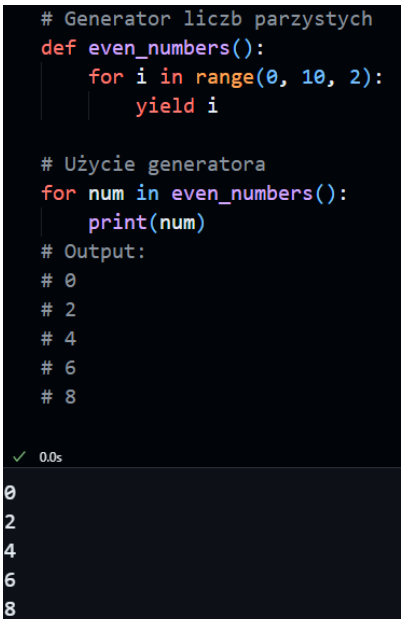
```
import numpy as np

# Definicja macierzy
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Operacje na macierzach
print(matrix)
# Output:
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]
```

✓ 0.0s

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

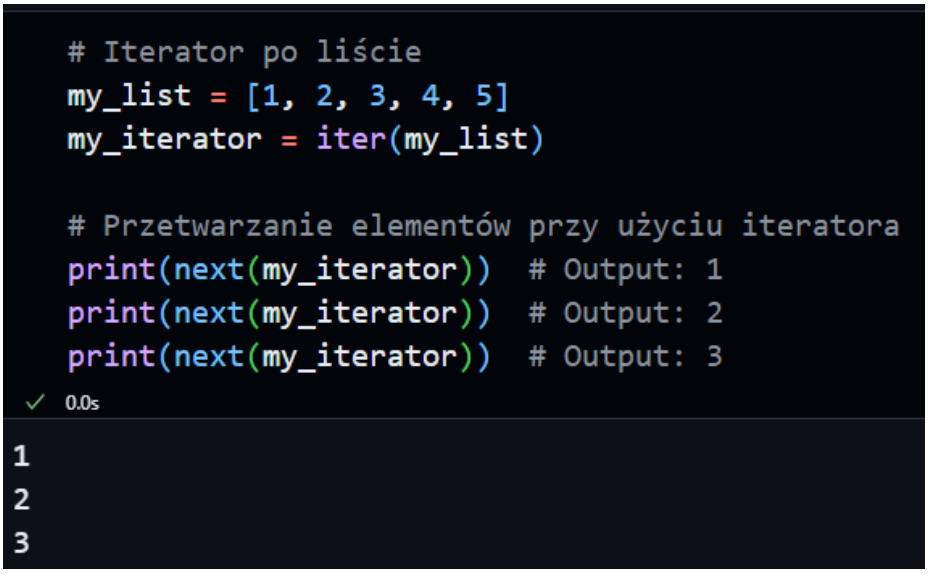
Generatory (generators)

Funkcje, które zwracają sekwencje wyników w trakcie ich wykonywania, zamiast oczekiwać na zakończenie całego obliczenia przed zwróceniem wyniku. Generatory są przydatne, gdy potrzebujemy wygenerować dużą ilość danych, ale nie chcemy ich przechowywać w pamięci.



Iteratory (iterators)

Obiekty, które umożliwiają iterację po sekwencji danych, takiej jak lista lub krotka, aż do momentu wyczerpania wszystkich elementów. Iteratory są używane w pętlach for do przetwarzania elementów sekwencji jeden po drugim, bez potrzeby przechowywania wszystkich danych w pamięci.



3. Pakiet collections

To moduł wbudowany w język Python, który zawiera specjalne typy danych i struktury, które rozszerzają możliwości podstawowych typów danych dostępnych w Pythonie.

Oto kilka najważniejszych typów danych dostępnych w pakiecie collections:

- **Counter**: jest obiektem, który służy do zliczania elementów w kolekcji. Może być używany do zliczania ilości wystąpień poszczególnych elementów w liście, krotce, łańcuchu znaków lub innym iterowalnym obiekcie.

```
from collections import Counter

# Definicja listy z powtarzającymi się elementami
my_list = ['a', 'b', 'c', 'a', 'b', 'a']

# Utworzenie obiektu Counter
counter = Counter(my_list)

# Wyświetlenie liczby wystąpień poszczególnych elementów
print(counter)
# Output: Counter({'a': 3, 'b': 2, 'c': 1})
```

✓ 0.0s

```
Counter({'a': 3, 'b': 2, 'c': 1})
```

- **defaultdict**: to specjalna odmiana słownika, która automatycznie tworzy domyślne wartości dla nowych kluczy. Może być przydatny, gdy próbujemy dodać wartość do klucza, który jeszcze nie istnieje w słowniku.

```
from collections import defaultdict

# Utworzenie defaultdict z domyślną wartością 0
default_dict = defaultdict(int)

# Dodanie kluczy i wartości do defaultdict
default_dict['a'] = 1
default_dict['b'] = 2

# Dostęp do nieistniejącego klucza
print(default_dict['c'])  # Output: 0
```

✓ 0.0s

```
0
```

- **namedtuple**: jest fabryką, która tworzy podklasy tuple z nazwanymi polami. Jest to przydatne w sytuacjach, gdy chcemy używać tuple, ale jednocześnie zachować czytelność poprzez dostęp do ich pól za pomocą nazw.

```
from collections import namedtuple

# Definicja namedtuple
Person = namedtuple('Person', ['name', 'age', 'city'])

# Utworzenie instancji namedtuple
person = Person(name='John', age=30, city='New York')

# Dostęp do pól przez nazwę
print(person.name)  # Output: John
```

✓ 0.0s

```
John
```

- **OrderedDict**: to słownik, który zachowuje kolejność wstawiania elementów. W przeciwieństwie do zwykłego słownika, OrderedDict pamięta kolejność dodawania elementów i zwraca je w tej samej kolejności podczas iteracji.

```
from collections import OrderedDict

# Utworzenie OrderedDict
ordered_dict = OrderedDict()

# Dodanie elementów z zachowaniem kolejności
ordered_dict['a'] = 1
ordered_dict['b'] = 2
ordered_dict['c'] = 3

# Wyświetlenie elementów z zachowaniem kolejności
print(ordered_dict)
# Output: OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

✓ 0.0s

```
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

- **deque**: to podwójnie zakończona kolejka, która pozwala na szybkie dodawanie i usuwanie elementów zarówno z przodu, jak i z tyłu. Jest to bardziej wydajna struktura danych niż standardowa lista w przypadku częstych operacji dodawania i usuwania elementów z początku lub końca kolejki.

```python
from collections import deque
my_deque = deque([1, 2, 3])

# Wyświetlenie początkowej zawartości kolejki
print("Początkowa kolejka:", my_deque)
# Output: Początkowa kolejka: deque([1, 2, 3])

# Dodanie elementu na początek kolejki
my_deque.appendleft(0)
print("Po dodaniu 0 na początek kolejki:", my_deque)
# Output: Po dodaniu 0 na początek kolejki: deque([0, 1, 2, 3])

# Dodanie elementu na koniec kolejki
my_deque.append(4)
print("Po dodaniu 4 na koniec kolejki:", my_deque)
# Output: Po dodaniu 4 na koniec kolejki: deque([0, 1, 2, 3, 4])

# Usunięcie elementu z początku kolejki
removed_left = my_deque.popleft()
print("Po usunięciu elementu z początku kolejki:", my_deque)
# Output: Po usunięciu elementu z początku kolejki: deque([1, 2, 3, 4])

# Usunięcie elementu z końca kolejki
removed_right = my_deque.pop()
print("Po usunięciu elementu z końca kolejki:", my_deque)
# Output: Po usunięciu elementu z końca kolejki: deque([1, 2, 3])
```

✓ 0.0s

```
Początkowa kolejka: deque([1, 2, 3])
Po dodaniu 0 na początek kolejki: deque([0, 1, 2, 3])
Po dodaniu 4 na koniec kolejki: deque([0, 1, 2, 3, 4])
Po usunięciu elementu z początku kolejki: deque([1, 2, 3, 4])
Po usunięciu elementu z końca kolejki: deque([1, 2, 3])
```