PODSTAWY PROGRAMOWANIA W JĘZYKU PYTHON

DR KATARZYNA SZULC

WIT Wyższa Szkoła Informatyki Stosowanej i Zarządzania
pod auspicjami Polskiej Akademii Nauk

ITZ Informatyczne Techniki Zarządzania

Rok akademicki 2023/2024

Spis treści

STUDIA ZAOCZNE:

Wykład :	Sobota	gr. 1+2+3	godz. 11:45–13:15	Teams
Laboratorium :	Niedziela	gr. 2	godz. 13:30–15:00	N412
		gr. 3	godz. 15:15–16:45	N410
Lab. 1, 3, 5, 7		gr. 1	godz. 17:00–18:30	N410
Lab. 2, 4, 6, 8			godz. 11:45–13:15	N412

STUDIA DZIENNE:

Wykład :	Poniedziałek	gr. 1+2	godz. 12:15–13:45	Teams
Laboratorium :	Wtorek	gr. 1	godz. 10:30–12:00	N410
		gr. 2	godz. 16:15–17:45	N410

REGULAMIN ZAJĘĆ

1. Obecność:
   - Obecność na wykładzie jest zalecana.
   - Obecność na laboratoriach jest obowiązkowa.
2. W trakcie semestru przewidziane są prace klasowe/domowe jako zaliczenie laboratoriów i jeden egzamin końcowy (w sesji). Aby móc przystąpić do egzaminu należy zaliczyć laboratoria.
3. Ocena końcowa:
   - Na ocenę końcową z laboratoriów składają się ocena z prac klasowych/domowych oraz obecność na zajęciach.
   - Na ocenę końcową z całego przedmiotu składają się ocena końcowa z laboratoriów oraz ocena z egzaminu.
4. Punktacja:
   - Maksimum punktacji wyznacza student, który zdobył najlepszy wynik z laboratorium/egzaminu.
   - Aby zaliczyć laboratorium/egzamin należy zdobyć minimum 50% maksymalnego wyniku.
5. W trakcie semestru (głównie na laboratoriach) punkty przyznawane są w następujący sposób:
   - Prace Klasowe/Domowe - około 50 punktów,
   - Obecność na laboratoriach - około 8 punktów.

   W przypadku zaliczenia laboratoriów na około 80% maksymalnej ilości punktów, student jest zwolniony z egzaminu.
6. Przewidziane są terminy poprawkowe dla studentów, którym nie udało się zaliczyć przedmiotu w pierwszym terminie - laboratoria i egzamin.
7. Materiały do zajęć znajdują się w systemie UBI.
8. Szczegółowy wykaz zagadnień: Tabela 1.
9. Szczegółowy kalendarz zajęć: Tabela 2.

Studia Zaoczne:

Data	Nr zajęć	Tematyka	Sala
7-8/10/2023	W+Lab 1	Wprowadzenie do programowania	Teams+Prac.Komp.
21-22/10/2023	W+Lab 2	Instrukcje warunkowe	Teams+Prac.Komp.
4-5/11/2023	W+Lab 3	Pętle	Teams+Prac.Komp.
25-26/11/2023	W+Lab 4	i funkcje	Teams+Prac.Komp.
9-10/12/2023	W+Lab 5	Listy, algorytmy sortowania	Teams+Prac.Komp.
13-14/01/2024	W+Lab 6	Kontenery danych	Teams+Prac.Komp.
27-28/01/2024	W+Lab 7	Rekurencja	Teams+Prac.Komp.
17-18/02/2024	W+Lab 8	Wstęp do obiektowości	Teams+Prac.Komp.
24/02/2024	E	Egzamin	
2/03/2024	E	Egzamin poprawkowy	

Studia Dzienne:

Data	Nr zajęć	Tematyka	Sala
16-17/10/2023	W+Lab 1	Wprowadzenie do programowania	Teams+Prac.Komp.
23-24/10/2023	W+Lab 2	Środowisko języka PYTHON	Teams+Prac.Komp.
30-31/11/2023	W+Lab 3	Łaćuchy znakowe i obiekty	Teams+Prac.Komp.
6-7/11/2023	W+Lab 4	Pętle i funkcje	Teams+Prac.Komp.
13-14/11/2023	W+Lab 5	Listy, algorytmy sortowania	Teams+Prac.Komp.
20-21/11/2023	W+Lab 6	Kontenery danych	Teams+Prac.Komp.
27-28/11/2023	W+Lab 7	Rekurencja	Teams+Prac.Komp.
4-5/12/2023	W+Lab 8	Wstęp do obiektowości	Teams+Prac.Komp.
11-12/12/2023	W+Lab 9		Teams+Prac.Komp.
8-9/01/2024	W+Lab 10		Teams+Prac.Komp.
15-16/01/2024	W+Lab 11		Teams+Prac.Komp.
22-23/01/2024	W+Lab 12		Teams+Prac.Komp.
29-30/01/2024	W+Lab 13		Teams+Prac.Komp.

Lp.	**Tematyka**	**Liczba godzin**
1.	Organizacja zajęć.	W+Lab=2+2
2.		W+Lab=2+2
3.		W+Lab=2+2
4.		W+Lab=2+2
5.		W+Lab=2+2
6.		W+Lab=2+2
7.		W+Lab=2+2
8.		W+Lab=2+2

1. Wprowadzenie do programowania z języku PYTHON

1.1. **Wstęp do programowania.** Programowanie polega na projektowaniu, tworzeniu, testowaniu i utrzymywaniu kodu źródłowego programów komputerowych oraz urządzeń wyposażonych w mikrokontrolery.

Programista wykorzystuje wiedzę z wielu dziedzin z których najważniejsze to struktury danych, algorytmika oraz wiedza o kompilatorach.

Przykład 1.1. Najprostszym przykładem programu (nie koniecznie komputerowego) jest przepis na jejecznicę...

- Dane: ogień, patelnia, widelec, jajka, czas...
- Przetwarzanie: wbij jajka na patelnię, postaw patelnie na ogniu, zamieszaj widelcem, smaż 2 minuty;
- Wynik: jajecznica.

Najważniejszym elementem programowania jest znajomość języków programowania i umiejętność ich praktycznego zastosowania.

Języki programowania są narzędziem, z których pomocą uzyskuje się zamierzony efekt, czyli wprowadza działanie wymyślonego algorytmu w życie. Podczas wyboru języka programowania najważniejsze jest strategiczne myślenie i jasno określony cel działania programu.

1.2. **Język Python. Python** jest językiem programowania ogólnego przeznaczenia typu *open source* zoptymalizowany pod kątem jakości, wydajności, przenośności i integracji. Jest on obecnie używany przez miliony programistów na całym świecie.

Python jest językiem najczęściej stosowanym do implementacji algorytmów sztucznej inteligencji, inteligencji obliczeniowej i uczenia maszynowego.

Python jest niezwykle zgrabnym językiem zorientowanym obiektowo (OOP – Open Oriented Programming), posiadającym czytelną składnię, łatwy w utrzymaniu i integracji z komponentami języka C, posiadający bogaty zbiór interfejsów, bibliotek i narzędzi programistycznych.

Python jest językiem interpretowanym, co daje większą łatwość modyfikacji gotowego programu, lecz obniża efektywność działania w stosunku do języków kompilowanych, takich jak C, jak również utrudnia wykrywanie błędów.

Program źródłowy napisany w języku **Python** (podobnie jak w Javie) może być najpierw kompilowany do postaci pośredniej (byte–code), która następnie wykonywana jest przez Wirtualną Maszynę Pythona (PVM) na konkretnej platformie obliczeniowej.

Twórcą języka Python jest Guido van Rossum.

1.3. **Środowiska deweloperskie.**

- **PyCharm** jest profesjonalnym środowiskiem developerskim (IDE) z wbudowanym trybem naukowym pozwalającym analizować dane.
- **Spyder** jest środowiskiem developerskim z bogatym pakietem bibliotek, posiadającym wersję dla Windows, Linux i macOS.
- **Jupyter Notebook** jest środowiskiem developerskim w formie notatnika pozwalającym nie tylko edytować kod, lecz również wyświetlać notatki w formie tekstowej i graficznej, jak również wyświetlać wyniki obliczeń.
- **Google Colab**

Zadanie 1.1. Napisać pełny algorytm smażenia jajecznicy.

1.4. **Zmienne.** Zmienna jest to obiekt w którym przetrzymywana jest wartość lub dana. Każda wartość w Pythonie ma swój typ, np. lista, krotka, łańcuch, słownik, itd. Python nie wymaga komendy deklarującej daną.

```
x=3 #liczba całkowita, int
y=3.8 #liczba dziesiętna, float
z="Student" #łańcuch znaków, str (można definiować w apostrofach lub cudzysłowach )
```

Możliwe jest nadawanie typu zmiennej, np.:

```
x=str(3) # x będzie łańcuchem '3'
y=int(3.8) # y będzie liczbą całkowitą 3
z=float(3) # z będzie liczbą dziesiętną 3.0
```

Uwaga 1.1. Symbol # oznacza rozpoczęcie komentarza w wierszu. Wszystko co pojawi się po tym znaku jest ignorowane przez interpreter Pythona. Końcem komentarza jest koniec wiersza.

1.5. **Nazwy zmiennych.** Zmienną można nazwać używając liter alfabetu, np. **a** lub **b**, a także całych wyrazów, np. **imie**, **moja_zmienna**. Należy jednak przestrzegać pewnych zasad nazywania zmiennych w Pythonie:

- nazwa zmiennej musi zaczynać się literą lub ewentualnie podkreśleniem "_"
- nazwa zmiennej nie może zaczynać się cyfrą
- nazwa zmiennej może zawierać tylko znaki alfa-numeryczne lub podkreślenie, czyli takie jak 0-9, a-z, _
- nazwy zmiennych są wrażliwe na wielkość liter, tzn. **moja_zmienna**, **Moja_zmienna**, **MOJA_ZMIENNA** są trzema różnymi zmiennymi.

```
mojazmienna= 1.0
moja_zmienna= 2
_moja_zmienna='3'
MojaZmienna=4.0
Moja_Zmienna=0.5
MOJA_ZMIENNA=6
mojazmienna2=7
MOJAZMIENNA=8
```

Można przypisać wartości do większej ilości zmiennych w tej samej linii:

```
a, b, c="Ananas", "Banan", "Cytryna"
```

Uwaga 1.2. Zmienne nie mogą posiadać **nazw zastrzeżonych** dla Pythona, takich jak:
and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, yield.

1.6. **Operatory.** Służą do wykonywania operacji matematycznych, logicznych i symbolicznych.

Dodawanie: $+$. Dodawanie do dotychczasowej wartości: $a+=2$ ($a=a+2$).
Odejmowanie: $-$. Odejmowanie od dotychczasowej wartości: $a-=2$ ($a=a-2$).
Mnożenie: $*$. Mnożenie przez dotychczasową wartość: $a*=2$ ($a=a*2$).
Druga potęga: $**$
Dzielenie: $/$. Dzielenie dotychczasowej wartości $a/=2$ ($a=a/2$).

Dzielenie całkowite: //. Dzielenie, po którym część ułamkowa jest obcinana: $9//2 = 4$.
Równość ==. Nierówność | =. Znaki nierówności liczbowej: $>, <, >=, <=$.
Dodawanie zbiorów: +.
Suma zbiorów: |.
Iloczyn zbiorów: &
Odejmowanie zbiorów: −.
Negacja: not.
Alternatywa: or.
Koniunkcja: and.

Kombinacja wartości, operatorów i zmiennych, ale także funkcji stanowią tzw. wyrażenia.

```
x=2
y=3
z=x+y
```

Przykład 1.2. Napisać program w języku Python wyświetlający zdefiniowane dane używając funkcji *print()*.

PROGRAM:

```
x=5
y="Student"
k, l, m="Klucz", "Litera", "Muzyka"
print(x)
print(y)
print(k)
print(l)
print(m)
```

WYNIK:

```
5
Student
Klucz
Litera
Muzyka
```

1.7. Typy danych w języku Python.

1.7.1. *Wbudowane typy danych.* Klasyfikacja danych odbywa się poprzez ich typy, które określają jakie operacje mogą być wykonywane na danej klasie danych. Wyróżniamy następujące wbudowane typy danych w jezyku Python:

tekstowy: **str**
numeryczny: **int**, **float**, **complex**
sekwencyjny: **list**, **tuple** (krotka), **range**
słownikowy: **dict**
zbiorów: **set**, **frozenset**
logiczny: **bool**
binarny: **bytes**, **bytearray**, **memoryview**

Typ danych jest określany w momencie przypisania wartości do zmiennej.

Example	Typ danej
x='Hello world'	str
x=20	int
x=20.5	float
x=1j	complex
x=['ananas', 'banan', 'cytryna']	list
x=('ananas', 'banan', 'cytryna')	tuple
x=range(6)	range
x='imie' : 'Jan', 'wiek' : 26	dict
x='ananas', 'banan', 'cytryna'	set
x=frozenset('ananas', 'banan', 'cytryna')	frozenset
x=True	bool
x=b'Hello'	bytes
x=bytearray(5)	bytearray
x=memoryview(bytes(5))	memoryview

Przykład 1.3. Napisać program w języku Python wyświetlający typy zdefiniowanych danych używając funkcji *type()*.

PROGRAM:

```
x=5
y="Student"
owoce=["Ananas", "Banan", "Cytryna"]
print(type(x))
print(type(y))
print(type(owoce))
```

WYNIK:

```
<type 'int'>
<type 'str'>
<type 'list'>
```

1.8. **Polecenia wejścia–wyjścia (*input–output*).** Proces wprowadzania danych do komputera nazywany jest **wejściem** a funkcja, która realizuje ten proces to funkcja **input()**. Proces wyprowadania wyniku przez komputer nazywany jest **wyjściem** a funkcja, która realizuje ten proces to funkcja **print()**.

Przykład 1.4. Napisać program w języku Python wczytujący nazwę użytkownika oraz wypisujący tę nazwę na ekranie.

PROGRAM:

```
username = input('Wprowadz swoje imie: ')
print('Twoje imie to: ' + username)
```

WYNIK:

```
Wprowadz swoje imie: Kasia
Twoje imie to: Kasia
```

Przykład 1.5. Program wyświetlający łańcuch znaków.

PROGRAM:

```
print('Witaj w swiecie jezyka Python!' )
```

WYNIK:

```
Witaj w swiecie jezyka Python!
```

Przykład 1.6. Napisać program wyświetlający wyniki dodawania, odejmowania, mnożenia i dzielenia dwóch zadanych liczb całkowitych.

PROGRAM:

```
x=8
y=2
suma=x+y
roznica=x-y
iloczyn=x*y
iloraz=x/y
print(suma)
print(roznica)
print(iloczyn)
print(iloraz)
```

WYNIK:

```
10
6
16
4
```

1.9. **Funkcje matematyczne.** Python zawiera zbiór wbudowanych funkcji matematycznych takich jak: **max(x,y)** oraz **min(x,y)** zwracających wartości odpowiednio największą i najmniejszą z liczb x i y, **abs(x)** zwracają wartość bezwzględną, czyli moduł liczby x, oraz **pow(x,y)** zwracającą wartość x do potęgi y.

Przykład 1.7. Program znajdujący wartość najmniejszą i największą wśród zadanych liczb.

PROGRAM:

```
x=min(5,10,15)
y=max(5,10,15)
print(x)
print(y)
```

WYNIK:

```
5
15
```

Przykład 1.8. Program dający wartość bezwzględną zadanej liczby

PROGRAM:

```
x=abs(-7.35)
print(x)
```

WYNIK:

```
7.35
```

Przykład 1.9. Program liczący wartość x podniesione do potęgi y.
PROGRAM:

```
x=pow(4,3)
print(x)
```

WYNIK:

```
64
```

Moduł **math** zawiera definicje najczęściej używanych funkcji matematycznych. Aby użyć modułu **math** należy go zaimportować komendą `import math`. Funkcje matematyczne wywołuje polecenie: math.*funkcja*()

Funkcja	Opis działania
ceil(x)	zwraca sufit z liczby rzeczywistej x, tj. najmniejszą nie mniejszą liczbę całkowitą
floor(x)	zwraca podłogę z liczby rzeczywistej x, tj. największą nie większą liczbę całkowitą
fabs(x)	zwraca wartość bezwzględną z liczby rzeczywistej x
modf(x)	zwraca krotkę zawierającą część całkowitą i część ułamkową liczby x
hypot(x,y)	zwraca odległość punktu o współrzędnych (x,y) od początku układu współrzędnych (0,0)
exp(x)	zwraca e do potęgi x
log(x)	zwraca logarytm naturalny z x
log(x,p)	zwraca logarytm przy podstawie p z liczby x
pow(x,p)	zwraca p-tą potęgę liczby x
sqrt(x)	zwraca pierwiastek z x
sin(x)	zwraca sinus z x
asin(x)	zwraca arcus sinus z x
cos(x)	zwraca cosinus z x
acos(x)	zwraca arcus cosinus z x
tan(x)	zwraca tangens z x
atan(x)	zwraca arcus tangens z x
degrees(x)	zwraca w stopniach miarę konta x wyrażoną w radianach
radians(x)	zwraca w radianach miarę konta x wyrażoną w stopniach

Przykład 1.10. Program liczący pierwiastek kwadratowy z liczby 64.
PROGRAM:

```
import math
x=math.sqrt(64)
print(x)
```

WYNIK:

```
8.0
```

Przykład 1.11. Program zwracający górne i dolne zaokrąglenie liczby rzeczywistej 1.4.
PROGRAM:

```
import math
x=math.ceil(1.4)
y=math.floor(1.4)
print(x)
print(y)
```

Wynik:

```
2
1
```

Przykład 1.12. Program wyświetlający liczbę π.

Program:

```
import math
x=math.pi
print(x)
```

Wynik:

```
3.14159265359
```

1.10. **Zadania.**

Zadanie 1.2. Napisać prosty kalkulator w języku Python.

2. Instrukcje warunkowe i pętle w języku Python

2.1. **Operatory logiczne.** W języku Python określone są następujące operatory logiczne.

Operator	Znaczenie
a==b	a jest równe b
a! =b	a nie jest równe b
a<b	a jest mniejsze od b
a<=b	a jest mniejsze lub równe b
a>b	a jest większe b
a>=b	a jest większe lub równe b

Mają one zastosowanie zarówno w instrukcji warunkowej jak również w pętlach.

2.2. **Instrukcja warunkowa.** Instrukcja warunkowa ma następującą składnię

```
if warunek1:          # Jeśli jest spełniony wykonaj blok instrukcji
    instrukcja1
    instrukcja2
    ...
elif warunek2:        # Jeśli nie to zbadaj kolejny warunek
    instrukcja3
    instrukcja4
    ...
...
else:                 # Jeśli nie jest spełniony wykonaj blok instrukcji
    instrukcja5
    instrukcja6
    ...
instrukcja7           # Instrukcja następna po instrukcji warunkowej
```

Słowo kluczowe **if** oznacza, że jeśli spełniony jest *warunek1* wówczas program wykona *instrukcja1*, *instrukcja2* , ...
Słowo kluczowe **elif** oznacza, że jeśli *warunek1* nie jest spełniony to wówczas należy spradzić *warunek2* i w przypadku gdy jest on spełniony program ma wykonać *instrukcja3*, *instrukcja4* , ...
Słowo kluczowe **else** oznacza, że jeśli wcześniejsze warunki nie były spełnione to program ma wykonać *instrukcja5*, *instrukcja6* , ...
Części **elif** i **else** są opcjonalne. Każda część instrukcji warunkowej kończy się dwukropkiem.

Przykład 2.1. Program wyznaczający największą z zadanych dwóch liczb.
Program:

```
a=45
b=209
if b>a:
    print("b jest wieksze od a")
```

Wynik:

```
b jest wieksze od a
```

Przykład 2.2. Program wyznaczający większą lub równą z zadanych dwóch liczb.
Program:

```
a=33
b=33
if b>a:
    print("b jest wieksze od a")
elif a==b:
    print("a i b sa rowne")
```

WYNIK:

```
a i b sa rowne
```

Przykład 2.3. Program wyznaczający większą lub równą z zadanych dwóch liczb.
PROGRAM:

```
a=200
b=33
if b>a:
    print("b jest wieksze od a")
elif a==b:
    print("a i b sa rowne")
else:
    print("a jest wieksze od b")
```

WYNIK:

```
a jest wieksze od b
```

Rozważmy teraz przykład w których nie ma **elif**:

Przykład 2.4. Program wyznaczający większą z zadanych dwóch liczb.
PROGRAM:

```
a=200
b=33
if b>a:
    print("b jest wieksze od a")
else:
    print("b nie jest wieksze od a")
```

WYNIK:

```
b nie jest wieksze od a
```

Wersje skrócone dla **if** oraz **if ... else**. Jeżeli po *warunku* jest tylko jedna *instrukcja* do wykonania, wówczas możemy ją zapisać w tym samym wierszu co **if**

Przykład 2.5. Program wyznaczający większą z zadanych dwóch liczb stosując skrót dla **if**.
PROGRAM:

```
x=100
y=40
if x>y: print("x jest wieksze od y")
```

WYNIK:

```
x jest wieksze od y
```

Jeżeli jest tylko jeden *warunek* do wykonania, tj. jeden dla **if** oraz jeden dla **else** wówczas można zapisać całą instrukcję warunkową w jednym wierszu.

Przykład 2.6. Program wyznaczający większą z zadanych dwóch liczb stosując skrót dla **if ... else**.

PROGRAM:

```
x=30
y=400
print("x jest wieksze") if x>y else print("y jest wieksze")
```

WYNIK:

```
y jest wieksze
```

2.2.1. *Słowo kluczowe* **and**. Słowo kluczowe **and** używane jest do łączenia warunków.

p	0	0	1	1
q	0	1	0	1
p **and** q	0	0	0	1

Przykład 2.7. Program wyznaczający największą z zadanych trzech liczb stosując słowo **and**.

PROGRAM:

```
pierwsza=200
druga=33
trzecia=500
if pierwsza>druga and trzecia>pierwsza:
    print("Obydwa warunki sa prawdziwe")
```

WYNIK:

```
Obydwa warunki sa prawdziwe
```

2.2.2. *Słowo kluczowe* **or**. Słowo kluczowe **or** jest operatorem logicznym i również służy do łączenia warunków

p	0	0	1	1
q	0	1	0	1
p **or** q	0	1	1	1

Przykład 2.8. Program wyznaczający największą z zadanych trzech liczb stosując słowo **or**.

PROGRAM:

```
pierwsza=200
druga=33
trzecia=500
if pierwsza>druga or pierwsza>trzecia:
    print("Jeden z warunkow jest prawdziwy")
```

WYNIK:

```
Jeden z warunkow jest prawdziwy
```

2.2.3. *Zagnieżdżanie.*

```
if warunek1:              # Jeśli jest spełniony wykonaj blok instrukcji
    instrukcja1
    if warunek2 :         # Zagnieżdżona instrukcja warunkowa
        instrukcja2
    else:                 # else zagnieżdżonej instrukcji warunkowej
        instrukcja3
else:                     # else nadrzędnej instrukcji warunkowej
    instrukcja4
```

Przykład 2.9. Program sprawdzający czy zadana liczba jest większa (lub nie) od 10 i od 20 stosując zagdnieżdżanie.

PROGRAM:

```
x=41
if x>10:
    print("Wieksza od 10")
    if x>20
        print("a takze wieksza od 20")
    else:
        print("ale nie wieksza od 20")
```

WYNIK:

```
Wieksza od 10
a takze wieksza od 20
```

2.2.4. *Instukcja* **pass**. Jeżeli **if** nie zawiera instrukcji do wykonania wówczas wpisanie **pass** zapobiega wypisaniu przez program błędu.

Przykład 2.10. Program używający instukcji **pass** aby uniknąć błędu.

PROGRAM:

```
a=33
b=100
if b>a:
    pass
```

WYNIK:

```
```

2.3. **Pętle w języku Python.** Pętla jest to wybrana sekwencja kodu, która jest wykonywana wielokrotnie aż do momentu gdy zadany warunek przestaje być spełniony. W języku Python występują dwa rodzaje pętli:

- pętla **while**
- pętla **for**

2.4. **Pętla while.** Pętla **while** jest wybranym blokiem kodu wykonywanym nieznaną i nieokreśloną ilość razy, aż do momentu gdy zadany warunek pętli przestaje być spełniony.

```
while warunek1:           # Jeśli jest spełniony wykonaj blok instrukcji
    instrukcja1
```

```
    instrukcja2
    ...
instrukcja3          # Instrukcja następna po pętli
```

Przykład 2.11. Program wypisujący liczby od 1 do 5 używający pętli **while** .
PROGRAM:

```
i=1
while i<6:
    print(i)
    i=i+1
```

WYNIK:

```
1
2
3
4
5
```

Zauważmy, że na początku programu należy zadeklarować index i jako zmienną (całkowitą) oraz nadać jej wartość początkową równą 1. Następnie należy zwiększać jej wartość w każdej iteracji o 1, w przeciwnym wypadku pętla wykonywała by się w nieskończoność.

Drugim istotnym aspektem są tutaj tzw. wcięcia. Zauważmy, że instrukcje wykonywane w bloku pętli są zapisane po tabulatorze. Zapisanie instrukcji bez tabulatora oznacza, że nie należy ona już do bloku pętli i bedzie wykonywana po spełnieniu warunku pętli i zakonczeniu jej działania.

2.4.1. *Instrukcja* **break**. Instrukcja **break** powoduje przerwanie działania i wyjście z pętli nawet jeżeli warunek pętli został spełniony

Przykład 2.12. Program przerywający pętlę **while** gdy i jest równe 3.
PROGRAM:

```
i=1
while i<6:
    print(i)
    if (i==3):
        break
    i=i+1
```

WYNIK:

```
1
2
3
```

2.4.2. *Instrukcja* **continue**. Można zatrzymać bierzącą iterację używając instrukcji **continue** , a następnie kontynuować kolejne iteracje.

Przykład 2.13. Program przerywający pętlę na iteracji 3 i kontynułujący od kolejnej iteracji.
PROGRAM:

```
i=0
while i<6:
    i+=1
    if (i==3):
        continue
    print(i)
```

WYNIK:

```
1
2
4
5
6
```

2.4.3. *Instrukcja* **else**. W pętli while możliwe jest wykonanie bloku instrukcji jeżeli warunek nie jest już prawdziwy

```
while warunek1:     # Jeśli jest spełniony wykonaj blok instrukcji
    instrukcja1
    instrukcja2
    ...
else:               # Jeśli nie jest spełniony wykonaj blok instrukcji
    instrukcja3
    instrukcja4
    ...
instrukcja5         # Instrukcja następna po pętli
```

Przykład 2.14. Program wypisujący tekst w przypadku gdy warunek pętli while jest fałszywy
PROGRAM:

```
i=1
while i<6:
    print(i)
    i+=1
else:
    print('i nie jest juz mniejsze od 6')
```

WYNIK:

```
1
2
3
4
5
i nie jest juz mniejsze od 6
```

W powyżym przykładzie instrukcje pętli **while** są wykonywane dopóki warunek jest spełniony. Gdy ten warunek przestaje być prawdziwy, wówczas wykonana zostanie instrukcja z bloku **else**

2.5. **Pętla for.** Pętla **for** używana jest do formułowania iteracji dla takich obiektów jak na przykład listy, krotki, słowniki, zbiory lub łańcuchy. Za pomocą tej pętli można wywołać sekwencję instrukcji dla każdego elementu listy, krotki, zbioru, itd.

Przykład 2.15. Program wypisujący poszczególne elementy z zadanej listy
PROGRAM:

```
owoce=["ananas", "banan", "cytryna"]
for x in owoce:
    print(x)
```

WYNIK:

```
ananas
banan
cytryna
```

Uwaga 2.1. Pętla **for** działa dla zmiennych nieindeksowanych.

2.5.1. *Pętla na łańcuchu znaków.* Możliwe jest iterowanie po poszczególnych znakach w łańcuchu znaków.

Przykład 2.16. Program wypisujący poszczególne elementy z zadanej listy
PROGRAM:

```
for x in "banan":
    print(x)
```

WYNIK:

```
b
a
n
a
n
```

2.5.2. *Instrukcja* **break**. Break przerywa działanie pętli w dowolnie wybranym przez nas miejscu.

Przykład 2.17. Program wychodzący z pętli instrukcją **break** gdy x = "banan" .
PROGRAM:

```
owoce=["ananas", "banan", "cytryna"]
for x in owoce:
    print(x)
    if x=="banan":
        break
```

WYNIK:

```
ananas
banan
```

2.5.3. *Instrukcja* **continue**. . Możliwe jest wstrzymanie wykonywania instrukcji pętli w pewnych iteracja a następnie powrót do ich wykonywania za pomocą instrukcji **continue**

Przykład 2.18. Program pomijający wypisanie elementu "banan" przy użyciu instrukcji **continue**.

PROGRAM:

```
owoce=["ananas", "banan", "cytryna"]
for x in owoce:
    if x=="banan":
        continue
    print(x)
```

WYNIK:

```
ananas
cytryna
```

2.5.4. *Funkcja* **range** *w pętli* **for**. Funkcja **range** określa ile razy pętla **for** będzie wykonywana. Domyśle odliczanie rozpoczyna się od 0 i zwiększa licznik o 1.

Przykład 2.19. Program wypisujący liczby od 0 do 5 przy użyciu funkcji **range()**.

PROGRAM:

```
for x in range(6):
    print(x)
```

WYNIK:

```
0
1
2
3
4
5
```

Można również zadać własny zakres, np. **range(2,6)** wówczas odliczanie rozpocznie się od 2 a skończy na 6 (ale nie włącznie).

Przykład 2.20. Program wypisujący liczby od 2 do 5 przy użyciu funkcji **range()**.

PROGRAM:

```
for x in range(2,6):
    print(x)
```

WYNIK:

```
2
3
4
5
```

Możliwe jest również zadeklarowanie skoku licznika w funkcji **range()**. Należy wówczas dodać dodatkowy parametr, np. **range(2,30,3)**.

Przykład 2.21. Program wypisujący ciąg liczb ze skokiem wynoszącym 3 (domyślnie skok wynosi 1).

PROGRAM:

```
for x in range(2,20,3):
    print(x)
```

WYNIK:

```
2
5
8
11
14
17
```

2.5.5. *Instrukcja* **else** *w pętli* **for**. Można dodać dodatkową instrukcję do pętli **for** w przypadku gdy zakonczy ona działanie.

Przykład 2.22. Program wypisujący ciąg liczb od 0 do 5 oraz wypisujący wiadomość gdy pętla zakoćzy działanie.

PROGRAM:

```
for x in range(6):
    print(x)
else:
    print("Wreszcie skonczone!")
```

WYNIK:

```
0
1
2
3
4
5
Wreszcie skonczone!
```

W przypadku gdy użyjemy w pętli komendy **break**, blok **else** nie zostanie wykonany.

Przykład 2.23. Program przerywający działanie pętli gdy x=3 oraz nie wypisujący wiadomości.

PROGRAM:

```
for x in range(6):
    if x==3: break
    print(x)
else:
    print("Wreszcie skonczone!")
```

WYNIK:

```
0
1
2
```

2.5.6. *Zagnieżdżanie pętli* **for**. Zagnieżdżanie pętli **for** polega na wywołaniu pętli **for** wewnątrz pętli **for**. Wówczas pętla "wewnętrzna" bedzie wykonywana w każdej iteracji pętli "zewnętrznej".

Przykład 2.24. Program wypisujący każdy z zadanych przymiotników do każdego z zadanych obiektów.

PROGRAM:

```
opis=['dojzaly','slodki','smaczny']
owoce=['ananas','melon']
for x in opis:
    for y in owoce:
        print(x,y)
```

WYNIK:

```
dojzaly ananas
dojzaly melon
slodki ananas
slodki melon
smaczny ananas
smaczny melon
```

3. Łańcuchy znaków i operacje na łańcuchach

Łańcuch (string) to uporządkowany ciąg znaków używany do przechowywania informacji tekstowej. Łańcuchy składające się z pojedynczych znaków można traktować jako znaki. Odczytanie kodu ASCII konkretnego znaku odbywa się przy użyciu funkcji **ord(c)**, natomiast operacje odwrotna, przy użyciu funkcji **chr(c)**:

Przykład 3.1. Kody ASCII dla pojedyńczych znaków.
Program:

```
print(ord('A'))
print(ord('1'))
print(chr(65))
print(chr(49))
print(chr(10))
```

Wynik:

```
65
49
A
1

```

3.1. **Operacje na łańcuchach.** Łańcuchy można mnożyć i dodawać:

Przykład 3.2. Mnożenie i dodawanie łańcuchów.
Program:

```
h = " Huraaa!"          # Uwaga: Pierwszy znak to spacja!
P = " Uczymy sie Pythona!"  # Tutaj tez!
print(3*h + P)
```

Wynik:

```
Huraaa! Huraaa! Huraaa! Uczymy sie Pythona!
```

Łańcuchy zaliczane są do niezmiennych sekwencji, czyli nie można ich zmieniać w obszarze przechowywania ich w pamięci. Stałe łańcuchowe mogą być zapisywane z użyciem apostrofów lub znaków cudzysłowu. Dzięki temu możemy wstawiać znak cudzysłowu/apostrofu wewnątrz łańcucha bez specjalnych zabiegów. Wielowierszowe bloki wygodnie jest zapisywać wewnątrz potrójnych znaków, ponieważ zachowywane są końce linii.

Długość łańcucha wyznacza funkcja **len(h)**,

Przykład 3.3. Program:

```
print(len(h))
```

Wynik:

```
8
```

Znaki w łańcuchu są indeksowane, a ich numeracja zaczyna się od 0. Poszczególne znaki w łańcuchu można odczytywać od początku lub od końca. Od początku są indeksowane liczbami: 0,

1, 2, 3..., a od końca: -1, -2, -3 itd. W podobny sposób możemy pobrać część łańcucha od przodu lub od tyłu podając interesujący nas przedział indeksów stosując dwukropek:

Przykład 3.4. PROGRAM:

```
print(h[2])
print(h[-5])
print(h[:3])
print(h[3:])
print(h[3:5])
print(h[1:7:2])     # Co drugi znak z przedzialu od 1 do 7
print(h[-6:-2])
```

WYNIK:

```
u
r
Hu
raaa!
ra
Hra
uraa
```

3.2. **Formatowanie łańcuchów.** Poniżej przedstawione zostały niektore operacje na łańcuchach służące do formatowania tego łańcucha.

Operacja	Wynik działania	Opis działania
s = 'python jest FAJNY'	python jest FAJNY	deklaracja łańcucha s
s.capitalize()	Python jest fajny	zamienia pierwszą litere łańcucha na wielką a pozostałe zamienia na małe
s.lower()	python jest fajny	zamienia wszystkie litery w łańcuchu na małe
s.upper()	PYTHON JEST FAJNY	zamienia wszystkie litery w łańcuchu na wielkie
s.swapcase()	PYTHON JEST fajny	odwraca wielkość liter w napisie
s.title()	Python Jest Fajny	zmienia wielkość liter jak w tytule
s.replace('FAJNY','ciekawy')	python jest ciekawy	zamienia wszystkie wystąpienia danego ciągu znaków na inny
s.rjust()	python jest FAJNY	wyrównuje łańcuch do prawej strony dodając spacje od lewej
s.center(64)	python jest FAJNY	służy do wyśrodkowania tekstu w polu o zadanej długości
s.center(64, '*')	***python jest FAJNY***	dodatkowo wypełnia puste miejsce znakiem *

Operacja	Wynik działania	Opis działania
s.find('y')	1	wskazuje indeks pierwszego wystąpienia wskazanego ciągu znaków
s.rfind('N')	15	znajduje ostatnie wystąpienie określonego ciągu znaków
s.isdigit()	False	sprawdza, czy łańcuch zawiera tylko cyfry
s.count('p')	1	zwraca ilość wystąpień łańcucha (tutaj 'p') w łańcuchu s
s.split()	['python', 'jest', 'FAJNY']	tworzy listę wyrazów z łańcucha
s.split('t')	['py', 'hon jes', ' FAJNY']	możemy wskazać znak rozdzielający
((s+'\n') * 3).splitlines()	['python jest FAJNY', 'python jest FAJNY', 'python jest FAJNY']	dzieli tekst na linie

Łańcuchy w Pythonie należą do typów niezmiennych! Oznacza to, że nie można podmieniać znaków w łańcuchu, ani do nich nic dodawać lub obcinać, lecz tylko utworzyć nowy łańcuch zawierający podmienione, dodane lub obcięte znaki:

Przykład 3.5. Program:

```
t1 = 'tak'
print(t1)
t1 = t1 + 'tyka'    # tworzy nowy łańcuch t1 na podstawie starego łańcucha t1
                    # zapis t1 += 'tyka' jest błędny
print(t1)
t1 = t1[:4]         # zwróci nowy łańcuch t1 o obciętej zawartości starego t1
print(t1)
```

Wynik:

```
tak
taktyka
takt
```

3.3. **Metoda** *format()***.** Metoda **format()** służy do formatowania wyświetlanego łańcucha znaków lub jego części.

Przykład 3.6. Program wyświetlający cenę baryłki ropy.
Program:

```
cena = 83
tekst = 'Cena barylki ropy to {} USD'
print(tekst.format(cena))
```

Wynik:

```
Cena barylki ropy to 83 USD
```

Możliwe jest dodanie parametrów wewnątrz nawiasów klamrowych w celu przekonwertowania wartości

Przykład 3.7. Program wyświetlający cenę w formacie dziesiętnym z dwoma miejscami po przecinku.

PROGRAM:

```
cena = 83.845
tekst = 'Cena barylki ropy to {:.2f} USD'
print(tekst.format(cena))
```

WYNIK:

```
Cena barylki ropy to 83.84 USD
```

Możliwe jest również dodanie wielu wartości wewnątrz łańcucha.

Przykład 3.8. Program wyświetlający wiele informacji dotyczących ceny baryłki ropy.

PROGRAM:

```
cena = 83.845
data = '30-10-2023'
zrodlo = 'TradingView'
tekst = "Cena barylki ropy na dzien {} to {:.2f} USD wedlug {}."
print(tekst.format(data,cena,zrodlo))
```

WYNIK:

```
Cena barylki ropy na dzien 30-10-2023 to 83.84 USD wedlug TradingView.
```

Dodawane wartości mogą być wstawiane w klamrach bezpośrednio w łańcuchu ale wówczas należy je zdefiniować w module **format()**

Przykład 3.9. Program wyświetlający wiele informacji dotyczących ceny baryłki ropy.

PROGRAM:

```
tekst = "Cena barylki ropy na dzien {data} to {cena:.2f} USD wedlug {zrodlo}."
print(tekst.format(data='30-10-2023',cena=83.845,zrodlo='TradingView'))
```

WYNIK:

```
Cena barylki ropy na dzien 30-10-2023 to 83.84 USD wedlug TradingView.
```

4. Struktury danych

W celu umożliwienia reprezentacji danych powiązanych ze sobą różnymi relacjami potrzebne są struktury danych pozwalające odwzorować te zależności. Do podstawowych struktur danych w Pythonie należą:

- **Lista** – to struktura liniowa zawierająca uporządkowany zestaw obiektów. Lista jest zmiennym typem danych umożliwiającym dodawanie i usuwanie elementów z listy. Elementy listy zapisujemy w nawiasach kwadratowych.

```
lista = ['jablko', 'gruszka', 'sliwka', 'morela', 'mango', 'ananas', 'brzoskwinia']
```

- **Krotka** – to typ niezmienny, a więc nie można jej modyfikować po jej utworzeniu. Elementy krotki zapisujemy w nawiasach okrągłych. Jeśli krotka jest jednoelementowa, obligatoryjnie stawiamy przecinek na końcu.

```
pora = ('noc',)
krotka = ('jablko', 'gruszka', 'sliwka', 'morela', 'mango', 'ananas', 'brzoskwinia')
```

- **Słownik** – to tablica asocjacyjna przechowująca klucze i przypisane im wartości, przy czym klucze muszą być unikalne. Zbiór takich par ujęty jest w nawiasy klamrowe:

```
slownik = {klucz1 : wartosc1, klucz2 : wartosc2, ..., kluczN : wartoscN}
```

  Słowniki nie sortują automatycznie danych, więc jeśli dane mają być posortowane, trzeba to zrobić zewnętrznie.
- **Zbiór** – to nieuporządkowany zestaw prostych obiektów, który umożliwia wykonywanie operacji na zbiorach oraz sprawdzanie występowania elementu w nim:
  Program:

```
Waluty = set (['USD', 'EUR', 'GBP', 'CHF'])
print('USD' in Waluty)
print('SEK' not in Waluty)
```

  Wynik:

```
True
True
```

- **Odniesienie** – to wskaźnik do miejsca w pamięci komputera, w którym znajdują się określone dane. Tworzenie odniesień to wiązanie (bindowanie) nazwy z obiektem.

4.1. Listy.

Lista – to typ zmienny, więc można podmieniać jej elementy, dodawać i usuwać. Lista może też być pusta:

```
lista = []
```

Listy mogą zawierać elementy różnych typów a także można tworzyć listę list lub lista może zawierać inną listę jako swój element:
Program:

```
x = 34; y = 12; z = 18
lista = [1, 3.5, x, 'pies']
listalist = [ [1, 2, 3, 4], [x, y, z], [ 'wtorek', 'czwartek', 'sobota' ] ]
```

Długość listy obliczymy przy pomocy funkcji **len(lista)**:
PROGRAM:

```
print(len(lista))
print(len(listalist))
```

WYNIK:

```
4
3     # gdyż listalist zawiera 3 elementy, które są listami
```

Dostęp do elementów listy uzyskujemy w podobny sposób jak dla łańcuchów:
PROGRAM:

```
print(lista[2])
print(listalist[1][2])
print(lista[-1])
print(lista[1:3])
print(lista[2:])
```

WYNIK:

```
34
18
'pies' ?
[3.5, 34]
[34, 'pies']
```

Listy można powielać:
PROGRAM:

```
LiczbyNaturalne = [1, 2, 3, 4, 5, 6, 7, 8, 9]
LiczbyNaturalne*=2
```

Spowoduje zwielokrotnienie:
WYNIK:

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

Elementy można podmieniać:

```
LiczbyNaturalne[9:18]=[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Spowoduje podmianę wskazanych elementów listy:
WYNIK:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Można też wykonywać operacje na poszczególnych jej elementach

```
for i in range (0,19): LiczbyNaturalne[i]*=2
```

Spowoduje przemnożenie każdego z jej elementów przez 2:
WYNIK:

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
```

Kasowanie elementów listy:

```
del LiczbyNaturalne[9:17]
```

skróci listę do
WYNIK:

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 36, 38]
```

Istnieje możliwość porównywania list. Wówczas:

- Listy są równie, jeśli wszystkie elementy obu list są równe.
- Listy są porównywane w porządku leksykograficznym, najpierw pierwsze, potem drugie i kolejne elementy, a dzięki temu można wyznaczyć, która lista jest większa lub mniejsza.
- Nie decyduje tutaj długość listy.
- Element nieistniejący jest zawsze mniejszy od każdego innego elementu

Przykład 4.1. Sprawdzanie, czy lista zawiera lub nie zawiera pewien element:
PROGRAM:

```
LiczbyNaturalne = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(1 in lista LiczbyNaturalne)
print(0 not in lista LiczbyNaturalne)
```

WYNIK:

```
True
True
```

Wszystkie sekwencje zmienne w Pythonie wskazują na miejsce w pamięci, w którym te zmienne się znajdują (wskaźniki). Zatem przypisanie:

```
NowaLista = LiczbyNaturalne
```

przepisze jedynie wskaźnik, a nie zrobi kopii elementów listy LiczbyNaturalne.

4.1.1. *Metody na listach.* Niech będzie dana lista

$$\text{Lista} = [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

Metoda **append()** dołącza do listy pojedynczy element na jej końcu:

Metoda	Wynik
Lista.append(10)	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Metoda **extend()** dołącza do listy inną listę na jej końcu:

Metoda	Wynik
Lista.extend([10, 11])	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 11]

Metoda **insert(i, w)** wstawia do listy wartość w na pozycję i:

Metoda	Wynik
Lista.insert(5, 10)	[1, 2, 3, 4, 5, 10, 6, 7, 8, 9, 10, 10, 11]

Metoda **pop(i)** zwraca wartość z i-tej pozycji listy i równocześnie usuwa ją z listy:

Metoda	Wynik
Lista.pop(7)	7

Metoda **remove(w)** usuwa z listy pierwszą wartość w:

Metoda	Wynik
Lista.remove(10)	[1, 2, 3, 4, 5, 6, 8, 9, 10, 10, 11]

Metoda **count(w)** liczy, ile razy występuje w liście wartość w:

Metoda	Wynik
Lista.count(10)	2

Metoda **index(w)** znajduje indeks pierwszego wystąpienia wartości w w liście:

Metoda	Wynik
Lista.index(10)	8

Można też ograniczyć przeszukiwanie do wybranej części listy **index(w, od, do)**:

Metoda	Wynik
Lista.index(10, 9, 10)	9

Metoda **reverse()** odwraca kolejność elementów listy:

Metoda	Wynik
Lista.reverse()	[11, 10, 10, 9, 8, 6, 5, 4, 3, 2, 1]

Metoda **sort()** porządkuje elementy listy w kolejności rosnącej:

Metoda	Wynik
Lista.sort()	[1, 2, 3, 4, 5, 6, 8, 9, 10, 10, 11]

Metoda **clear()** usunie wszystkie elementy listy

Metoda	Wynik
Lista.clear()	[]

4.2. **Krotki (***ang. Tuple***).** Krotka to typ niezmienny, a więc nie można podmieniać jej elementów, dodawać ani ich usuwać. Pozostałe cechy krotek przypominają listy. Krotki mogą zawierać różne elementy również te o typie zmiennym:

PROGRAM:

```
x = 34; y = 12; z = 18
LiczbyNaturalne = [1, 2, 3, 4, 5, 6, 7, 8, 9]
krotka = (1, 3.5, x, 'pies', LiczbyNaturalne)
```

Krotka może zawierać listę typu zmiennego, co oznacza, że listy zmienić nie możemy, natomiast możemy wykonywać operacje na elementach listy jak na typie zmiennym: Dostęp do poszczególnych elementów krotki uzyskujemy podobnie jak w liście:

PROGRAM:

```
print(krotka[1])
krotka[4][5]*=2 # dokonuje operacji na liscie wewnatrz krotki
print(len(krotka))
```

WYNIK:

```
3.5
(1, 3.5, 34, 'pies', [1, 2, 3, 4, 5, 12, 7, 8, 9])
5
```

Do elementów krotki uzyskujemy dostęp podobnie jak dla listy:

PROGRAM:

```
print(krotka[4:])    # zwróci krotkę jednoelementową składającą się z listy:
                     # a jednoelementowe krotki kończą się zawsze przecinkiem
```

WYNIK:

```
([1, 2, 3, 4, 5, 12, 7, 8, 9],)
```

Skrócenie krotki można wykonać tylko przez utworzenie nowej krotki na podstawie starej obciętej o pożądaną ilość elementów, np.:

PROGRAM:

```
krotka = krotka[:4]
print(krotka)
```

WYNIK:

```
(1, 3.5, x, 'pies')
```

Przedłużenie krotki również wymaga stworzenie nowej krotki przez dodanie do starej krotki nowych elementów:

PROGRAM:

```
krotka = krotka + (5,)
print(krotka)
```

WYNIK:

```
(1, 3.5, 34, 'pies', 5)
```

W przypadku krotek dochodzi do fizycznego kopiowania elementów, więc takie operacje na dużych krotkach są bardziej czasochłonne niż na listach:

PROGRAM:

```
krotka2 = krotka
```

4.2.1. *Konwersje sekwencji.*

- **list()** – zamienia typ sekwencyjny na listę

- **tuple()** – zamienia typ sekwencyjny na krotkę
- **str()** – zamienia typ sekwencyjny na napis (uwaga nie jego elementy):

Przykład 4.2. Konwersja krotki na listę oraz lańcucha na kortkę.
PROGRAM:

```
krotka = (1, 2, 3, 4)
lancuch = 'abcd'
lista1 = list(krotka)
print(lista1)
lista2 = list(lancuch)
print(lista2)
krotka = tuple(lista2)
print(krotka)
```

WYNIK:

```
[1, 2, 3, 4]
['a', 'b', 'c', 'd']
('a', 'b', 'c', 'd')
```

Przykład 4.3. Konwersja typu sekwencyjnego na łańcuch znaków.
PROGRAM:

```
napis1 = str(lista1)
napis2 = str(lista2)
napis3 = str(krotka)
print(napis1)
print(napis2)
print(napis3)
# lub krócej przy pomocy odwróconych apostrofów:
napis4 = `lista1`
print(napis4)
```

WYNIK:

```
'[1, 2, 3, 4]'
"['a', 'b', 'c', 'd']"
"('a', 'b', 'c', 'd')"
'[1, 2, 3, 4]'
```

4.3. **Słowniki.** Słowniki – nazywane też tablicami asocjacyjnymi wiążą ze sobą dwie wartości: wartość klucza z wartością przechowywanej danej. Słowniki od tablic, list, krotek i sekwencji różni to, iż są one indeksowane wartościami kluczy, a nie kolejnymi liczbami naturalnymi. Słownik jest więc zbiorem par:

(klucz, wartość)

Klucze muszą być unikalne, gdyż każdemu kluczowi przypisana jest jakaś wartość. Klucz musi być dowolnym obiektem niezmiennego typu.

Słownik odwzorowuje więc obiekty dowolnego typu niezmiennego na inne obiekty dowolnego typu. Odwzorowanie to jest z punktu widzenia szybkości działania słownika jednostronne, aczkolwiek można szukać klucza odpowiadającego wartości.

Przykład 4.4. Program:

```
telefonyalarmowe = {'pogotowie': 999, 'straz': 998, 'policja': 997}
print(telefonyalarmowe["pogotowie"])
```

Wynik:

```
999
```

Słownik można rozszerzać o nowe pary elementów.

Przykład 4.5. Program:

```
telefonyalarmowe ["pogotowie gazowe"] = 992
print(telefonyalarmowe)
```

Wynik:

```
{'pogotowie': 999, 'straz': 998, 'policja': 997, 'pogotowie gazowe': 992}
```

Możemy też modyfikować istniejące wartości:
Program:

```
telefonyalarmowe ["policja"] = 112
print(telefonyalarmowe)
```

Wynik:

```
{'pogotowie': 999, 'straz': 998, 'policja': 112, 'pogotowie gazowe': 992}
```

Zawartość jednego słownika możemy przekopiować do innego słownika

Przykład 4.6. Program:

```
telefony = {'taxi': 9622, 'dom': 123456789}
telefondomowy = {'dom': 129876543}
telefonyalarmowe = {'pogotowie': 999, 'straz': 998, 'policja': 997}
mojetelefony = telefony.copy() # tworzy nowy słownik, do którego kopiuje pary
print(mojetelefony)
```

Wynik:

```
{'taxi': 9622, 'dom': 123456789}
```

Można także usuwać elementy słownika

Przykład 4.7. Program:

```
del telefony['taxi'] # usuwa parę wskazaną przez klucz
print(telefony)
telefony.clear() # czyści cały słownik usuwając wszystkie pary
print(telefony)
del telefony # usuwa cały słownik
print(telefony)
```

Wynik:

```
{'dom': 123456789}
{}
NameError: name 'telefony' is not defined
```

Funkcja **.update()** aktualizuje wartości w słowniku wg innego słownika na podstawie kluczy

Przykład 4.8. PROGRAM:

```
telefony = {'taxi': 9622, 'dom': 123456789}
telefondomowy = {'dom': 129876543}
telefony.update(telefondomowy)
print(telefony)
```

WYNIK:

```
{'taxi': 9622, 'dom': 129876543}
```

Funkcja **.pop()** łączy operację odczytu wartości z usunięciem pary o tym kluczu ze słownika po odczytaniu wartości

Przykład 4.9. PROGRAM:

```
print(telefony.pop('taxi'))
print(telefony)
```

WYNIK:

```
9622
'dom': 129876543
```

Funkcja **.popitem()** usuwa ostatnią parę ze słownika po jej odczycie

Przykład 4.10. PROGRAM:

```
print(telefonyalarmowe.popitem())
print(telefonyalarmowe)
```

WYNIK:

```
('policja', 997) # otrzymamy krotkę !
'pogotowie': 999, 'straz': 998
```

Funkcja **.keys()** zwraca listę kluczy
PROGRAM:

```
print(telefonyalarmowe.keys())
```

WYNIK:

```
dict keys(['pogotowie', 'straz'])
```

Funkcja **.values()** zwraca listę wartości
PROGRAM:

```
print(telefonyalarmowe.values())
```

WYNIK:

```
dict values([999, 998])
```

W celu sprawdzenia, czy określony klucz występuje w słowniku stosujemy słowo kluczowe **in**.
PROGRAM:

```
print('taxi' in telefony)
```

WYNIK:

```
False
```

W celu sprawdzenia, czy określona wartość występuje w słowniku użyjemy instrukcji:
PROGRAM:

```
print(997 in telefony.values())
```

WYNIK:

```
False
```

Określenie ilości par w słowniku zwraca funkcja **len()**
PROGRAM:

```
print(len(telefonyalarmowe))
```

WYNIK:

```
2
```

Konwersja słownika na łańcuch (napis):
PROGRAM:

```
print(str(telefonyalarmowe))
```

WYNIK:

```
"{'pogotowie': 999, 'straz': 998}"
```

4.4. **Zbiory.** W Pytonie występują dwa rodzaje zbiorów:

- zbiory zmienne
- zbiory niezmienne

Przykład 4.11. Program definiujący zbiory zmienne i niezmienne.
PROGRAM:

```
O={}                              # zbior pusty
A = set([1, 2, 3, 4, 5, 6, 7])            # zbior zmienny
B = frozenset(['apple',"banana",'cherry'])      # zbior niezmienny
print(O)
print(A)
print(B)
```

WYNIK:

```
()
{1, 2, 3, 4, 5, 6, 7}
frozenset({'apple', 'banana', 'cherry'})
```

Zbiory są

- nieuporządkowane, to znaczy elementy zbioru nie mają ustalonego porządku, czyli mogą wyświetlać się w różnej kolejności, a co za tym idzie, nie można odwoływać się do nich przy pomocy indeksu.
- niezmiennicze to znaczy nie można zmieniać elementów zbioru jeżeli został on już zdefiniowany. Można natomiast usuwać elementy ze zbioru a także dodawać do niego nowe elementy.
- nie zawierające duplikatów, to znaczy nie można mieć w jednym zbiorze dwóch identycznych elementów.

Przykład 4.12. Program definiujący zbiory zmienne i niezmienne.
PROGRAM:

```
AA = set([1, 3, 3, 6, 4, 4, 4, 5])
BB = frozenset(['apple',"banana",'cherry'])
CC = {'apple',"banana",'cherry'}
print(AA)
print(BB)
print(CC)
```

WYNIK 1:

```
{1, 3, 4, 5, 6}
frozenset({'apple', 'banana', 'cherry'})
{'apple', 'banana', 'cherry'}
```

WYNIK 2:

```
{1, 3, 4, 5, 6}
frozenset({'apple', 'cherry', 'banana'})
{'apple', 'cherry', 'banana'}
```

WYNIK 3:

```
{1, 3, 4, 5, 6}
frozenset({'banana', 'cherry', 'apple'})
{'banana', 'cherry', 'apple'}
```

Zauważmy, że w przypadku zbiorów nie wiadomo w jakiej kolejności wystąpią jego elementy.

Aby wyznaczyc ilość elementów zbioru wykorzystamy funkcje **len()**
PROGRAM:

```
CC = {'apple',"banana",'cherry'}
print(len(CC))
```

WYNIK:

```
3
```

Przykład 4.13. Program definiujący (i wyświetlający) zbiór zawierający elementy różnych typów (string, int, boolean, float)

PROGRAM:

```
Z = {"abc", 34, True, 40.7}
print(Z)
print(type(Z))
```

WYNIK:

```
{40.7, True, 34, 'abc'}
<class 'set'>
```

Funkcja **set()** zastępuje nawiasy klamrowe podczas definiowania zbioru (patrz. Przykład **??**)

Przykład 4.14. PROGRAM:

```
MojZbior = set(('apple', 'banana', 'cherry'))
print(MojZbior)
```

WYNIK:

```
{'apple', 'banana', 'cherry'}
```

4.4.1. *Dostęp do elementów zbioru.* Elementy zbioru nie są indeksowane zatem odwołanie do danego elementu zbioru odbywa się poprzez przeszukiwanie zbioru za pomocą pętli.

Przykład 4.15. Program przeszukujący zbiór i wypisujący jego elementy

PROGRAM:

```
Z = {"abc", 34, True, 40.7}
for x in Z:
    print(x)
```

WYNIK:

```
40.7
True
34
abc
```

Przykład 4.16. Sprawdzanie czy dany element należy do zbioru

PROGRAM:

```
Z = {"abc", 34, True, 40.7}
print(34 in Z)
print(False not in Z)
```

WYNIK:

```
True
True
```

4.4.2. *Dodawanie i usuwanie elementów ze zbiorów.* W celu dodania elementu do zbioru używamy metody **add()** lub **update()**, wówczas element zostanie dodany ale nie koniecznie na końcu zbioru. Usuwanie elementu ze zbioru odbywa się za pomocą metod **remove()** lub **discard()**

Przykład 4.17. Program dodający i usuwający zadane elementy

PROGRAM:

```
owoce = {'apple', 'banana', 'cherry'}
owoce.add('orange')
print(owoce)
tropikalne= { 'mango', 'papaya'}
owoce.update(tropikalne)
print(owoce)
owoce.remove('apple')
owoce.discard('cherry')
print(owoce)
```

WYNIK:

```
{'banana', 'orange', 'apple', 'cherry'}
{'papaya', 'orange', 'apple', 'cherry', 'banana', 'mango'}
{'papaya', 'orange', 'banana', 'mango'}
```

Możliwe jest również zastosowanie metody **pop()** w celu usunięcia ostatniego elementu zbioru, ale ponieważ zbiory nie są uporządkowane zatem nigdy nie wiemy, który elementy zostanie usunięty.

4.4.3. *Operacje na zbiorach.* Suma zbiorów lub inaczej łączenie zbiorór odbywa się poprzez metody **union()** i **update()**.

Przykład 4.18. PROGRAM:

```
z1 = {'a', 'b', 'c'}
z2 = {1, 2, 3}
z3 = z1.union(z2)
print(z3)
z1.update(z2)
print(z1)
```

WYNIK:

```
{1, 'a', 3, 2, 'c', 'b'}
{2, 'a', 'b', 3, 1, 'c'}
```

Część wspólną zbiorów otrzymamy poprzez metodę **intersection_update()**

Przykład 4.19. PROGRAM:

```
z1 = {'apple', 'banana', 'cherry'}
z2 = {'microsoft', 'apple'}
z1.intersection_update(z2)
print(z1)
```

WYNIK:

```
{'apple'}
```

5. Funkcje w języku Python

Funkcja jest to blok kodu przeznaczony do wykonywania określonego ciągu operacji. Przykładem takiej funkcji, predefiniowanej w Pythonie jest funkcja **print()**.

W celu zdefiniowania funkcji używane jest słowo kluczowe **def**. Ogólna składnia funkcji w Pythonie jest następująca

```
def NazwaFunkcji(argument1, argument2, ..., argumentN):
    instrukcja1
    instrukcja2
    ...
    return WynikFunkcji
```

Istnieją cztery typy funkcji:

- bezargumentowe nie zwracające wartości
- bezargumentowe zwracające wartość
- argumentowe nie zwracające wartości
- argumentowe zwracające wartość

Przykład 5.1. Program deklaryjący funkcję bezargumentową nie zwracającą żadnej wartości
Program:

```
def MojaFunkcja():
    print("Czesc to ja twoja funkcja")
```

Wynik:

```
Czesc to ja twoja funkcja
```

Argumenty. Informacja wprowadzana do funkcji nazywana jest argumentem. Argumenty wprowadza się po nazwie funkcji i w nawiasie okrągłym. Argumenty mogą być dowolnego typu oraz może ich być dowolna ilość przy czym rozdzielane są przecinkami.

Przykład 5.2. Funkcja pobierająca argument w celu wypisania pełnego zdania.
Program:

```
def MojaFunkcja(powitanie):
    print(powitanie + "Python")
MojaFunkcja('Witaj')
MojaFunkcja('Hello')
MojaFunkcja('Hola')
```

Wynik:

```
Witaj Python
Hello Python
Hola Python
```

Funkcje należy wywoływać z taką ilością argumentów z jaką została ona zdefiniowana

Przykład 5.3. Funkcja pobierająca dwa argumenty w celu wypisania pełnego zdania.
Program:

```
def MojaFunkcja(a,b):
    print(a + " " + b)
MojaFunkcja("Funkcje sa","uzyteczne")
```

WYNIK:

```
Funkcje sa uzyteczne
```

Jeżeli w trakcie definiowania funkcji nie wiadomo z ilu argumentów będzie korzystać funkcja, wówczas można użyć gwiazdki $*$ przed nazwą argumentu.

Przykład 5.4. Funkcja o nieznanej liczbie argumentów.
PROGRAM:

```
def MojaFunkcja(*kids):
    print("Najstarsze dziecko to " + kids[1])
MojaFunkcja("Adam","Bartosz","Cezary")
```

WYNIK:

```
Najstarsze dziecko to Adam
```

5.1. **Argumenty jako słowa kluczowe.** Funkcje można wywołać z argumentami przypisanymi do kluczy.

Przykład 5.5. Funkcja pobierająca argument w celu wypisania pełnego zdania.
PROGRAM:

```
def MojaFunkcja(child1,child2,child3):
   print("Najstarsze dziecko to " + child1)
MojaFunkcja(child1="Adam",child2="Bartosz",child3="Cezary")
```

WYNIK:

```
Najstarsze dziecko to Adam
```

W przypadku gdy nie jest znana ilość kluczy do argumentów przekazanych do funkcję można zastosować symbol ** przed nazwą klucza

Przykład 5.6. PROGRAM:

```
def MojaFunkcja(**kid):
   print("Jego imie to " + kid["imie"])
MojaFunkcja(imie="Adam", nazwisko="Nowak")
```

WYNIK:

```
Jego imie to Adam
```

5.1.1. *Funkcje z domyślną wartością parametru.* Możliwe jest nadanie domyślnej wartości parametrowi funkcji wówczas wywołanie jej bez parametru przypisze parametrowi wartość zdefiniowaną jako domyślna.

Przykład 5.7. PROGRAM:

```
def MojaFunkcja(kraj="Norwegia"):
   print("Moj kraj urodzenia to " + kraj)
MojaFunkcja("Szwecja")
MojaFunkcja("Holandia")
MojaFunkcja()
MojaFunkcja("Hiszpania")
```

WYNIK:

```
Moj kraj urodzenia to Szwecja
Moj kraj urodzenia to Holandia
Moj kraj urodzenia to Norwegia
Moj kraj urodzenia to Hiszpania
```

5.1.2. *Lista jako artument funkcji.* Argument funkcji może być dowolnego typu.

Przykład 5.8. PROGRAM:

```
def MojaFunkcja(jedzenie):
   for x in jedzenie:
      print(x)
owoce = ['jablko', 'gruszka', 'wisnia']
MojaFunkcja(owoce):
```

WYNIK:

```
jablko
gruszka
wisnia
```

5.1.3. *Instrukcja return().* Funkcja może zwracać wartość używając instrukcji **return()**

Przykład 5.9. PROGRAM:

```
def MojaFunkcja(x):
   return 5*x
print(MojaFunkcja(3))
print(MojaFunkcja(5))
print(MojaFunkcja(9))
```

WYNIK:

```
15
25
45
```

5.2. **Rekurencja.** Rekurencja polega na wywoływniu przez funkcję samej siebie. Python dostarcza wielu narzędzi ułatwiających funkcje rekurencyjne. Należy jednak być uważnym ponieważ nieprawidłowe zaprogramowanie funkcji rekurencyjnej niesie za sobą poważne błedy polegające na przykład na zaprogramowaniu funkcji które nigdy się nie kończą a przy tym pochłaniają spore zasoby pamięci.

Przykład 5.10. Funkcja rekurencyjna
PROGRAM:

```
def rekurencja(k):
    if k>0:
        wynik = k + rekurencja(k-1)
        print(wynik)
    else:
        wynik = 0
return wynik
rekurencja(6)
```

WYNIK:

```
1
3
6
10
15
21
```

5.2.1. *Zakresy zmiennych.* Zmienna jest zawsze dostępna w obszarze, w którym jest zdefiniowana. Zmienne w Pythonie dzielimy na zmienne lokalne i zmienne globalne.

Przykład 5.11. Zmienna zdefiniowana wewnątrz funkcji jest zmienna lokalną.
PROGRAM:

```
def mojafunkcja():
    x=300
    print(x)
mojafunkcja()
```

WYNIK:

```
300
```

Przykład 5.12. Funkcja zdefiniowana wewnątrz funkcji.
PROGRAM:

```
def funkcjazew():
    x=300
    def funkcjawew():
        print(x)
funkcjazew()
```

WYNIK:

```
300
```

Przykład 5.13. Zmienna zdefiniowana w głównym obszarze kodu nazywana jest zmienną globalną. Zmienna globalna moze być wykorzystywana przez wszystkie funkcje zdefiniowane w programie.
PROGRAM:

```
x=300
def funkcja():
   print(x)
funkcja()
print(x)
```

Wynik:

```
300
300
```

5.2.2. *Nazewnictwo zmiennych lokalnych i globalnych.* Jeżeli operujemy na zmiennej o tej samej nazwie zarówno wewnątrz jak i na zewnątrz funkcji, wówczas Python traktuje pierwszą z nich jako zmienną lokalną natomiast drugą jako zmienną globalną.

Przykład 5.14. Program:

```
x=300
def funkcja():
   x=200
   print(x)
funkcja()
print(x)
```

Wynik:

```
200
300
```

5.2.3. *Słowo kluczowe global.* Możliwe jest wymuszenie statusu danej zmiennej jako globalna wewnątrz funkcji używając instrukcji **global**

Przykład 5.15. Program Python pokazujący wymuszenie globalności zmiennej.
Program:

```
def funkcja():
   global x
   x=300
funkcja()
print(x)
```

Wynik:

```
300
```

Przykład 5.16. Program:

```
x=300
def funkcja():
   global x
   x=200
funkcja()
print(x)
```

WYNIK:

```
200
```

5.3. **Funkcje anonimowe.** Jeśli funkcję zamierzamy wykorzystać tylko w jednym miejscu w programie, możemy wykorzystać funkcję **lambda** Jest to funkcja, która może przyjąć dowolną ilość argumentów ale zwraca tylko jedno wyrażenie. Składnia funkcji to:

lambda *argumenty* : *wyrażenie*

Jest to odwzorowaniem przestrzeni danych w przestrzeń wyników, które możemy od razu wykonać. Wadą wyrażeń lambda jest niemożność wykorzystania w nich instrukcji nie będących wyrażeniami, np. print, if, for, while. Wtedy trzeba skorzystać z funkcji.

Przykład 5.17. Program dodający 10 do argumentu i zwracający wynik:
PROGRAM:

```
x = lambda a: a+10
print(x(5))
```

WYNIK:

```
15
```

Przykład 5.18. Program mnożący argumenty funkjci przez siebie z wykorzystaniem funkjci lambda dwuargumentowej PROGRAM:

```
x=lambda a,b: a*b
print(x(5,6))
```

WYNIK:

```
30
```

Przykład 5.19. Program wyznaczający pierwiastek z 4 przy użyciu funkcji lambda
PROGRAM:

```
print (lambda x,y: x**y)(4, 0.5)
```

WYNIK:

```
2
```

Przykład 5.20. Program sumujący trzy argumenty
PROGRAM:

```
x=lambda a,b,c: a+b+c
print(x(5,6,2))
```

WYNIK:

```
13
```

Przykład 5.21. Program podwajający i potrajający zadaną wartość.
PROGRAM:

```
def mojafunkcja(n):
   return lambda a: a*n
podwojenie = mojafunkcja(2)
podtrojenie = mojafunkcja(3)
print(podwojenie(11))
print(podtrojenie(11))
```

WYNIK:

```
22
33
```

5.4. **Generatory.** Generatory są mechanizmem leniwej ewaluacji funkcji, tzn. zamiast zwracać obciążającą pamięć listę, generator jest obiektem przechowującym stan ostatniego wywołania, mogącym wielokrotnie wchodzić i opuszczać ten sam dynamiczny zakres. Wywołanie generatora umożliwia więc zwracanie kolejnych elementów z dynamicznego zakresu zamiast całej listy, co daje wygodny mechanizm do iterowania po kolejnych elementach dynamicznie generowanych elementów listy:

Przykład 5.22. Program definiujący generator N kwadratów liczb całkowitych.
PROGRAM:

```
def kwadraty(N):
   for i in range(N):
      yield i**2
for x in kwadraty(4):
   print(x)
```

WYNIK:

```
0
1
4
9
```

Generator zamiast słowa kluczowego *return* stosuje *yield* zwracający kolejny wygenerowany element ciągu / listy. Można to zapisać również w postaci:

Przykład 5.23. PROGRAM:

```
kwadraty = (i**2 for i in range(4))
for x in kwadraty:
   print(x)
```

WYNIK:

```
0
1
4
9
```

6. Moduły w języku Python

Funkcje można agregować i tworzyć moduły, (tak zwane biblioteki funkcji), które można zaimportować do wielu różnych programów przy pomocy instrukcji **import**. Nazwą pliku jest nazwa modułu z dodanym rozszerzeniem .py

Istnieje również standardowa biblioteka Pythona, którą importujemy za pomocą instrukcji **import sys**

Wbudowana funkcja **dir()** jest używana do sprawdzenia, jakie nazwy definiuje moduł. Zwraca uporządkowaną listę ciągów znaków:

Program:

```
import sys
print(dir(sys))
```

Wynik:

```
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', '__interactivehook__',
'__loader__', '__name__', '__package__', '__spec__', '__stderr__', '__stdin__', '__stdout__',
'__unraisablehook__', '_base_executable', '_clear_type_cache', '_current_exceptions',
'_current_frames', '_debugmallocstats', '_framework', '_getframe', '_getquickenedcount',
'_git', '_home', '_stdlib_dir', '_xoptions', 'abiflags', 'addaudithook', 'api_version', 'argv',
'audit', 'base_exec_prefix', 'base_prefix', 'breakpointhook', 'builtin_module_names',
'byteorder', 'call_tracing', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
'excepthook', 'exception', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
'get_int_max_str_digits', 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit',
'getrefcount', 'getsizeof', 'getswitchinterval', 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info', 'intern', 'is_finalizing', 'maxsize', 'maxunicode', 'meta_path',
'modules', 'orig_argv', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'platlibdir', 'prefix', 'pycache_prefix', 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth',
'set_int_max_str_digits', 'setdlopenflags', 'setprofile', 'setrecursionlimit', 'setswitchinterval',
'settrace', 'stderr', 'stdin', 'stdlib_module_names', 'stdout', 'thread_info', 'unraisablehook',
'version', 'version_info', 'warnoptions']
```

Do funkcji i zmiennych wewnątrz modułu odwołujemy się przy pomocy kropki, wskazując wyraźnie przed kropką z jakiego modułu ta funkcja lub zmienna pochodzi.

Program:

```
sys.path
sys.argv
```

Chcąc uzyskać informacje o katalogu, w którym jesteśmy można posłużyć się instrukcją:

Program:

```
import os
print(os.getcwd())
```

Moduły można kompilować w celu przyspieszenia operacji na nich, wtedy posiadają rozszerzenie .pyc.

6.1. **Importowanie.** Można wybiórczo importować funkcje z wybranych modułów instrukcją **from ... import ...** lub zaimportować wszystkie funkcje z wybranego modułu: **from sys import ***

PROGRAM:

```
from sys import argv
from sys import *
```

Należy jednak unikać importowania wszystkiego z każdego modułu, gdyż powoduje to zaśmiecenie przestrzeni nazw a ponadto może dojść do konfliktu nazw pomiędzy różnymi modułami. Importujemy tylko najczęściej stosowane funkcje w celu ominięcia konieczności stosowania przedrostków wskazujących na nazwy modułów.

6.2. **Moduł liczb losowych.** Po zaimportowaniu modułu **random** można generować liczby pseudolosowe, uruchamiając generator **seed()**

PROGRAM:

```
import random
random.seed()
```

Metoda **randint(***od***,** *do***)** generuje liczbę pseudolosową z podanego zakresu liczb

Przykład 6.1. PROGRAM:

```
x=random.randint(1, 10)
y=random.randint(1, 10)
print(x)
print(y)
```

WYNIK:

```
2
5
```

W celu uniknięcia zapisu z kropką możemy zaimportować do programu najczęściej wykorzystywane funkcje, a wtedy możemy już wywoływać tą funkcję bez podawania nazwy modułu:

Przykład 6.2. PROGRAM:

```
from random import randint
x=randint(1, 10)
print(x)
```

WYNIK:

```
3
```

Uwaga 6.1. *Można też zażądać zaimportowania wszystkich nazw z modułu, np.:* **from random import *** *ale jeśli w różnych modułach importowane nazwy się powtarzają, nie da się tak zrobić!*

6.3. **Wartości losowe.** Losowy element możemy wybrać z listy, wówczas metoda **random.choice()** zwraca losowy element tej listy.

Przykład 6.3. PROGRAM:

```
import random
Lista = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
z=random.choice(Lista)
print(z)
```

Wynik:

```
F
```

Ta sama metoda może też wybrać losowy element z sekwencji.

Przykład 6.4. Program:

```
import random
Sekwencja = 'Ala ma kota!'
z=random.choice(Sekwencja)
print(z)
```

Wynik:

```
a
```

Metoda **random.uniform(a,b)** zwraca zmiennopozycyjną losową liczbę wymierną ze wskazanego zakresu [a, b):

Przykład 6.5. Program:

```
import random
print(random.uniform(1, 100))
```

Wynik:

```
86.48283719011177
```

7. Formatowanie

Do formatowania wykorzystywany jest operator % w połączeniu z ciągiem formatującym:

Przykład 7.1. Program:

```
print("%s"%10) # konwertuje każdy typ danych na łańcuch/tekst
print("%c"%33) # oznacza pojedynczy znak w kodzie ASCII
print("%i"%2.8) # konwertuje na liczbę całkowitą
print("%x"%10) # konwertuje na liczbę szesnastkową
print("%o"%10) # konwertuje na liczbę ósemkową
print("%e" %10) # konwertuje na postać naukową
print("%f"%10) # konwertuje na liczbę zmiennopozycyjną
```

Wynik:

```
10
!
2
a
12
1.000000e+01
10.000000
```

Stałą szerokość pola do wyświetlania liczb uzyskamy dzięki:
Program:

```
for x in range(1,100,10):
   print("%4i%6i%8i" % (x,x**2,x**3))
```

Wynik:

```
   1     1       1
  11   121    1331
  21   441    9261
  31   961   29791
  41  1681   68921
  51  2601  132651
  61  3721  226981
  71  5041  357911
  81  6561  531441
  91  8281  753571
```

Możemy też formatować ilość miejsc po przecinku liczb zmiennopozycyjnych:
Program:

```
print("Pierwiastkiem liczby %2i jest %5.3f" %(34, 34**0.5))
```

Wynik:

```
Pierwiastkiem liczby 34 jest 5.831
```

Znak + – wymusza wyświetlanie znaku liczby, także dla liczb dodatnich:

PROGRAM:

```
for x in range (-5,5):
    print("%+i" % x)
```

WYNIK:

```
-5
-4
-3
-2
-1
+0
+1
+2
+3
+4
```

8. Zadania

Zadanie 8.1. Napisz program Kalkulator

- Wersja prostsza: **Na wejściu:** podawany jest w jednym wierszu ciąg znaków w kolejności: 1) znak działania (+,-,*,/,itd), 2) pierwsza liczba, 3) druga liczba.
- Wersja trudniejsza: **Na wejściu:** podawany jest w jednym wierszu ciąg znaków

Na wyjściu zwracany jest wynik działania.

Zadanie 8.2. Napisz program, który dla danego wzorca P (ang. pattern) i tekstu T wypisze pozycje, na których znajduje się wzorzec P jako podsłowo tekstu T.

Na wejściu podawane są kolejno: 1) wzorzec P, 2) tekst T.

Na wyjściu zwracane są pozycje, na którch wystepuje wzorzec w tekście.

Zadanie 8.3. Napisz program który poprawia błędy w tekście.

Na wejściu podawany jest tekst oraz dwa łańcuchy znaków: zły i poprawny.

Na wyjściu wypisany zostaje tekst z zamienionymi łańcuchami ze złych na poprawne.

Zadanie 8.4. Napisz program wyznaczający pierwiastki równania kwadratowego.

$$ax^2 + bx + c = 0$$

Na wejściu podawane są wartości współczynników a, b i c.

Na wyjściu wypisane zostają wartości pierwiastów - o ile istnieją.

Zadanie 8.5. Napisz program, który wykonuje dodawanie liczb rzymskich.

Na wejściu podawane są dwie liczby rzymskie wykorzystujące litery $I = 1$, $V = 5$, $X = 10$, $L = 50$, $C = 100$, $D = 500$, $M = 1000$.

Na wyjściu zwaracana jest suma dodawania zapisana również jako liczba rzymska.

Zadanie 8.6. Wyznacz przybliżoną wartość liczby e korzystające ze wzoru:

$$e = \lim\_n \to \infty \left(1 + \frac{1}{n}\right)^n$$

Na wejściu podawana jest liczba naturalna n oznaczając ilość iteracji. Zauważ, że $n \to \infty$.

Na wyjściu zwracana jest wartość liczby e ($\cong 2.718...$)

Zadanie 8.7. Napisz program, który dla zadanej wartości $n \in \mathbb{N}$ wyznacza wartość $n!$ (silnia):

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \cdots \cdot (n-1) \cdot n$$

Pamiętaj, że $0! = 1$.

Na wejściu podawana jest liczba naturalna n.

Na wyjściu zwracana jest wartość silnia z liczby n.

Zadanie 8.8. Uczniowie podczas semestru otrzymują za aktywność na lekcji plusy, a za brak pracy domowej minusy. Na koniec semestru nauczyciel zlicza wszystkie plusy i minusy, przy czym jeden plus redukuje jeden minus i odwrotnie. Po ewentualnej redukcji, za każde trzy plusy, uczeń otrzymuje piątkę, a za każde trzy minusy - dwójkę. Jeśli po redukcji pozostaną mniej niż trzy plusy lub trzy minusy, uczeń nie otrzymuje żadnej oceny. Napisz program, który ułatwi pracę nauczycielowi i na podstawie ciągu zgromadzonych plusów i minusów wypisze, jakie oceny należy wystawić uczniom.

Na wejściu w pierwszym i jedynym wierszu znajduje się ciąg składający się ze znaków + lub -. Długość ciągu nie przekracza 100 znaków.

Na wyjściu należy wypisać oceny (rozdzielone znakiem spacji) jakie uczeń otrzyma po wymianie plusów i minusów. Jeśli uczeń nie otrzyma żadnej oceny, należy wypisać słowo BRAK.

Przykład 1: Dla danych wejściowych:

```
+++-+++--+++++-+++++
```

poprawna odpowiedź to:

```
5 5 5
```

Przykład 2: Dla danych wejściowych:

```
---+-+----
```

poprawna odpowiedź to:

```
2
```

Przykład 3: Dla danych wejściowych:

```
+-++---++-+--+++-+
```

poprawna odpowiedź to:

```
BRAK
```