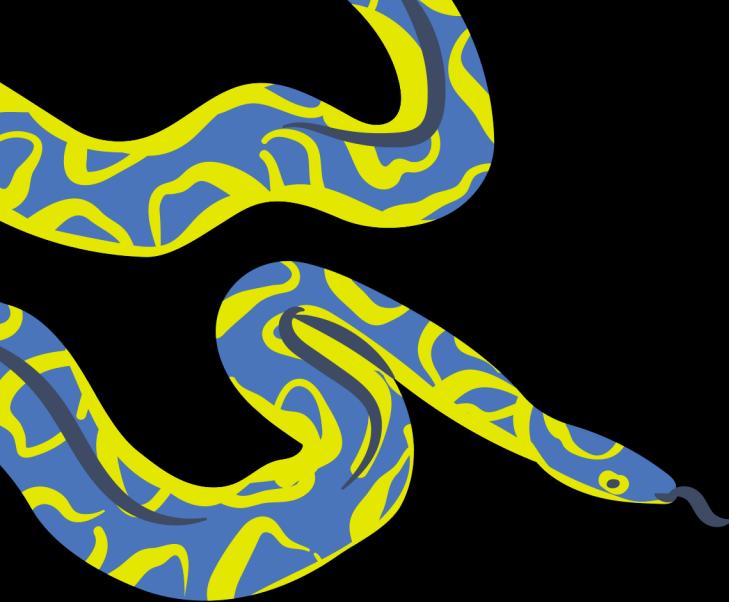




SciByteHub



Python

07.05



RECAP - import

Importowanie bibliotek do kodu:

`import biblioteka`

Importowanie biblioteki jako alias:

`import biblioteka as alias`



RECAP - Pętla *for*

Pętla *for* (dla każdego) używana jest do wymieniania elementów z list, krotek, słowników, generatorów, itp.



RECAP - Pętla for

```
lista = ["element1", "element2", "element3"]
for element in lista:
    print(element)
```

Pętla ta wybierze kolejne elementy z listy i wyświetli je jako output.



RECAP - Pętla while

Pętle **while** (dopóki) wykonują kod pod nimi dopóki zadany im warunek jest spełniony i zwraca *True*. Ich syntax wygląda następująco:

```
i = 0
```

```
x = 0
```

```
while i <= 20:
```

```
    x = x + i
```

```
    i = i + 1
```



RECAP - Warunki

Mamy 3 główne słowa-klucze: *if* (jeżeli), *elif* (w przeciwnym wypadku jeżeli) oraz *else* (w przeciwnym wypadku). Twierdzenia te podążają podobnym tokiem logicznym jak język mówiony czy pisany.



RECAP - Warunki

Przykład - znajdywanie konkretnej litery w słowie:

slowo = 'banan'

if 'c' in slowo:

print(f"Litera C jest w słowie {slowo}")

elif 'n' in slowo:

print(f"Litery C nie było, natomiast była litera N w słowie {slowo}")

else:

print(f"Ani N ani C nie występują w słowie {slowo}")



Macierze i ich mnożenie

$$3 \cdot 0 + 1 \cdot 2 + 0 \cdot 0 = 2$$

$$\begin{bmatrix} 3 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 4 \\ 2 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & & \\ & & \end{bmatrix}$$

2 × 3

3 × 2

2 × 2



Macierze w Pythonie

```
import numpy as np  
  
A = np.array([[1, 2, 3], [4, 5, 6]])  
print(A)  
✓ 0.0s
```

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

```
import numpy as np  
  
zerowa_macierz = np.zeros((2,3))  
print(zerowa_macierz)  
✓ 0.0s
```

```
[[0. 0. 0.]  
 [0. 0. 0.]]
```

```
import numpy as np
```

```
print('B: ')  
B = np.arange(4)  
print(B)
```

```
print("C: ")  
C = np.arange(6).reshape(2, 3)  
print(C)
```

```
✓ 0.0s
```

```
B:  
[0 1 2 3]  
C:  
[[0 1 2]  
 [3 4 5]]
```

Mnożenie macierzy

```
import numpy as np

A = np.array([[3,1,0],[0,2,1]])
B = np.array([[0,4],[2,2],[0,1]])

np.dot(A,B)
```

✓ 0.0s

```
array([[ 2, 14],
       [ 4,  5]])
```

Transponowanie macierzy

$$A = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}_{2 \times 3}$$

$$A^T = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix}_{3 \times 2}$$

Transponowanie macierzy w Pythonie

```
import numpy as np

print("Macierz A:")
A = np.array([[1, 2, 3], [4, 5, 6]])
print(A)

print("Transponowana macierz A")
A_transponowane = A.transpose()
print(A_transponowane)
```

✓ 0.0s

```
Macierz A:
[[1 2 3]
 [4 5 6]]
Transponowana macierz A
[[1 4]
 [2 5]
 [3 6]]
```

Sprawdzanie kształtu macierzy

Aby sprawdzić kształt macierzy możemy użyć komendy np.`shape`(macierz). Zwróci nam ona krotkę (rzędy, kolumny).

Zastosowanie:

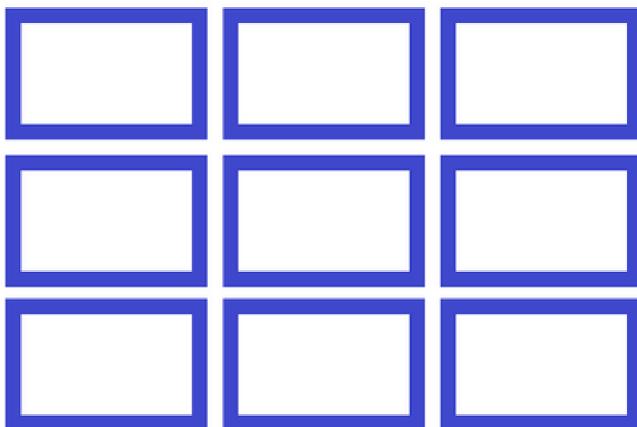
rzedy, kolumny = np.*shape*(macierz)

LUB

print(np.*shape*(macierz))

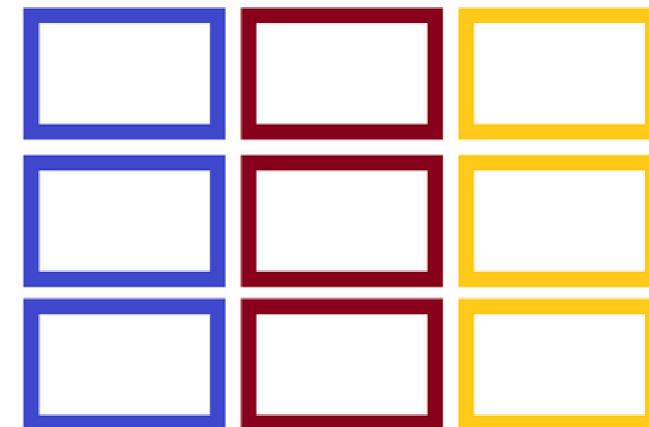
Macierz kontra Data Frame

Macierz



Przechowuje tylko dane numeryczne

Data Frame



Przechowuje zarówno dane numeryczne, jak i stringi lub dane logiczne (boolean type - True, False)



Kontrola przepływu programu

Jeśli wiemy, że wprowadzone do programu wartości mogłyby spowodować błąd, czy da się coś z tym zrobić, albo zwrócić użytkownikowi komunikat, który dokładnie powie co się stało?

Oczywiście że tak!



Kontrola przepływu programu

W tym celu wykorzystuje się blok *try...except*! Jest to konstrukt, który pozwala zdefiniować próbę wykonania programu (*try*), oraz jeśli cos pójdzie nie tak, pozwolić na poprawienie błędu lub przekierowanie kursu programu w celu jego bezpiecznego zakończenia (*except*).



Przyjrzyjmy się temu bliżej

try:

```
x = input("Wprowadź liczbę całkowitą: ")
```

```
x_num = int(x)
```

```
print(f"Liczba Całkowita: {x_num}")
```

```
# Pobierze od użytkownika tylko liczby całkowite, inaczej zwróci  
ValueError
```

except ValueError:

```
print("Wprowadzona wartość nie jest liczbą całkowitą!")
```

```
print("Program zostanie wyłączony!")
```

```
exit(-1) # Wyłącza program z kodem -1(błąd, brak sukcesu, itp.)
```



```
BaseException
├── BaseExceptionGroup
├── GeneratorExit
└── KeyboardInterrupt
└── SystemExit
└── Exception
    ├── ArithmeticError
    │   ├── FloatingPointError
    │   ├── OverflowError
    │   └── ZeroDivisionError
    ├── AssertionError
    ├── AttributeError
    ├── BufferError
    └── EOFError
    └── ExceptionGroup [BaseExceptionGroup]
        ├── ImportError
        │   └── ModuleNotFoundError
        ├── LookupError
        │   ├── IndexError
        │   └── KeyError
        ├── MemoryError
        ├── NameError
        └── UnboundLocalError
    └── OSError
        ├── BlockingIOError
        ├── ChildProcessError
        ├── ConnectionError
        │   ├── BrokenPipeError
        │   ├── ConnectionAbortedError
        │   ├── ConnectionRefusedError
        │   └── ConnectionResetError
        ├── FileExistsError
        ├── FileNotFoundError
        ├── InterruptedError
        ├── IsADirectoryError
        ├── NotADirectoryError
        ├── PermissionError
        ├── ProcessLookupError
        └── TimeoutError
    └── ReferenceError
    └── RuntimeError
        ├── NotImplementedError
        └── RecursionError
    └── StopAsyncIteration
    └── StopIteration
    └── SyntaxError
        └── IndentationError
            └── TabError
    └── SystemError
    └── TypeError
    └── ValueError
        └── UnicodeError
            ├── UnicodeDecodeError
            ├── UnicodeEncodeError
            └── UnicodeTranslateError
    └── Warning
        ├── BytesWarning
        ├── DeprecationWarning
        ├── EncodingWarning
        ├── FutureWarning
        ├── ImportWarning
        ├── PendingDeprecationWarning
        ├── ResourceWarning
        ├── RuntimeWarning
        ├── SyntaxWarning
        ├── UnicodeWarning
        └── UserWarning
```

Typy błędów

Istnieje wiele typów błędów wbudowanych w języku Python, ich definicje i przyczyny znajdzicie w dokumentacji języka:

<https://docs.python.org/3/library/exceptions.html>



Wywoływanie błędów ręcznie

Python umożliwia nam również ręczne wywołanie błędu poprzez wykonanie komendy *raise* (wznieś).

Pozwoli nam to wyłapanie potencjalnie niebezpiecznych lub niepożądanych działań programu podczas jego działania i terminowanie go z podaniem komunikatu dla użytkownika.



Wywoływanie błędów ręcznie

Przykład:

x = 10

y = 0

if y == 0:

raise ZeroDivisionError("Nie można dzielić przez zero!")

else:

print(x/y)



Definiowanie własnych błędów

Możemy też w prosty sposób zdefiniować własny typ błędu, jeżeli spotykamy się z problemem spoza zakresu błędów wbudowanych. Są to dwie linijki kodu wyglądające następująco:

```
class WlasnyBlad(BaseException):  
    pass
```



Kontrola przepływu programu -

finally



Kontrola przepływu programu - *finally*

Do bloku *try...except* można również dodać klauzulę *finally*, która dosłownie oznacza “ostatecznie”. Kod napisany pod nią zostanie wykonany zawsze, nawet jeśli kod spod bloku *try* został wykonany poprawnie.



Kontrola przepływu programu - *finally*

Ponadto, jeśli w bloku try wystąpił błąd który nie jest obsłużony blokiem *except*, błąd tem zostanie wywołany ponownie na końcu kodu w bloku *finally*.



Kontrola przepływu programu - *finally*

Użycie tej klauzuli ogranicza się na dobrą sprawę jako 'clean-up' kodu wykonywanego w blokach *try...except* przy większym ich skomplikowaniu aby zwolnić pamięć zajmowaną przez np. zmienne testowe czy inne dane nie potrzebne do dalszego działania programu.

Kontrola przepływu programu - *finally*

Przykład działania:

```
x = input("Wprowadź dzielną: ")
y = input("Wprowadź dzielnik: ")
try:
    x_num = int(x)
    y_num = int(y)
    if y_num == 0:
        raise ZeroDivisionError(f"Podany dzielnik {y} jest równy zero, nie można kontynuować")
    else:
        print(f"Wynik ilorazu to {x_num / y_num = }")
except ValueError:
    print("Jedna z wprowadzonych danych nie jest liczbą całkowitą. Sprawdź i popraw:\n",
          f"{x = }\n",
          f"{y = }")
    exit(-1)
finally:
    print("Wykonywanie finally")
```



SciByteHub