Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej


Biuro Projektu:
ul.Nadbystrzycka 38H
20 -618 Lublin


# Techniki Eksploracji Danych

*Workbook*


Autor: dr Marcin Bogucki



Lublin, 2021 rok



PROGRAM WIEDZA EDUKACJA ROZWÓJ

Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny

# 01 Typy danych. Funkcje i pamięć środowiska. Zmienne i wyrażenia

**Indeks zagadnień:** środowisko języka Python, klasa, obiekt, konstrukcja programu, słowa kluczowe języka, podstawowe typy danych, zasady nadawania nazw zmiennym, wyrażenia i ich interpretacja, instrukcje iteracji, przykładowy kod programu i jego analiza, błędy w kodzie programu i ich klasyfikacja.

### #01 Wykonanie kodu zapisanego w języku Python

Skrypt języka Python może być uruchomiony w trybie:

- **interaktywnym** - bezpośrednio w środowisku interpretera:
  ```
  ted@x230$ python
  Python 3.8.1 (default, Jan  8 2020, 22:29:32)
  [GCC 7.3.0] :: Anaconda, Inc. on linux
  Type "help", "copyright", "credits" or "license" for more information.
  >>>
  ```
  W linii komend wpisujemy kod programu;
- **skryptowym** - przez interpreter języka Python
  ```
  ted@x230$ python nazwa_modułu.py
  ted@x230$./nazwa_modułu.py
  ```
  o ile w pierwszym wierszu modułu znajduje się dyrektywa #!/usr/sbin/env python, a plik ma nadany atrybut wykonywania (system Linux: chmod -x nazwa.py);

### #02 Poglądowy obraz obiektu w środowisku interpretera języka Python

W trakcie wykonywania programu interpreter przechowuje dane w tzw. **pamięci środowiska.** Reprezentacje struktur danych w pamięci środowiska nazywa się mianem **obiektów**;

Każdy obiekt jest tworzony w pamięci środowiska na podstawie zdefiniowanej wcześniej **klasy** (słowo kluczowe **class**); Można to rozumieć jako kalkę lub wzorzec do tworzenia niezależnych kopii danych.

W innych językach programowania klasa to tzw. **typ danych**. Typ danych definiuje sposób przechowywania danych (ich strukturę) w pamięci programu wraz ze zbiorem dopuszczalnych, adekwatnych operacji, jakie mogą być wykonywane na tychże danych. Wspomniane operacje określa się jako **metody**.

**Klasy** mogą być wbudowane tzn. predefiniowane w środowisku interpretera: np. **int**, **float**, **str, list, tuple, dict**, lub utworzone przez programistę i aktywowane w czasie wykonywania programu. Co interesujące, wszystkie dane przechowywane w pamięci środowiska też są obiektami (moduły, klasy, funkcje, o czym dowiemy się w dalszych rozdziałach opracowania);

### #03 Konstrukcja programu w języku Python

Cztery fundamentalne składowe programu zapisanego w języku Python to:

- **Kod programu** w języku Python jest złożony z szeregu modułów, które z kolei mogą być organizowane w postaci pakietów; Moduły mają postać plików *.py i zawierają kod języka



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

Python; Użycie modułów wymaga ich wcześniejszego importu; Moduły są być importowane do programu/skryptu instrukcją: **import nazwa_modułu** lub **from** nazwa_modułu **import** ...;

- **Moduły** są budowane z **instrukcji** tworzących strukturę programu;
- **Instrukcje** operują na tzw. **wyrażeniach** lub wchodzą w ich skład;
- **Wyrażenia** tworzą, przetwarzają i/lub modyfikują dane zapisywane w **obiektach** reprezentowanych przez **zmienne**, czyli referencje do obiektów;

Do zapisu instrukcji, wyrażeń i definicji funkcji i zmiennych używane są zastrzeżone słowa kluczowe języka Python (patrz: **#5**);

#### #04 Co to jest zmienna i do czego służy?

Aby posługiwać się obiektami i zapisanymi w nich danymi potrzebujemy tzw. zmiennych. W dużym uproszczeniu **zmienna** to nic innego jak nazwa/referencja przypisana do obiektu, która umożliwia posługiwaniem się obiektem: odczytem lub modyfikacją danych przechowywanych w tymże obiekcie. Zmienna jest tworzona w wyniku instrukcji przypisania.

Dla przykładu utwórzmy obiekt, który będzie przechowywać w pamięci środowiska liczbę całkowitą o wartości 1893829. Klasa wbudowana służąca do reprezentacji liczb całkowitych to **int**.

```
>>> calkowita = int(1893829)
>>> calkowita
1893829
```

lub prościej z użyciem tzw. literału:

```
>>> calkowita = 1893829
>>> calkowita
1893829
```

#### #05 Słowa kluczowe języka Python

| False | assert | continue | except | if | nonlocal | return |
|---|---|---|---|---|---|---|
| None | async | def | finally | import | not | try |
| True | await | del | for | in | or | while |
| and | break | elif | from | is | pass | with |
| as | class | else | global | lambda | raise | yield |

Komentarz podczas wykładu lub w zalecanej literaturze.



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

**#06 W środowisko interpretera są wbudowane (tzn. wcześniej zdefiniowane) proste typy liczbowe języka Python:**

- **int** - liczby całkowite;
- **float** - liczby zmiennoprzecinkowe (liczb rzeczywiste);
- **bool** - True/False - wartości logiczne;
- **complex** - liczby zespolone;
- **Decimal** - liczby dziesiętne o stałej precyzji;
- **Fraction** - liczby ułamkowe;

**#07 Wbudowane, złożone typy kolekcji są wykorzystywane do tworzenia sekwencji danych lub rozbudowanych struktur danych:**

- **str** - ciągi znaków (sekwencja znaków/tekstu);
- **list** – lista (może być traktowana jako tablica - sekwencja danych);
- **dict** – słownik (tzw. tablica asocjacyjna);
- **tuple** – krotka (sekwencja dowolnych danych np. typów prostych **int**);
- **set** - zbiór (unikalny zbiór danych np. typów prostych);

**#08 Interpreter wie jak interpretować tzw. literały na podstawie zdefiniowanych konwencji; To tak samo jak posługiwanie się zwykłym kalkulatorem - obowiązuje tu pewna konwencja; LITERAŁ to inaczej „WARTOŚĆ DOSŁOWNA”.**

```
>>> 1234                    # zmienna typu int
>>> 123.452                 # zmienna typu float
>>> ‘Ala Ma Kota’           # ciąg znaków - zmienna typu str
>>> ‘’                      # pusty ciąg znaków - zmienna typu str
>>> 2 + 3j                  # liczba zespolona - zmienna typu complex
>>> 2.01 + 3.14j            #
>>> True                    # literał True oznacza prawdę - zmienna typu boolean
>>> False                   #
>>> (12, 32, ‘A’)           # typ/klasa tuple - tzw. krotka - o tym później
>>> [23, 12, 32]            # typ/klasa list - lista obiektów/wartości
>>> {‘Pi’: 3.141596}        # typ/klasa dict - słownik z kluczem i obiektem/wart.
```

Te i inne konwencje zapisu literałów **należy znać i rozróżniać!** Wszystkie powyższe literały są zamieniane na obiekty stosownych typów/klas danych i umieszczane w pamięci środowiska. Co się dzieje z tymi danymi zapisanymi w pamięci?

**#09 Skoro interpreter zna literały i skojarzone z nimi typy danych, tym bardziej wie, jak interpretować następujące, jawne wywołania konstruktorów obiektów:**

```
>>> int(1234)               # zmienna typu int
>>> float(123.452)          # zmienna typu float
>>> str(‘Ala Ma Kota’)# ciąg znaków - zmienna typu str
>>> str(‘’)                 # pusty ciąg znaków - zmienna typu str
>>> complex(2 + 3j)         # liczba zespolona - zmienna typu complex
>>> complex(2, 3)           #
```



PROGRAM WIEDZA EDUKACJA ROZWÓJ

Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny

01 - Typy danych. Funkcje i pamięć środowiska. Zmienne i wyrażenia;

```
>>> bool(True)              # literał True oznacza prawdę - zmienna typu bool
>>> bool(False)             #
>>> tuple((12, 32,'A'))     # typ/klasa tuple - tzw. krotka - o tym później
>>> list((23, 12, 32))      # typ/klasa list
>>> dict(Pi=3.141596) # typ/klasa dict - słownik z kluczem i obiektem/wart.
```

**#10 Co się dzieje z tymi danymi zapisanymi w pamięci tuż po ich utworzeniu?**

Co się stanie, gdy w linii komend napiszemy:

```
>>> 1999                    # utworzenie obiektu int zawierającego wartość 1999
???
```

lub

```
>>> id(1999)
139918934801840
```

Zostanie utworzony obiekt typu **int** przechowujący wartość **1893829**, a następnie zostanie usunięty z pamięci w tzw. **procesie oczyszczania**; Funkcja id() zwróci adres obiektu int (139918934801840) zawierającego wartość 1999, a następnie obiekt spod adresu 139918934801840 zakończy swe krótkie życie w środowisku interpretera.

**#11 Co się dzieje w pamięci środowiska interpretera w czasie tworzenia zmiennej?**

Rozważmy następującą instrukcję przypisania:

**>>> liczba = 1893829**

Powyższy przykład jest bardzo prosty, ale i bardzo ważny! Warto mieć świadomość tego, co wykonuje interpreter, gdy napotyka w kodzie instrukcję przypisania:

- tworzy obiekt na podstawie definicji klasy/typu **int**;
- tworzy zmienną - nazwę/etykietę przypisaną do obiektu i umieszcza ją w tzw. **ramce przestrzeni nazw**;
- w zmiennej jest przechowywany **adres pamięci**, pod którym można znaleźć obiekt typu **int** wraz z zachowaną wartością **1893829** (w innych językach programowania (np. w języku C) ten **adres** nazywa się mianem **referencji** lub **wskaźnika**);
- dysponując nazwą, mamy dostęp do obiektu typu **int** i jego metod, dzięki czemu interpreter wie jak wykonywać działania na tej zmiennej;

**#12 Zasady nadawania nazw zmiennym:**

- Muszą zaczynać się literą lub znakiem podkreślenia _;
- Mogą zawierać litery, cyfry i znaki podkreślenia;
- Wielkość znaków jest rozróżnialna;

**Przykłady:**
Właściwe : liczba , _liczba, _01_liczba



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

Niepoprawne : 01_liczba, $liczba, @liczba
Rozróżnialność : Liczba , LICZBA, lICZBA

**#13 Konwencje nadawania nazw zmiennym**

- stałe zapisujemy dużymi literami:
  **SYSTEM_OPERACYJNY** = 'Linux version 5.11.4'
- w przypadku długich nazw zmiennych używamy separatora '_' jako separatora:
  **system_operacyjny** = 'Windows 10 Professional'
- notacja **camelCase**, w której kolejne wyrazy nazwy zmiennej pisane są łącznie, rozpoczynając każdy kolejny wyraz poza pierwszym - wielką literą:
  **systemOperacyjny =** 'DOS (Disc Operating System) 5.0'
- nazwy zmiennych odnoszone do indeksów są jednoliterowe, zgodne z konwencją używaną w matematyce np. **i**, **j**, **k**. Tych nazw używamy w zazwyczaj w pętlach/iteracjach;
- inne specyficzne dla języka Python: **key**, **value**, **args**, **kwargs**, występujące w pewnych specyficznych konstrukcjach (przykłady na kolejnych wykładach);

To są tylko konwencje, ... ale warto je przestrzegać; Dzięki temu nasz kod będzie bardziej czytelny dla innych programistów;

**#14 Jak zarządzać/zachować porządek nad zmiennymi (ich nazwami)?**

Środowisko interpretera posiada wbudowane (czytaj: wcześniej zdefiniowane) funkcje, które pozwalają na zarządzanie stanem zmiennych, ich odczytem, wyświetlaniem wartości przechowywanych w obiekcie i wiele innych.

Przykłady nawiązujące do wcześniej utworzonej zmiennej:

**id**(liczba) - numer/adres pamięci pod którym znajduje się obiekt typu **int;**
**type**(liczba) - informacja o typie danych obiektu wskaz. przez zm. **liczba;**
**dir**() - wyświetl. nazwy zmiennych w aktualnej ramce przest. nazw;
**help**(liczba) - tu: informacje o sposobie użycia typu **int;**
**print**(liczba) - tu: wyświetli w konsoli wartość: **1893829;**
**input**() - liczba = int(input()); pobranie ciągu znaków (literału);
**del**(liczba) - usunięcie zmiennej i obiektu ze środowiska;

Na tę chwilę potraktujmy te funkcje jako obiekty zawierające kod umożliwiający wykonanie operacji na innych obiektach;

**#15 Moduł główny**

Zmienne są tworzone w ramach modułów. Program w języku Python rozpoczyna swe działanie w tzw. module głównym o nazwie: __main__;

```
>>> dir()
>>> __name__   # lub print(__name__)
'__main__'
```



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

**#16 Organizacja zmiennych**

Zmienne są tworzone w ramce bieżącego modułu lub modułu, który zostanie zaimportowany. Import polega utworzeniu w pamięci środowiska obiektu reprezentującego kodu bajtowy (binarny) modułu i nadaniu mu nazwy zdefiniowanej w tymże module; Przykład wczytania modułu 'math':

```
>>> import math              # import modułu math
>>> dir(math)
```

Warianty tej instrukcji zostaną omówione szczegółowo później, podczas rozważania organizacji modułów i programu;

**#17 Obiekty wbudowane nie wymagają jawnego importu**

Obiekty wbudowane są dostępne dla programisty bez potrzeby ich jawnego importu z innych modułów i wchodzą w skład każdego programu zapisanego w kodzie języka Python. Wywołajmy następującą funkcję:

```
>>> dir(__builtins__)
```

Zostaną wyświetlone inne wbudowane obiekty oraz funkcje wbudowane w środowisko.

**#18 Operacje numeryczne - budowanie elementarnych wyrażeń (arytmetycznych) z wykorzystaniem zmiennych i operatorów typów liczbowych;**

Typy liczbowe int, float (i inne: complex, Decimal, Fraction) posiadają metody, w których zdefiniowano działania operatorów arytmetycznych:

- + - (__add__) dodawanie
- - - (__sub__) odejmowanie;
- * - (__mul__) mnożenie;
- / - (__div__) dzielenie;
- ** - (__pow__) eksponent;
- % - (__mod__) reszta z dzielenia;
- // - (__floordiv__) dzielenie całkowite;
- ... i wiele innych;

Wywołaj w środowisku funkcję: dir(float) lub dir(int) i zwróć uwagę na listę metod tego typu (lub obiektu tego typu);

```
>>> dir(int)   # wybrano tylko niektóre metody nie zachowując kolejności

[...__abs__, __add__, ... , __mod__, __pow__, __str__, __sub__]

>>> a = 23; b = 300
>>> a + b                    # zapis: a + b jest tożsamy z: a.__add__(b)
>>> 323                      # i jest wywołaniem metody __add__ typu/klasy int
```



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

**#19 Kolejność wykonywania operacji arytmetycznych**

Gdy zapisujemy złożone wyrażenia arytmetyczne, interpreter stosuje działania wynikające z symboli operatorów arytmetycznych. Działania te są wykonywane w następującej kolejności:

- () - działania zgrupowane w nawiasach ();
- \- - negacja (znak stoi przed liczbą)
- ** - potęgowanie **działa od prawej do lewej strony wyrażenia**
- *,/ - mnożenie i dzielenie;
- +,- - dodawanie i odejmowanie;
- działania są interpretowane **od lewej do prawej** strony wyrażenia oprócz wyjątku, jakim jest operator ******;

Przykłady:

```
>>> 3 + 2**5 / 4*2 + 5
>>> 24.0
>>> (3 + 2)**5 / (4*2 + 5)
>>> 240.3846153846154
>>> -2**4
>>> -16
```

W przypadku wątpliwości co do kolejności wykonywanych działań używajmy nawiasów. Taką samą konwencję stosujemy w arytmetyce.

**#20 Przypadek złożonych wyrażeń**

Złożone wyrażenia najlepiej jest podzielić je na proste człony, a wyniki pośrednich obliczeń zapamiętać w zmiennych;

```
>>> ((3 + 2)**5 + 123) / ((4*2 + 5)**2 – 234)
-49.96923076923077
```

można zapisać jako:

```
>>> a = (3 + 2)**5 + 123
>>> b = (4*2 + 5) – 234
>>> a/b
>>> -49.96923076923077
```

**#21 Konwersje typu zamiennych: jawna i niejawna;**

Aby wykonać operacje arytmetyczne, obiekty muszą być tego samego typu. Rozważmy kolejny prosty przykład:

```
>>> 5/3
>>> 1.6666666666666667
```

Jak widzimy, liczby w wyrażeniu są typu całkowitego **int**, natomiast wynik obliczeń jest liczbą zmiennoprzecinkową typu **float.** Jest to przykład **niejawnej** konwersji typu. Gdybyśmy chcieli



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

wykonać dzielenie całkowite, należałoby użyć operatora // lub jawnej konwersji wyniku do typu **int**.

```
>>> 5//3
>>> 1
>>> int(5/3)
>>> 1
```

**#22 Konwersje typu zamiennych (jawna i niejawna) - drugi przykład;**

```
>>> 5 + '3.14159'                    # tu występuje niezgodność typów danych
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Zapiszmy to inaczej:

```
>>> 5 + float('3.14159')      # jawna konwersja typu str do typu float
>>> 8.14159
>>> 5 + float('A3.14159')     # próba konwersja typu str do typu float
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'A3.14159'
```

Inne przykłady:

```
>>> str(5) + '3.14159'        # jawna konwersja typu int do str
>>> 53.14159                  # operator + dodawania sumuje ciągi znaków
```

**#23 Komentarze w kodzie;**

- po znaku # wszystkie kolejne znaki w kodzie programu są ignorowane;
- ‘’’ ... ‘’’ lub ””” ... ””” - potrójny cudzysłów może służyć za komentarz, ale najczęściej jest wykorzystywany do zapisu długich ciągów znaków lub do komentarza dotyczącego dokumentacji funkcji, modułu, klasy (tzw. docstring);

**#24 Przykład programu/modułu ilustrującego pojęcia przedstawione w skrypcie**

Zawartość pliku trojmian.py:

```
#!/usr/bin/env python
# Powyższa dyrektywa powinna być pierwszą linią programu. Jeżeli ten skrypt będzie
# wykonywany z wiersza poleceń, system odszuka i uruchomi interpreter języka python
# i wykona ten skrypt;
import math

if __name__ == "__main__":
    """
    Program oblicza miejsca zerowe trójmianu kwadratowego
    Wejście: współczynniki trójmianu: a, b, c;
```



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

```
    Wyjście: x11, x2: miejsca zerowe funkcji;
    """
    # pobierz dane do obliczeń
    print('Podaj warości współczynników trójmianu: ax**2 + b*x +c')
    a = float(input('Wartość współczynnika a = '))
    b = float(input('Wartość współczynnika b = '))
    c = float(input('Wartość współczynnika c = '))

    # oblicz wyznacznik ...
    delta = b**2 – 4*a*c
    # ... i wyznacz rozwiązania
    if delta >= 0:
        if delta == 0:
            # równanie kwadratowe ma tylko jedno rozwiązanie
            x_0 = -b/(2*a)
            print("Równanie kwadratowe ma tylko jedno rozwiązanie:", x_0)
        else:
            # równanie ma dwa rozwiązania
            x_1 = (-b – math.sqrt(delta))/(2*a)
            x_2 = (-b + math.sqrt(delta))/(2*a)
            print("Równanie kwadratowe ma dwa rozwiązania:", x_1, x_2)
    else:
        print("Równanie kwadratowe nie ma rozwiązań w dziedzinie liczb rzeczywistych")
```

### #25 Czy ten program poprawnie szacuje miejsca zerowe trójmianu?

Spróbujmy uruchomić program, by przekonać się, jakie otrzymamy wyniki dla przykładowych danych wejściowych: a, b, c;

```
Podaj wartości współczynników trójmianu: ax**2 + b*x +c
Wartość współczynnika a = 1
Wartość współczynnika b = 5
Wartość współczynnika c = 2
Równanie kwadratowe ma dwa rozwiązania: -4.561552812808831 -0.4384471871911697
```

### #26 Sytuacje krytyczne i ich obsługa

O ile użytkownik programu poda właściwe wartości współczynników, program wykonuje się poprawnie. Ale dla przykładu, gdy wartości współczynników wynoszą:

a = 1, b = 5, c = A,jako wynik otrzymamy:

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-0c206e301fb2> in <module>
     17     a = float(input('Wartość współczynnika a = '))
     18     b = float(input('Wartość współczynnika b = '))
---> 19     c = float(input('Wartość współczynnika c = '))
     20 
     21     # oblicz wyznacznik równania
ValueError: could not convert string to float: 'A'
```

Jest to przykład **błędu wykonania**. Funkcja input() wczytująca wartość trzeciego współczynnika wygenerowała tzw. **wyjątek ValueError**, co oznacza tu niezgodność literału z typem danych float. Tej sytuacji można uniknąć przez obsługę wyjątku (co będzie wyjaśnione podczas dalszych wykładów).

... lub, gdy wartości współczynników wynoszą:

```
a = 0, b = 1, c = 2          # funkcja liniowa i rozwiązanie: -c/b = -2erotic photos
a = 0, b = 0, c = 0          # tożsamość 0 = 0
```

```
Podaj wartości współczynników trójmianu: ax**2 + b*x +c
Wartość współczynnika a = 0
Wartość współczynnika b = 1
Wartość współczynnika c = 2 ---------------------------------------------------------------------------
ZeroDivisionError                          Traceback (most recent call last)<ipython-input-6-961e0787bf60>
in <module>
     47         if delta == 0:
     48             # równanie kwadratowe ma tylko jedno rozwiązanie
---> 49             x_0 = -b/(2*a)
     50             print("Równanie kwadratowe ma tylko jedno rozwiązanie:", x_0)
     51         else:
ZeroDivisionError: float division by zero
```

... a tego nie przewidziano w programie. Tu właśnie mamy do czynienia z **błędem sematycznym**, wynikającym z tego, że programista nie rozważył innych rozwiązań (tożsamości 0 = 0, sprzeczności c = 0 i funkcji liniowej). Jak poprawić ten błąd? Błędy semantyczne są najtrudniejsze do wykrycia w programie.

### #27 Typy błędów występujące w programie i ich śledzenie

- **błędy składni** (niezgodność ze składnią języka) są komunikowane przez interpreter i są zazwyczaj oczywiste (dla osób znających składnię);
- **błędy wykonania** (niezależne od programu, wynikające z sytuacji zewn.) prowadzące do tzw. sytuacji wyjątkowych; Są wyzwalane przez środowisko PVM o ile nie są obsłużone przez programistę;
- **semantyczne** występują kiedy kod jest wprawdzie poprawny ze względu na składnię, ale wynik jego działania nie jest zgodny założeniami projektowymi oprogramowania. Błędy semantyczne są najtrudniejsze do wykrycia/zauważenia; Mogą być wykryte przez testy jednostkowe lub narzędzia pozwalającym na etapowe, krokowe wykonywanie programu (debuger);

### #28 Pytania kontrolne

1. Za co odpowiada środowisko interpretera języka Python?
2. Co rozumiesz przez pojęcie klasa?
3. Wymień podstawowe typy danych służących do reprezentacji liczb całkowitych i zmiennoprzecinkowych.

01 - Typy danych. Funkcje i pamięć środowiska. Zmienne i wyrażenia;

4. W jaki sposób jest tworzony obiekt w środowisku języka Python?
5. Co to jest zmienna i jaki jest jej związek z obiektem?
6. Wymień podstawowe instrukcje wykorzystywane do zapisu kodu w języku Python.
7. Z czego składa się wyrażenie i jakie ma znaczenia w zapisie kodu programu?
8. Co to jest konwersja typu zmiennych i kiedy zachodzi?
9. Wymień trzy sposoby umieszczania komentarzy w kodzie.
10. Co to jest DocString i do czego jest wykorzystywany?
11. Jaka jest klasyfikacja błędów występujących w oprogramowaniu?
12. Jakie funkcje środowiska języka Python są używane do zarządzania zmiennymi i obiektami?



PROGRAM WIEDZA EDUKACJA ROZWÓJ

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

## 02 Typ logiczny. Operacje logiczne. Instrukcje warunkowe. Iteracje

**Indeks zagadnień:** typ logiczny, interpretacja wyrażeń logicznych, operatory logiczne, kolejność wyliczania złożonych wyrażeń, wyliczanie leniwe, grupowanie bloków instrukcji, instrukcja ‘if’, ‘in’, ‘pass, analiza kodu programu wskazująca złe praktyki zapisu programu.

### #01 Typ danych reprezentujący wartości logiczne

W języku Python wartości logiczne są reprezentowane przez typ **bool.** Obiekty tego typu mogą przyjmować jedynie dwie wartości:

- **True** # równą 1, oznaczającą PRAWDĘ oraz ...
- **False** # równą 0, oznaczającą FAŁSZ

```
>>> type(Type)                 # wynik: <class ‘bool’>
>>> int(True)                  # wynik: 1
>>> int(False)                 # wynik: 0
>>> bool(79.35)                # wynik: True
>>> bool(0.0), bool(‘’)        # wynik: False, False
```

Funkcja/konstruktor bool() konwertuje wartość obiektu do wartości typu bool.

### #02 Z każdym typem danych jest związana wartość logiczna: True lub False

W języku Python obowiązuje następująca umowa dotycząca wartości logicznych: wszystkim obiektom danych, które mają posiadają **niezerową wartość, przyporządkowana** jest wartość logiczna **True**; Podobnie, każda niepusta sekwencja danych (tuple, str, list, dict), choćby nawet zawierała tylko wartości zerowe, ma przypisaną wartość **True** - w przeciwnym bądź razie, wartość logiczną **False**;

```
>>>  a = 23; b = 70.34; c = 0
>>> bool(a), bool(b), bool(c);       # wynik: (True, True, False)
>>> bool(‘Ala ma kota’), bool(‘’)    # wynik: (True, False)
```

### #03 Operatory logiczne języka Python działają na wyrażeniach logicznych

Podobnie jak w innych językach programowania wyróżnia się trzy operatory logiczne pozwalające na łączenie wyrażeń logicznych w zdania:

- **not** # negacja
- **and** # koniunkcja
- **or** # alternatywa

**WAŻNA UWAGA**: Operatory **and** i **or** zwracają wynik/wartość wyrażenia logicznego (**True**, **False**) **o ile wyliczana/zwracana wartość jest typu logicznego**. Jeżeli tak nie jest, operatory te zwracają wartość obiektu, z którym związania jest ostatnio wyliczana wartość logiczna. Wyliczanie wyrażenia opiera się na regułach tzw. „leniwego oszacowania”.



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

### #04 Operatory relacji (porównania danych)

W ramach elementarnych typów danych języka Python zdefiniowano operatory/metody relacji pozwalające na wykonywanie porównań wartości obiektów. Należą do nich następujące operatory:

| # operator | # komentarz | # przykład | # wynik |
|---|---|---|---|
| > | # większy niż warość | 10 > 1 | True |
| < | # mniejszy niż wartość | 10 < 1 | False |
| >= | # większy lub równy wartości | 10 >= 9 | True |
| <= | # mniejszy lub równy wartości | 10 <= 9 | False |
| == | # równy wartości | 'Ala' == 'Bela' | False |
| != | # różny od wartością | 'A' > 'B' | False |

Te operatory zwracają wartości logiczne;

Dalsze przykłady:

```
>>> a = 23; b = 70.34
>>> a < b            # wynik: True - tu wykonywana jest niejawna konwersja typów
>>> a == b           # wynik: False
>>> a < b <= 230     # ... ciekawy przykład równoważny z ...
True                 # (a < b) and (b <= 230)
```

### #05 Kolejność wyliczania wartości operatorów relacji…

Wszystkie operatory relacji mają ten sam priorytet wyliczania/oszacowania wartości i są analizowane od lewej strony wyrażenia do prawej. Wyjątkiem jest podwójna relacja:

**a** R1 **b** R2 **c**

równoważna:

(**a** R1 **b**) **and** (**b** R2 **c**)

gdzie **R1** i **R2** to dwa dowolne operatory relacji spośród: >, <, ... i/lub pozostałych.

```
>>> a = 23; b = 70.34
>>> a < b <= 230                  # wynik: True
>>> (a < b) and (b <= 230)    # 'and' to op. logiczny łączący dwa wyr. relacji
```

### #06 Zestawienie operatorów arytmetycznych, relacji oraz logicznych

Kolejność występowania operatorów w tabeli wyznacza jednocześnie priorytet ich wyliczania. W pierwszej kolejności wykonywane są operacje arytmetyczne, później operacje relacji, i ostatecznie, operacje logiczne;

| # operatory | # komentarz | # kierunek obliczania |
|---|---|---|
| ** | eksponent (potęgowanie); | prawy do lewego (wyjątek) |
| - | negacja; | lewy do prawego |
| *, /, //, % | mn., dz., dz.cał, dz.modulo; | lewy do prawego |
| +, - | dod., odejm. | lewy do prawego |
| <, >, <=, >=, !=, == | operatory relacji | lewy do prawego |
| not | negacja | lewy do prawego |



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

| | | |
|---|---|---|
| **and** | **koniunkcja** | **lewy do prawego** |
| **or** | **alternatywa** | **lewy do prawego** |

**Warto zwrócić uwagę,** że priorytet operatorów logicznych nie jest jednakowy. Przy braku nawisów w wyrażeniu kolejność jest następująca: **not,** w drugiej kolejności **and**, a w ostatniej kolejności operator **or**. W przypadku niejednoznaczności skomplikowane ciągi wyrażeń połączonych operatorami arytmetycznymi, relacji i logicznymi powinno zapisywać się w nawiasach okrągłych ”()” wyznaczającymi kolejność ich wyliczania;

**#07 Kombinacje wyrażeń arytmetycznych, relacji i logicznych oraz ... tzw. leniwe oszacowanie wyrażeń logicznych**

Przyjrzyjmy się następującym kombinacjom wyrażeń:

```
>>> False or True and False               # wynik: False, co jest równoważne:
>>> False or (True and False)
>>> True  or True and False               # True and False nie jest wyliczane
>>> False or True and False or True       # wynik: True,  co jest równoważne:
>>> (False or True) and (False or True)
>>> 5 != 0 and 10 == 0 and 0 != 0         # wynik: False
>>> 5 and 10 and 0                        # tu niespodzianka: 0, ale nie False
>>> 5 and 10 or 0                         # wynik: 10 - ost. wyr. nie jest ew.
>>> not (10 < 0 or 5 + 5**2)              # wynik: False
>>> not 10 < 0 and 5 + 5**2               # wynik: 30
>>> not 10 < 0 and 5 + 5**2 > 0           # wynik: True
```

Operatory logiczne zwracają wartość obiektu, z którym związania jest ostatnio wyliczana wartość logiczna, co nazywa sięmianem „leniwego oszacowania”. UWAGA: **oszacowanie leniwe stosuje się wyłącznie do wyrażeń logicznych !**

**#08 Bloki instrukcji w języku Python**

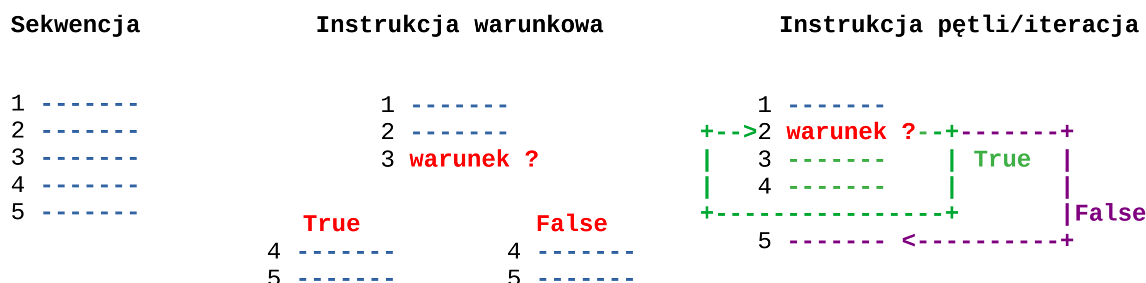
Dotychczas wykorzystywaliśmy jedynie instrukcję **przepisania** oraz **sekwencję**. Do innych instrukcji należą m. in.:

- instrukcja warunkowa **if**; — # wybór ścieżki wykonania kodu
  ```
  if warunek:                             # warunek == wyrażenie logiczne
          instrukcja(-e)
  ```
- iteracja - pętla określona **for**; — # przetwarzanie sekwencji
  ```
  for element in sekwencja elementów:     # elementów: list, tuple, dict
          instrukcja(-e)
  ```
- iteracja - pętla nieokreślona **while**; — # przetwarzanie elementów, aż do
  ```
  while warunek:                          # wystąpienia wartości False
          instrukcja(-e)
  ```
- instrukcja zawierania **in** — # zwraca wartość **True** lub **False**
  ```
   element in sekwencja elementów
  ```
- inne np: instrukcja if trój-wyrażeniowa : w1 **if warunek else** w2 zwracająca wartość w1 lub w2



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

Sekwencja, instrukcja warunkowa oraz iteracja mogą, ale nie muszą grupować inne instrukcje w bloki. Niech ciąg znaków ------- oznacza dowolną instrukcję.




UWAGA: **język Python ma specjalną konwencję zapisu bloków kodu polegającą na tym, że sekwencje instrukcji zapisywane są wiersz po wierszu w tym samym poziomie wcięć** (ang. indentation - tłum. wcięcie). Niezachowanie tej konwencji prowadzi do generowania błędów składni języka;

**#09 Instrukcja warunkowa „if”**

Instrukcja if jest wykorzystywana do warunkowego wykonania kodu. Jeżeli wartość wyrażenia logicznego stojąca po „if” jest prawdziwa, wykonywany jest bezpośrednio występujący po nim blok instrukcji. Jeżeli w instrukcji warunkowej występuje dodatkowo słowo kluczowe „else”, a pierwotny warunek nie jest spełniony, wykonywany jest blok instrukcji występujący po „else”.

**Konstrukcja**

```
if warunek:
    -------
else:
    -------
```

**Przykłady**

```
if 36.3 < temp <= 36.6:
    print('Zdrowy pacjent')
else:
    print('Możliwa gorączka?')
```

```
# jednoblokowa instrukcja if

if temp > 41.0:
    print('Pacjent jest ugotowany')
```

Drugi człon instrukcji warunkowej jest opcjonalny. Instrukcje warunkowe mogą się zagnieżdżać, jak również mogą stanowić bloki innych instrukcji (np. if, while, for, in i innych).

**#10 Wcięcia bloków instrukcji**

Instrukcje w pierwszym i drugim bloku if (else) muszą być wcięte o tą samą liczbę pustych znaków (tabulacja lub spacja). Preferowanym sposobem akcentowania wcięć są spacje (w liczbie czterech). Nie można mieszać ze sobą w kodzie tabulacji i spacji.

**Prawidłowo**

```
if warunek:
    -------
    -------
else:
```

**Nieprawidłowo**

```
if warunek:
    -------
    -------
else:
```

```
if warunek:
    -------
    -------
  else:
```

```
if warunek:
      -------
      -------
else:
```



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

**#11 Wariant instrukcji warunkowej ze słowem kluczowym 'elif'**

W innych, proceduralnych języka programowania np. w języku C występuje instrukcja, nie alternatywnego, lecz wielokrotnego wyboru bloku kodu (switch/case/default). W języku Python taki wielowarunkowy zapis kodu można osiągnąć przez modyfikację instrukcji „if" dodatkowym słowem kluczowym „elif".

```
if warunek_0:               # Jeżeli pierwszy warunek_0 jest spełniony
    --------_0              # wykona się --------_0, a następnie --------_n+2;
elif warunek_1:             # W przeciwnym bądź razie będą testowane kolejne
    --------_1              # instrukcje warunkowe w tym miejscu: warunek_1;
elif warunek_2:
    --------_2
...                         # Dalsze, ewentualne instrukcje elif;
elif warunek_n:
    --------_n
else:                       # Jeżeli żaden z warunków z bloków elif nie jest spełniony
    --------_n+1            # wykona się domyślny blok else: --------_n+1
--------_n+2
```

**#12 Instrukcja pusta 'pass' i jej zastosowania**

Instrukcja '**pass**'jest tzw. instrukcją pustą, z którą nie są związane żadne działania ze strony interpretera. Programista umieszcza ją w kodzie tuż po złożonej instrukcji (\np. if, while, for), która po znaku ':' według reguł składni wymaga przynajmniej jednej, dodatkowej instrukcji.

```
if temp > 60:                # lub if temp > 60: pass
    pass

while warunek:             # pętla jest tylko oczekiwaniem na negację
    pass                   # warunku logicznego 'warunek' (kodowanie wątków)
```

Oprócz powyższych przykładów programiści używają tej instrukcji do tymczasowego zaznaczenia obligatoryjnych części kodu, które na daną chwilę jeszcze nie zostały wprowadzone do programu. Ich brak może uniemożliwiać uruchomienie fragmentu kodu (i jego ewentualne testowanie).

## #13 Instrukcja 'in' - badanie przynależności elementu do sekwencji elementów

```
element in sekwencja_elementow       # składnia instrukcji przynależności
```

Ten rodzaj instrukcji jest wykorzystywany do przetwarzania koelkcji elementów (tuple, list, str i innych) i zwraca wartość logiczną **True**, o ile wartość obiektu należy sekwencji przeszukiwanych elementów. Ta instrukcja jest często składową instrukcji warunkowych if/elif.



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

Bez tej instrukcji, przeszukiwanie elementów wprowadzałoby pewien narzut w postaci dodatkowego kodu.

```
primes = [1, 3, 5, 7, 11, 13, 17]    # sekwencja elementów listy
number = 7

if number in primes:                 # czy liczba jest w liście ?
     print(‘Liczba pierwsza z zakresu 1 ... 17’)
```

**#14 ... inny przykład wykorzystania instrukcji ‘in’**

```
option  = (‘Tak’, ‘tak’, ‘TAK’, ‘Nie’, ‘nie’, ‘NIE’, ‘n’, ‘N’)
choose_one  = input(‘Czy zgadzasz się na przedstawione warunki licencji?’)

if  choose_one in option:
    print(‘Wybrano:’, choose_one)
elif:
    print(‘Nie ma innej opcji niż Tak lub Nie)
```

Instrukcja **in** występuje także w pętli for wykorzystywanej do przetwarzania sekwencji obiektów, o czym będzie mowa w dalszych skrypach.

### #15 Przykład kodu wykorzystującego instrukcję warunkową typu ‘if/elif/else’ oraz instrukcje ‘in’ oraz ‘pass’

```
#!/usr/bin/env python

# informacja dla użytkownika programu
#
print('Program dokonuje konwersji jednostek temperatury w trzech skalach:')
print('(C) Celcjusza')
print('(K) Kelwina')
print('(F) Fahrenheita')
print('Wybierz skalę, w której jest wyrażony twój aktualny pomiar temperatury C/K/F\n')

# wybór skali do konwersji jednostek oraz pobranie wartości temperatury
#
scale = input('Podaj oryginalną skalę temperatury: C, K, lub F: ')
temp  = float(input('Podaj aktualne wskazanie temperatury: '))

# konwersja jednostek i wyświetlenie wyniku
#
if scale == 'C':
    conv_to_Fahr = 9/5 * temp + 32
    conv_to_Kelv = temp + 273.1
    print('Aktualny pomiar temperatury {:06.2f} C, to:'.format(temp))
    print('{:06.2f} F, w skali Fahrenheita i ...'.format(conv_to_Fahr)
    print('{:06.2f} K, w skali Kelwina;'.format(conv_to_Kelv))
elif scale == 'K':
    conv_to_Celc = 273.15 – temp
```

```
    conv_to_Fahr = 9/5 * conv_to_Celc + 32
    print('Aktualny pomiar temperatury {:06.2f} K, to:'.format(temp))
    print('{:06.2f} F, w skali Fahrenheita i ...'.format(conv_to_Fahr))
    print('{:06.2f} C, w skali Celcjusza'.format(conv_to_Celc))
elif scale == 'F':
    conv_to_Celc = (temp - 32) * 5/9
    conv_to_Kelv = temp + conv_to_Celc
    print('Aktualny pomiar temperatury {:06.2f} F, to:'.format(temp))
    print('{:06.2f} K, w skali Kelwina i ...'.format(conv_to_Kelv)
    print('{:06.2f} C, w skali Celcjusza'.format(conv_to_Celc))
elif scale in 'AB DE GHIJ LMNOPRST'         # przykład instrukcji 'in' i 'pass'
    pass
else:
    print('\nWybrano niewłaściwą skalę temperatury.')
    print('Aktualny pomiar to: {:06.2f} w twojej własnej, unikalnej skali'.format(temp))
```

**Co jest złą praktyką w powyższym kodzie:**
(1) Łamana jest zasada **DRY** (**D**on't **R**epeat **Y**ourself - tłum. nie powtarzaj się)**!** W programie występują wielokrotnie powtarzające się sekwencje (winny być zastąpione funkcjami);
(2) Kod nie zawiera żadnej kontroli wprowadzanych danych (np. blok: **try:** ... **except:** ...);
(3) Nie ma żadnych ograniczeń na niskie wartości temperatury poniżej -273.17 C (0 K);
(4) Proszę zwrócić uwagę na bloki instrukcji i na warunki logiczne w wyrażeniach;
(5) Podobnie: wyrażenia arytmetyczne; (6) Instrukcje formatujące wartości obiektów numerycznych; (6) Brak testów jednostkowych, weryfikujących poprawność semantyczną poszczególnych części programu;
(7) Pobieranie wartości temperatury, pomimo że skala może być niewłaściwa;
(8) Instrukcje: 'in' i 'pass' i ich użycie;
(9) Konwersja wyniku (logika i arytmetyka programu) powinna być oddzielona od prezentacji danych - tu: nie jest wprawdzie to błąd krytyczny; W przypadku bardziej rozbudowanych programów może to utrudnić ich konserwację;

**#16 Instrukcja iteracji 'while' (pętla nieokreślona)**

Oprócz sekwencji, instrukcji warunkowej oraz funkcji instrukcja iteracji jest podstawowym budulcem kodu i jest wykorzystywana do ponownego powtarzania tego samego bloku instrukcji. W języku Python iteracje występują w dwóch odmianach:

- instrukcji **while**, tzw. pętli nieokreślonej;
- instrukcji **for**, nazywanej odpowiednio, pętlą określoną;

Druga z instrukcji zostanie przedstawiona przy okazji dyskusji kolekcji elementów, gdyż tak naprawdę taki jest zwykle cel stosowania instrukcji 'for'.

Instrukcja **while** ma następującą konstrukcję:

**Konstrukcja instrukcji while - pierwsze, proste podejście**

```
while warunek:          # test/sprawdzenie warunku logicznego
    -------             # jeżeli warunek jest spełniony (warunek == True)
    -------             # wykonywane są instrukcje bloku kodu
    -------
-------                 # warunek nie jest spełniony (wartość == False), dlatego
-------                 # wykonywane są dalsze instrukcje występ. po bloku while
```

gdzie ‘warunek’ jest wyrażeniem logicznym. Instrukcje znajdujące się w bloku instrukcji **while** są wykonywane dopóty, dopóki wyrażenie logiczne jest spełnione (wartość logiczna wynosi ‘True’). W przeciwnym razie wykonywana jest kolejna instrukcja po bloku **while**.

Przykłady użycia pętli **while**:

```
while True:          # Pętla nieskończona: tego nie powinno się robić, bo ...
    pass             # ... program zwiesi się program;
                     # Awaryjne wyjście z programu - kombinacja: <Ctrl+C>
```

**#17 Rozbudowana wersja instrukcji iteracji ‘while’**

Instrukcji **while** może towarzyszyć opcjonalna instrukcja **else**, która jest wykonywana, wtedy i tylko wtedy, gdy warunek w instrukcji while zmieni się z True na False. Instrukcja po członie **else** nie jest wykonywana, jeżeli pętla jest opuszczana przez zastosowanie instrukcji **break** - patrz dalej.

Blok instrukcji występujący po else jest najczęściej stosowany do wykonania instrukcji po prawidłowo zakończonej pętli while.

W pętli **while** mogą również występować dwie instrukcje: **break** i/lub **continue**. Pierwsza z nich powoduje wyjście z pętli bez sprawdzenia warunku, następnie interpreter wykonuje pierwszą kolejną instrukcję spoza bloku while. Instrukcja **continue** powoduje zignorowanie kolejnych instrukcji występujących w bloku while i powrót sprawdzenia warunku logicznego pętli.

Poniżej zilustrowano konstrukcję pętli while z opcjonalnym członem else oraz instrukcjami break i continue.

**Konstrukcja instrukcji while - drugie podejście**

```
-------              # Pierwsza instrukcja przed wejściem do pętli;
while warunek:       # Wejście do pętli. Test/sprawdzenie warunku logicznego;
    -------          # Jeżeli warunek jest spełniony (warunek == True) to
    -------          # wykonywane są instrukcje bloku kodu;
    -------
    if warunek_a:    # Wewnątrz pętli sprawdzany jest warunek_a, który może
        break        # aktywować instrukcję break, która powoduje wyjście
    -------          # z pętli; instrukcji break może występować bez if;
    -------          #
    if warunek_b:    # Jeżeli warunek_b to wykonywanie pętli wraca do
        continue     # punktu wejścia; instrukcji continue nie musi towarzyszyć
```

```
    -------          # konstrukcja if;
    -------          #
else:                # Ta instrukcja else należy do konstrukcji while i jest
    -------          # wykonywana, gdy warunek nie jest już spełniony (po
    -------          # prawidłowym opuszczeniu pętli - bez instrukcji break);
-------              # Wyjście z pętli while.  Warunek nie jest spełniony
-------              # (wartość == False), dlatego  wykonywane są dalsze
-------              # instrukcje występępujące po bloku while
```

**#18 Przykład użycia petli while z instrukcjami break i continue:**

```
while True:
    leave_while = input('Czy wyjść z pętli? : ')
    if leave_while == "Tak":
        break
    if leave_while == "Nie":
        continue
        print('Jeżeli nie podano ciągu: ”Nie”, ten komunikat powinien się wyświetlić.')
else:
    print('Warunek nadal aktualny (True), dlatego ta linia kodu nigdy się nie wykona.')
    print('... a dlaczego ? Bo wyjście z pętli powoduje tylko instrukcja break.')

print('... ale ta instrukcja już się wykona.')
```

**#19 Udoskonalona wersja programu do przeliczania jednostek temperatury**

Każdy cyklicznie działający program musi być wykonywany w pętli. Zmieńmy strukturę programu do przeliczania jednostek, przedstawiony innym opracowaniu, tak by był wykonywany wielokrotnie przez użytkownika. W tym celu wzbogaćmy go o pętlę **while**, w której wprowadzimy warunek logiczny decydujący o jego ponownym wykonaniu.

```
#!/usr/bin/env python

print('Program dokonuje konwersji jednostek temperatury w trzech skalach:')
print('(C) Celcjusza')
print('(K) Kelwina')
print('(F) Fahrenheita')
print('Wybierz skalę, w której jest wyrażony twój aktualny pomiar temperatury C/K/F\n')

while True:
    scale = input('Podaj oryginalną skalę temperatury: C, K, lub F: ')
    temp  = float(input('Podaj aktualne wskazanie temperatury: '))

    # blok kalkulacji jednostek i wyświetlania wyników obliczeń
    #

    if scale == 'C':
        conv_to_Fahr = 9/5 * temp + 32
        conv_to_Kelv = temp + 273.15
        print('Aktualny pomiar temperatury {:06.2f} C, to:'.format(temp))
        print('{:06.2f} F, w skali Fahrenheita i ...'.format(conv_to_Fahr))
```

```
        print('{:06.2f} K, w skali Kelwina;'.format(conv_to_Kelv))
    elif scale == 'K':
        conv_to_Celc = 273.15 – temp
        conv_to_Fahr = 9/5 * conv_to_Celc + 32
        print('Aktualny pomiar temperatury {:06.2f} K, to:'.format(temp))
        print('{:06.2f} F, w skali Fahrenheita i ...'.format(conv_to_Fahr))
        print('{:06.2f} C, w skali Celcjusza'.format(conv_to_Celc))
    elif scale == 'F':
        conv_to_Celc = (temp - 32) * 5/9
        conv_to_Kelv = temp + conv_to_Celc
        print('Aktualny pomiar temperatury {:06.2f} F, to:'.format(temp))
        print('{:06.2f} K, w skali Kelwina i ...'.format(conv_to_Kelv))
        print('{:06.2f} C, w skali Celcjusza'.format(conv_to_Celc))
    else:
        print('\nWybrano niewłaściwą skalę temperatury.')
        print('Aktualny pomiar to: {:06.2f}\n'.format(temp))

    # blok decyzyjny
    # Niepisana zasada: zmienne powinny być utworzone najbliżej miejsca ich użycia;
    #

    nagative_options = ('Nie', 'nie', 'NIE', 'n', 'N')
    decision = input(
       '\nMasz nieodpartą chęć wykonania kolejnych obliczeń ???\n
       Tak: naciśnij dowolny klawisz by kontynuować, lub ...\n
       Nie: wpisz (N)ie by zakończyć.\nTwoja decyzja: ')
    if decision in nagative_options:
        break
    else:
        print('\n ... będę znowu działać !!!\n')

print('To naprawdę już koniec …')
```

**#20 Pytania kontrolne**

1. Scharakteryzuj typ logiczny i wskaż jego zastosowania w kodzie programu.
2. Jaka jest kolejność wyliczania wyrażeń? Co rozumiesz przez wyliczanie leniwe?
3. Do czego służy grupowanie instrukcji?
4. Jak wygląda zapis złożonej instrukcji ‘if’?
5. Jak wygląda zapis złożonej instrukcji ‘while’?
6. Jaka jest rola instrukcji ‘break’ i ‘continue’?
7. Wymień kilka dobrych praktyk zapisu kodu programu.
8. Do czego służą instrukcje ‘pass’ i ‘in’?



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

# 03. Kolekcje obiektów: listy,krotki,ciągi znaków. Iteracja for;

**Indeks zagadnień:** generalna klasyfikacja typów danych, kolekcje obiektów, typy danych reprezentujące kolekcje, listy, krotki, ciągi znaków, operator indeksowania, operacje na listach, generator range, listy składane, przetwarzanie kolekcji z wykorzystaniem pętli 'for', instrukcja 'enumerate';

#### #01 Typy proste niemodyfikowalne

Jak dotąd posługiwaliśmy się przede wszystkim skalarnymi typami danych, takich jak: float, int, bool (wspomniano o typach Complex, Decimal i Fracion). Wymienione typy danych służą do przechowywania w pamięci środowiska wartości numerycznych. Obiekty tworzone na podstawie tych typów są niemodyfikowalne (ang. immutable). Za każdym razem, gdy zmieniamy zawartość takiego obiektu w środowisku, tworzony jest nowy obiekt:

```
>>> a = 3.234, b = 2;
>>> id(a)                          # Wynik: 140303616411568
>>> a = a + b                      # lub inaczej a += b
>>> id(a)                          # Wynik: 140303615860112
```

#### #02 Czym są kolekcje/sekwencje obiektów?

Używając typów prostych, możemy formułować wyrażenia i wykonywać operacje na obiektach (np. logiczne i arytmetyczne) i zapisywać ich wynik do zmiennych, co przedstawiono we wcześniejszych skryptach. Na tej podstawie możliwe jest również wykonywanie sekwencji operacji opisanych w pętlach i instrukcjach warunkowych (while/else, if/elif/else).

W jaki sposób przedstawiać i przetwarzać sekwencje danych w programach zapisanych w kodzie języka Python? Oto kilka przykładów sekwencji danych składających się z elementów tego samego i różnych typów danych:

```
Ciąg wartości numerycznych:      123, 43.34, 23, 128.1, 453, 1024
Seria ciągów znaków:             'Ala', 'ma', 'kota'
Dowolny ciąg obiektów:           True, 23.45, 'System Linux', False, 234
```

Gdyby nadać nazwy tym sekwencjom (tzn. przyporządkować im zmienne) i ustalić standardowy sposób dostępu do ich elementów składowych, wówczas znacznie ułatwiłoby to zapis kodu odpowiedzialnego za przetwarzanie danych w pętlach, funkcjach,w tym porównywanie ze sobą obiektów danych:

```
>>> a = 12, 23, 34, 12.45          # Pierwsza sekwencja
>>> b = 12, 33.23, 13, 1234.45     # Druga sekwencja
>>> a == b                         # Wynik: False
```

Chcielibyśmy reprezentować sekwencję danych tylko za pomocą jednej zmiennej; Jak to zapisać w kodzie języka Python?



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

## #03 Sekwencje danych w kodzie języka Python

W języku Python zdefiniowano następujące typy wykorzystywane do reprezentowania sekwencji danych:

```
# nazwa       # komentarz         # przykład

list          # lista        a = [12, 32, 3.23, 12.3]
tuple         # krotka              b = (12, 32, 3.23, 12.3)
str           # ciągi znaków c = 'Ala ma kota'
dict          # słowniki            e = {'id': 10, 'imię': 'Jan', 'nazwisko':'Nowak'}
set           # zbiór        d = {12, 32, 3.23, 12.3}
```

## #04 Lista, czyli modyfikowalna i uporządkowana kolekcja (obiektów) danych

Listy to obiekty, które służą do sekwencyjnego gromadzenia obiektów w pamięci środowiska; tworzą uporządkowane (indeksowane) struktury danych. W kodzie języka Python Lista może być utworzona na parę sposobów:

```
>>> a = []                          # Z użyciem nawiasów kwadratowych; pusta lista;
>>> a = list()               # Użycie konstruktora typu list; pusta lista;
>>> b = [12, 23, 23]         # Lista obiektów numerycznych - typu int
>>> b = list((12, 23, 23))          # Jawne użycie konstruktora listy
>>> c = [12.0, 23, 23.23]           # Lista obiektów numerycznych – typy: int i float
>>> d = ['Ala', 'Bela']             # Lista obiektów zawieracjących ciągi znaków: typ str
>>> e = ['Ala', 23, 234.3]          # Lista obiektów typu: str, int, float
```

(!) Pamiętajmy, że pustą listę reprezentujemy literałem '[]'; Jest to przydatne w instrukcjach warunkowych, pętlach i wyrażeniach logicznych;

## #05 Dostęp do elementów listy

Utwórzmy dla przykładu listę następującą obiektów/danych:

```
>>> lista = [10, 20.01,  30,  'id']
   indeksy:  0     1     2     3
```

| #indeks | # elementy | # typ obiektu | # element | # adres pamięci |
|---|---|---|---|---|
| 0 | 10 | int | id(lista[0]) | 94923493168896 |
| 1 | 20.01 | float | id(lista[1]) | 140303614436272 |
| 2 | 30 | int | id(lista[2]) | 94923493168256 |
| 3 | 'id' | str | id(lista[3]) | 140303615860144 |

```
>>> id(lista), type(lista)
(140303613275264, class <'list'>)
```

Do obiektów ujętych w listę odwołujemy się przez zmienną nadaną przy jej utworzeniu oraz indeks/pozycję elementu listy: lista[i], gdzie i jest indeksem elementu liczonym od początku listy tj. zawsze od wartości 0:

```
>>> lista[0]                    # Pierwszy element listy
10
>>> n = len(list)               # Funkcja środowiska zwr. liczbę el. Listy
4                               # tu: lista zawiera cztery obiekty
>>> lista[n-1]              # n – 1 = 3 indeks ostatniego elementu listy
'id'
>>> lista[-1], list[-2]         # Ostatni oraz przedostatni element listy
('id', 30)
```

**#06 Operacje na listach**

Poniżej zestawiono typowe operacje wykonywane na listach wraz z komentarzem do kodu.

```
>>> lista.append(3.1415)  # Dodawanie elementu na koniec listy
>>> lista                     # [10, 20.01, 30, 'id', 3.1415]
>>> lista[3] = 2.7182         # Zastępowanie elementu spod danego indeksu '3'
>>> lista                     # [10, 20.01, 30, 2.7182, 3.1415]
>>> del lista[2]              # Usunięcie elementu listy spod indeksu '2'
>>> lista                     # [10, 20.01, 30, 2.7182, 3.1415]
>>> lista.insert(1, 13.12)# wstawienie nowego elem. 'new' pod indeks 1
>>> lista                     # [10, 13.12, 20.01, 2.7182, 3.1415]
>>> lista.sort()              # Uporządkowanie elem. w kolejnosci rosnącej
>>> lista                     # [2.7182, 3.1415, 10, 13.12, 20.01, 30]
>>> lista.sort(reverse=True)  # Uporządkowanie elem. w kolejności malejącej
>>> lista                     # [30, 20.01, 13.12, 10, 3.1415, 2.7182]
>>> lista.reverse()           # Odwrócenie kolejności elementów listy
>>> lista                     # [2.7182, 3.1415, 10, 13.12, 20.01, 30]
>>> lista.count(13.12)        # Zlicza liczbę elementów o wartości 13.12; tu: 1
1                             # jedno wystąpienie wartości 13.12
>>> lista.index(20.01)        # Zwraca indeks elementu 20.01;
4                             # indeks 4, czyli 5-ta pozycja el. 20.01 na liście
```

UWAGA: Te operacje zmieniają elementy listy i/lub ich kolejność występowania. Przy sortowaniu listy, elementy listy muszą być tego samego typu.

**#07 Problem powiązania list (ang. aliasing)**

Rozważmy zmienną „a" odwołującą się do listy zawierającej pewne elementy, a następnie przypiszmy referencje tej listy do nowej zmiennej „b":

```
>>> a = [1, 2, 3, 4]            # Utworzenie listy z elementami
>>> b = a                       # b jest teraz listą
>>> id(a), id(b), a is b        # Pokaż adresy list: a i b
(140576608955360, 140576608955360, True)    # b reprezentuje tę samą listę co a
```



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

```
>>> b.append(5); b.reverse()        # Operacje na liście
>>> b                               # [5, 4, 3, 2, 1]
```

Dwie listy są ze sobą powiązane, gdyż dwie zmienne wskazują na ten sam obiekt. Należy uważać na takie operacje, choć ta cecha list jest także i ich zaletą (kopiowane są referencje do list – adresy, a nie zawartość list – co mogłoby być czasochłonne oraz nieekonomiczne, jeżeli chodzi o niepotrzebnie zajęte zasoby pamięci.

**#08 Tworzenie niezależnej kopii listy**

W przypadku konieczności utworzenia niezależnych list zawierających te same elementy wykorzystujemy następującą konstrukcję:

```
>>> a = [1, 2, 3, 4]                # Utworzenie listy z elementami;
>>> b = a[:]                        # b jest teraz listą będącą kopią listy a;
>>> b                               # ':' oznacza wybierz wszystkie elemenenty Listy;
[1, 2, 3, 4]                        # Kopia głęboka wymaga funkcji copy.deepcopy();
```

**#09 Porównywanie elementów list**

Aby sprawdzić, czy dwie zmienne wskazują na tę samą referencję, używamy operatora 'is', co zostało przedstawione w poniższych przykładach. Z kolei, jeżeli chcemy ocenić, czy dwie listy zawierają te same elementy, używamy operatora '==';

```
>>> a = [1, 2, 3, 4]                # Utworzenie list a,...
>>> b = a                           # ... b oraz ...
>>> c = [5, 6, 7, 8]                # ... c wraz z elementami
>>> a == b, b == c, c == a          # Czy listy są równe  ?
(True, False, False)                #
>>> a is b, b is c, c is a          # Czy zmienne odnoszą się do tych samych list tzn. ...
(True, False, False)                # ... czy zmienne wskazują na tą samą referencję?
>>> b == []                         # Czy lista jest pusta ?
False                               #
```

**#10 Wybór elementów listy za pomocą operatora indeksowania '[]'**

W przypadku listy oraz innych typów sekwencji danych język Python oferuje bardzo użyteczną formę wyboru elementów:

```
>>> a = [0, 1, 2, 3, 4, 5, 6] # Utworzenie listy a;
>>> a[:]                            # Wybór wszystkich elementów listy;
[0, 1, 2, 3, 4, 5, 6]               #
>>> a[:-1]                          # Wybór wszystkich elementów listy prócz
[0, 1, 2, 3, 4, 5]                  # ostatniego;
>>> a[1:]                           # Wybór wszyst. elementów listy prócz pierwszego
[1, 2, 3, 4, 5, 6]                  # ... lub od drugiego;
>>> a[1:6]                          # Wybór wszyst. elementów od drugiego do piątego
>>> [1, 2, 3, 4, 5]                 #
>>> a[1:-1]                         # ... w typ przypadku to samo co powyżej
```

Podsumowując:



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

- -1 oznacza indeks ostatniego elementu (analogicznie -2 indeks przedostatniego elementu itp.)
- a[n:m] oznacza wybór od n-tego elementu listy, aż do m – 1 włącznie;
  gdy n = 0 (początek listy), wówczas jesteśmy zwolnieni ze wpisywania indeksu 0; np.: a[0:m] jest tożsame z a[:m];

**#11 Inne użyteczne funkcje i operacje na listach**

Poniższy kod ilustruje działanie użytecznych funkcji i operatorów wbudowanych w środowisko, które są często wykorzystywanie w kodzie programu.

```
>>> a = [1, 2, 3, 4]                  # Dowolne listy a ib
>>> b = [5, 6, 7, 8]                  #
>>> a + b + [9]                       # Połączenie list
[1, 2, 3, 4, 5, 6, 7, 8, 9]           #
>>> 2*a                               # a*k lub k*a, gdzie k to licz. naturalna
[1, 2, 3, 4, 1, 2, 3, 4]              # wielokrotna duplikacja elementów listy
>>> sum(a), min(a), max(a), len(4)    # suma, min, maks oraz rozmiar Listy
(10, 1, 4, 4)                         # odpowiednio: suma, min., max. i roz.listy
>>> 10 in a                           # operator przynależności
False
```

**#12 Pętla 'for' czyli iteracje po elementach sekwencji/kolekcji danych**

Z racji tego, że kolekcje danych (list, krotek, słowników, ciągów znaków czy zbiorów) zawierają skończoną (znaną) liczbę elementów, do ich przetwarzania najczęściej wykorzystuje się pętlę określoną for. Konstrukcja tej instrukcji jest następująca:

```
for element in kolekcja:                    # Bieżący/kolejny element z kolekcji
    -------                                 # Sekwencja instrukcji, w której wykorzystuje się
    -------                                 # bieżący element kolekcji
```

Pętla **for** pozwala na przegląd wszystkich elementów kolekcji jeden po drugim. W pierwszym kroku iteracji pobierany jest pierwszy element kolekcji (o indeksie 0), który jest reprezentowany przez zmienną element, w drugim - drugi itp. Pętla kończy działanie z chwilą osiągnięcia ostatniego elementu lub gdy użyta zostanie instrukcja **break**. Przykład zastosowania pętli for:

```
lista   = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]   # Kolekcja/lista liczb;
                                            #
for liczba in lista:                        # liczba to tzw. zmienna pętli
    if liczba % 2 == 0:                     # Oblicza i wyświetla kw. liczb
        print(liczba, '**2=', liczba**2)    # ... o ile liczba jest parzysta
    if liczba % 3 == 0:                     # Oblicza i wyświetla sz. Liczb
        print(liczba, '**3=', liczba**3)    # ... o ile liczba jest dziel. 3
    if liczba % 9 == 0:                     # Opuszczam pętlę gdy liczba jest
       break                                # dzielnikiem 9; użycie break
```

**#13 Rozwiązanie tego samego problemu, ale z wykorzystaniem pętli while**

Poniższy kod zwraca te same wyniki. Jakie są różnice w obydwu wersjach programu? Czy pętla **for** użyta do przetwarzania sekwencji ma jakąkolwiek przewagę, niż pętla **while**? Spróbujmy odpowiedzieć na to pytanie porównując konstrukcje obydwu iteracji. Przykład przetwarzania sekwencji z wykorzystaniem pętli **while**:

```
lista   = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
n = len(lista)                               # Ustalamy długość listy
i = 0                                        # Wprowadzono licznik pętli
while i < n:                                 # Odwołujemy się do listy
    if lista[i] % 2 == 0 :                   # W takich operacjach zawartość
    print(lista[i], '**2=', lista[i]**2)     # listy może być zmieniona, co
    if lista[i] % 3 == 0:                    # nie musi być zamierzone
    print(lista[i], '**3=', lista[i]**3)
    if lista[i] % 9 == 0:
        break                                # Wyjście z petli
    i += 1                                   # (!!!)
                                             # Konieczna jest obsługa licznika pętli
```

**#14 Wbudowana funkcja 'range()'**

Zauważmy, że tworzenie długich list z wykorzystaniem literałów jest bardzo niewygodne. Zazwyczaj listy będą zawierać dane odczytane z plików, komunikacji sieciowej lub generowane za pomocą tzw. list składanych (o czym dalej). Niemniej, w niektórych przypadkach zachodzi potrzeba elastycznego generowania listy elementów bezpośrednio w pętli for. Do generowania liczb porządkowych, które mogą być wykorzystywane jako indeksy kolekcji lub argumenty wyrażeń obliczanych w pętli służy funkcja środowiska range().

```
                                 # start – pierwsza liczba ciągu
range(start, stop, step)         # stop – ostatnia liczba ciągu (z jej wyłączeniem)
                                 # step – krok zmienności (domyślnie 1)

>>> list(range(1, 100, 2))           # lista 50 nieparzystych liczb od 1 do 99
>>> list(range(-100, -1, 2))  # [-100, -98, -96, ..., -2]
>>> list(range(100, 0, -2))   # [ 100,  98,  96, ...,  2]
```

Przykład: Utworzenie wartości funkcji sin(x) w przedziale (0,2π) z dokładnością do 2*pi/n rad.

```
import math

n       =     1000               # liczba wartości argumentów
step    =     2*math.pi/n        # krok zmienności argumentu
values  =     list()             # równoważnie: values = []
arg     =     0                  # pierwsza wartość argumentu

for x in range(0, n):            # iteracja od 0 do 999 elementu
    value = math.sin(arg)        # krócej: values.append(math.sin(arg))
    values.append(value)         # Dodanie wyniku oszacowania do listy
    arg += step                  # przejście do kolejnej wartości argumentu
```

```
print('Wartości funkcji sinus z przedziału (0, 2π):\n', values)
```

**#15 Listy składane (ang. comprehension lists)**

Język Python posiada bardzo użyteczną konstrukcję ułatwiającą generowanie i przetwarzanie złożonych sekwencji danych nazywaną mianem list składanych. Jej zapis przedstawia się następująco:

```
[wyrażenie/element for element in kolekcja if warunek else wyrażenie]
[wyrażenie/element for element in kolekcja]
```

Przykłady:

```
>>> liczby = [-1, 1, -2, 2, -3, 3, -4, 4]
>>> [x for x in liczby if x > 0]          # Wydobycie z kolekcji liczb dodatnich
>>> [x**2 for x in range(1, 100, 2)]
>>> [znak for znak in 'Ala ma kota']       # Rozbicie ciągu znaków na pojedyncze znaki
>>> samogloski = ['a', 'e', 'i', 'o', 'u'];
>>> ciag = 'Ala ma kota'
>>> [s for s in ciag if s in samogloski]   # Tylko samogłoski w ciagu
['a', 'a', 'o', 'a']
```

W każdym z przykładów jako wynik zwracana jest kolekcja elementów (tu: lista).

**#16 Stałe sekwencje danych: tuple (inaczej: krotki)**

Krotki to niemodyfikowane sekwencje/kolekcje danych dowolnych typów. Tworzenie i zastosowania krotek jest bardzo podobne do list, niemniej należy pamiętać, że po ich utworzeniu,ani ich rozmiar, ani zawartość przechowywanych w nich obiektów nie może ulec zmianie!

```
>>> t = (0, 1, 100, ‘kota’)               # Utworzenie krotki – I sposób;
>>> t = tuple((0, 1, 100,‘kota’))         # Utworzenie krotki – II sposób;
>>> t = ()                                # UWAGA na zapis: ()        - to jest pusta krotka;
>>> t = (123,)                            # UWAGA na zapis: (element,) - to jest krotka
                                          # jednoelementowa (!!!)
>>> t = (123, (1, 100, ‘kota’))           # Krotka zagnieżdżona;
>>> t[i]                                  # Odczyt elementu krotki spod indeksu i
>>> t[1]                                  # Odczyt drugiego elementu krotki - ...
(0, 1, 100, ‘kota’)                       # ... to ta sama konwencja co przy listach;
```

W przypadku krotek nie występuje efekt powiązania (patrz: #7 – listy) – krotki nie mogą być modyfikowalne (ang. unmutable).

**#17 Operacje na krotkach**

Działania na krotkach są podobne do tych, wykonywanych na listach (patrz: # 8 – 11) oprócz operacji, które mogłyby zmodyfikować zawartość krotki lub liczbę jej elementów. Jeżeli niezbędne jest jej uporządkowanie lub inne działania modyfikujące krotkę, należy utworzyć jej kopię i przekształcić ją w listę.

```
>>> krotka = (0, 1, 100, 18, 232.23)
>>> lista = list(krotka)
>>> lista.sort()                            # Sotrowanie listy [0, 1, 18, 100, 232.23];
>>> krotka = tuple(lista)                   # Powrót do krotki;
>>> krotka = krotka + (1,)                  # Dodawanie elementu i utworzenie nowej krotki;
>>> a = 1; b = 2                            # Sprytna zamiana wartości między zmiennymi;
>>> b, a = a, b                             # Po lewej i prawej stronie są krotki
>>> (b, a) = (a, b)                         # Tożsame z poprzednim zapisem: b, a = a, b
```

**#18 Po co właściwie są krotki skoro pełnią podobną funkcję do list?**

Niemodyfikowalność krotek zapewnia ich zabezpieczenie przed zmianami ze strony innych programistów, co nie jest możliwe do osiągnięcia przez zastosowanie list. Krotki należy traktować jako rodzaj stałych.

Jak przekonamy się później, krotki reprezentują **argumenty wywołania funkcji** oraz mają zastosowanie przy **agregacji i rekonstrukcji** sekwencji danych (zip, operator '*' i '**'). Krotki są argumentami wielu funkcji ze standardowych bibliotek języka Python oraz są często wykorzystywane w tzw. **wyrażeniach formatujących**, o których będzie mowa przy okazji wykładu dotyczącego przetwarzania tekstu. Dodatkowo, krotki są często zwracane przez funkcje, co jest traktowane jako jednoczesne zwracanie wielu wartości.

**#19 Przykład działania krotek - wbudowana funkcja 'enumerate()'**

Funkcja enumerate zwraca tzw. krotkę zawierającą numer porządkowy pobranego elementu listy oraz kolejny element.

```
>>> l = list(range(10))                     # tworzę listę liczb od 0 do 9
>>> list(enumerate(l))                      # zwraca listę krotek zawierających
                                            # uporządkowane wartości 0 do 9
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8), (9, 9)]
```

To może mieć zastosowanie w pętli for przy przetwarzaniu sekwencji danych. Jak pamiętamy, pętla **for** nie udostępnia w sposób jawny bieżącej wartości licznika pętli, tak jak to widzieliśmy w przypadku pętli **while**. Funkcja (generator) **emumatrate** zwraca bieżącą wartość licznika, co pozwala na wykonywanie  indeksacji kolekcji lub innych operacji w pętli. Przykład: modyfikacja programu do obliczeń kwadratów i sześcianów liczb.

```
lista   = list(range(10, 20))       # zakres od 10 do 19
                                            #
for (i, liczba) in enumerate(lista):        # w każdej iteracji tworzę krotkę (i, liczba)
    if liczba % 2 == 0:
        print(i, liczba, '**2=', liczba**2)
    if liczba % 3 == 0:
        print(i, liczba, '**3=', liczba**3)
    if liczba % 9 == 0:
        break
```

**#21 Sekwencje ciągów znaków**

Ciągi znaków są reprezentowane przez typ str, który ma bardzo podobne zastosowania, właściwości i operacje jak typ tuple (krotki: patrz # 18 – 19). Ciągi znaków są niemodyfikowalne. Wszystkie operacje na ciągach znaków wymagają tworzenia obiektów pośrednich (tak jak w przypadku krotek).

```
>>> s = ''                              # posty ciąg znaków
>>> s = 'Ala ma kota'                   # Utworzenie ciągu znaków; Równie dobrze
>>> s = str('Ala ma kota')              # można użyć podwójnego cudzysłowu ””
>>> s[1:6]                              # Zwraca wybrane elementy; patrz: #10)
'la ma'
>>> 'Ala' + ‘ma’ + '...'                # Konkatacja: ‘+’ łączy ciągi znaków
'Alama...'
>>> s.upper()                           # Zamiana dużych znaków na małe
'ALA MA KOTA'                           # ... , ale zawartość s nie ulega zmianie
>>> s == 'Bela ma psa'
False
```

Więcej przykładów - patrz: help(str). Typ str ma bardzo wiele użytecznych funkcji do formatowania, poszukiwania, zamiany znaków, ustalenia kodowania znaków, podziału ciągów znaków i wiele wiele innych, które będą przedmiotem oddzielnego skryptu. Ciągi znaków przetwarza się tak samo, jak i inne sekwencje/kolekcje danych. Przykład: przetwarzanie ciągu znaków:

```
>>> ciag = 'Ala ma kota'
>>> for i, znak in enumerate(ciag):         # Jednoczesna iteracja po indeksie i znakach
        print(i,'\t', znak.upper(), end='') # Wyświetlenie w oddzielnym wierszu ind. i znaku
                                            # lub zapamiętanie wyniku w liście:
>>> ciag_duzych_liter = [znak.upper() for znak in ciag]
```

**#22 Jednoczesne przetwarzanie wielu różnych sekwencji**

Niejednokrotnie, w pętli **for** przetwarzamy jednocześnie kilka sekwencji znaków (mogących się różnić długością kolekcji). W takim przypadku użyteczna jest funkcja **zip**, która łączy elementy poszczególnych sekwencji w krotki, a następnie udostępnia te krotki z każdej z iteracji pętli. Przykład: użycie wielu sekwencji w jednej pętli **for** (to zadanie jest uciążliwe do wykonania w przypadku stosowania pętli **while**).

```
lista_01   = list(range(10, 21))            # [10, 11, 12, ..., 20]
lista_02   = list(range(21, 31))            # [21, 22, 23, ..., 30]
ciag       = 'Ala ma kota'                  # ‘A’,’l’,’a’, ...,’a’

for i, (l1, l2, znak) in enumerate(zip(lista_01, lista_02, ciag)):
    print('[', i, ']', znak, l1, l2, l1*l2, l1/l1)
```

W każdej iteracji jest tworzona/zwracana nowa krotka zawierające bieżące wartości przetwarzanych sekwencji. **Proszę zapamiętać ten przykład**, gdyż jest bardzo użyteczny. W przypadku różnej długości agregowanych/składanych sekwencji pętla będzie wykonywana tyle razy,



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

ile wynosi długość najkrótszej sekwencji; Proszę o wykonanie tego kodu dla sekwencji: ciąg = 'Ala', bez zmiany elementów list lista_01 i lista_02.

**#23 Przykłady kodu języka Python wykorzystujące przedstawione konstrukcje**

Zapiszmy program z paragrafu #14 w bardziej zwarty sposób, tym razem, korzystając z list składanych:

```
# Wcześniejsza wersja kodu. Tak pisze się programy w tradycyjnych językach programowania;
import math                         # import biblioteki
n       =     1000                  # liczba wartości wyliczanych argumentów
step    =     2*math.pi/n           # krok zmienności argumentu
values  =     list()                # równoważnie: values = []
arg     =     0                     # pierwsza wartość argumentu

for x in range(0, n):        # iteracja od 0 do 999 elementu
    value = math.sin(arg)           # krócej: values.append(math.sin(arg))
    values.append(value)            # Dodanie wyniku oszacowania do listy
    arg += step                     # przejście do kolejnej wartości argumentu

print('Wartości funkcji sinus z przedziału (0, 2π):\n', values)

# Zwarta wersja kodu zapisana z wykorzystaniem list składanych;
import math
values = [math.sin(x) for x in (x*2*math.pi/1000 for x in range(0, 1000))]
print('Wartości funkcji sinus z przedziału (0, 2π):\n', values)
```

Dyskusja:

```
values = [                          # Obliczenia są wykonywane w dwóch krokach
           math.sin(arg)            # 2. Utworzenie listy wartości na podstawie wartości
           for arg in               #    argumentów;
           (x*2*math.pi/1000 # 1. Utworzenie krotki zawierającej wartości argumentów;
           for x in                 #
           range(0, 1000))          #
          ]
```

**#24 Pytania kontrolne:**

1. Jak reprezentowane są kolekcje danych w języku Python. Wymień przynajmniej trzy klasy wykorzystywane do przetwarzania sekwencji?
2. Jaka jest zasadnicza różnica między krotką a listą?
3. Co rozumiesz przez pojęcie lista składana?
4. Jakie są zalety stosowania list składanych?
5. Scharakteryzuj konstrukcję pętli ‘for’.
6. Jaka jest rola instrukcji ‘enumerate’ w zapisie pętli ‘for’.



PROGRAM WIEDZA EDUKACJA ROZWÓJ
Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny