Wydział: **Zarządzania i Modelowania Komputerowego**
Katedra Technologii Informatycznych
Przedmiot: **Technologie informacyjne**
Rok **I**

PYTHON - ĆWICZENIE 3

Pętle to konstrukcje języka programowania, które umożliwiają zdefiniowanie grupy instrukcji, które będą wielokrotnie powtarzane (cykliczne wykonywane określoną liczbę razy, do momentu zajścia pewnych warunków).

1. Instrukcja pętli *for*

W przypadku tej pętli powinno być z góry wiadome, ile razy instrukcje z zakresu pętli mają się wykonać. Przy czym "z góry" oznacza, że w momencie, gdy wykonywanie programu dotrze do tej instrukcji. Składnia instrukcji **for** jest następująca:

for *licznik* in *sekwencja*:
instrukcja1
instrukcja2
…

Iterowanie odbywa się po elementach dowolnej sekwencji (uporządkowanego zbioru elementów – najczęściej listy), w kolejności, w jakiej te elementy występują w sekwencji.

W przypadku, gdy iteracja przebiega po sekwencji liczb, przydatna jest funkcja wbudowana

range([start,] stop [, krok])

gdzie parametry **start** i **krok** są opcjonalne; pominięcie **start** jest równoznaczne z przyjęciem wartości startowej **0**, pominięcie parametru **krok** oznacza przyjęcie kroku (odległości między elementami sekwencji) równego **1**; parametr **stop** jest ograniczeniem sekwencji, już do niej **nie należy**.

Zauważmy jednak, co się dzieje, gdy próbujemy wyprowadzić polecenie:

```
>>> print(range(10))
range(0, 10)
>>>
```

Pod wieloma względami obiekt zwracany przez **range** zachowuje się tak, jakby był listą, ale w rzeczywistości nią nie jest. Jest to obiekt, który zwraca kolejne elementy żądanej sekwencji podczas iteracji po nim, ale tak naprawdę nie tworzy listy, oszczędzając w ten sposób miejsce.

Pojęcie **lista** zostanie wprowadzone później, ale gdyby zachodziła potrzeba sprawdzenia, jakie elementy znajdą się w sekwencji podczas iteracji, można wykorzystać :

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Zadanie

Sprawdź, jak przy użyciu **range** uzyskać sekwencję złożoną z liczb całkowitych:

- ciąg arytmetyczny - wartość początkowa **0**, końcowa **9**, krok**1**
- ciąg arytmetyczny – wartość początkowa **1**, końcowa **9** krok **1**
- ciąg arytmetyczny – wartość początkowa **1**, końcowa **9**, krok **2**
- ciąg arytmetyczny – wartość początkowa **9**, końcowa **1**, krok **-2**

```
>>> print(range(10))
range(0, 10)
>>> print(list(range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print(range(1,10))
range(1, 10)
>>> print(list(range(1,10)))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print(range(1,10,2))
range(1, 10, 2)
>>> print(list(range(1,10,2)))
[1, 3, 5, 7, 9]
>>> range(9, 0, -2)
range(9, 0, -2)
>>> print(list(range(9,0,-2)))
[9, 7, 5, 3, 1]
>>>
```

Przykład 1

Zapisz instrukcję, która wydrukuje kwadraty liczb od 0 do 10. W trybie interaktywnym po każdej instrukcji należy wcisnąć <Enter>

```
>>> for x in range(11):
        print (x,x**2)


0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
>>>
```

W celu wyrównania wyprowadzonych tabelarycznie wyników można uzupełnić instrukcję o formatowanie.

```
>>> for x in range(11):
        print('%4i  %5i' % (x, x**2))


   0      0
   1      1
   2      4
   3      9
   4     16
   5     25
   6     36
   7     49
   8     64
   9     81
  10    100
>>>
```

Przykład 2

Napisz i uruchom skrypt, który wylicza wartość n! dla n=0, 1, 2, … 10. Wykorzystaj definicję:

0!=1
n!=(n-1)!*n

```
File  Edit  Format  Run  Options  Window  Help
print('n! dla n=0,1,2,... 10')
s=1
print(' 0! =%8i'% s)
for n in range(1,11):
    s=s*n
    print('%2i! =%8i'%(n,s))
```

Po uruchomieniu pojawią się poniższe wyniki:

```
====================== RESTART:
n! dla n=0,1,2,... 10
 0! =       1
 1! =       1
 2! =       2
 3! =       6
 4! =      24
 5! =     120
 6! =     720
 7! =    5040
 8! =   40320
 9! =  362880
10! = 3628800
>>>
```

Zadania dodatkowe

Napisz, uruchom i przetestuj skrypty:

- **pierwiastki.py**, który przedstawi pierwiastki kwadratowe oraz pierwiastki trzeciego stopnia z liczb od 0 do 100 z krokiem 5
- **potegi.py**, który pokaże kolejne potęgi liczby x (x^n); wykładnik n zmienia się od 0 do 20 z krokiem 2.
- **srednia.py**, który wyznaczy średnią arytmetyczną n kolejno wprowadzanych liczb.
- **sred_plus.py**, który zmodyfikuje skrypt z poprzedniego zadania tak, by wyznaczana była średnia tylko dodatnich, spośród wprowadzonych liczb.
- **pomiary.py**, który wyznacza (w %) „wadliwość" badanej partii wyrobów (w wyniku statystycznej kontroli jakości wyrobów dokonano **n** pomiarów cechy **x**; wartość nominalna cechy wynosi **d**, a tolerancja jest równa **h**).

2. Instrukcja pętli while

while *warunek*:
    instrukcja1
    instrukcja2
    …

Pętla tego typu powtarza blok instrukcji wiele razy, aż ***warunek*** zostanie oceniony jako **False**; ***warunek*** jest podawany przed blokiem instrukcji i jest testowany przed każdym wykonaniem zakresu pętli. (innymi słowy blok instrukcji jest wykonywany wtedy, gdy ***warunek*** jest prawdziwy, po jego wykonaniu ***warunek*** sprawdzany jest po raz kolejny) Powtarzany blok instrukcji zapisywany jest z wcięciem (tabulatorem).

Zazwyczaj pętla **while** jest używana, gdy niemożliwe jest ustalenie dokładnej liczby iteracji z góry, ale nie tylko wtedy.

Przykład 3

Wczytaj liczbę całkowitą i podaj, przy użyciu ilu cyfr została ta liczba zapisana.

```
File Edit Format Run Options Window Help
print('sprawdzanie ile cyfr ma wczytana liczba całkowita')
n = int(input('dana liczba n='))
ile = 0
while n > 0:
    n //= 10
    ile += 1
print(ile)
```

Przy każdej iteracji odcinana jest ostatnia cyfra, przy zastosowaniu dzielenia całkowitego przez 10, Zmienna **ile** zapamiętuje ile razy to było wykonane.

```
========== RESTART: F:\PYTHON\ile_cyfr.py =
sprawdzanie ile cyfr ma wczytana liczba całkowita
dana liczba n=4376193
7
>>>
```

Istnieje inny, łatwiejszy sposób rozwiązania tego problemu, dzięki operacjom na napisach (string). Wystarczy wykorzystać funkcję **len,** która zwraca liczbę znaków w łańcuchu znaków**.**

Przykład 4

Zliczanie, ile spośród liczb podawanych przez użytkownika jest parzystych. Zakończenie obliczeń nastąpi po wprowadzeniu liczby niedodatniej.

```
File Edit Format Run Options Window Help
m=0
a=int(input('podaj liczbę a='))
while a>0:
    if a%2==0:
        m=m+1
    a=int(input('podaj liczbę a='))
if m==0:
    print('brak liczb parzystych')
else:
    print('iczb parzystych było : ',m )
```

Przykład 5

Wyznaczanie kolejnych elementów ciągu Fibonacciego, mniejszych od zadanej liczby . Ciąg Fibonacciego to ciąg, którego dwa początkowe elementy to 0 i 1, a każdy kolejny wyraz jest sumą dwóch poprzednich.

```
File Edit Format Run Options Window Help
print('generowanie ciągu Fibonacciego o elementach mniejszych od n')
n=int(input('n='))
a=0
b=1
while a<n:
    print(a, end=' ')
    c=a+b
    a=b
    b=c
print()
```

```
========== RESTART: F:\PYTHON\fibo.py ======
generowanie ciągu Fibonacciego o elementach mniejszych od n
n=2000
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
>>>
```

Powyższy algorytm można zapisać krócej, wykorzystując instrukcję przypisania wielokrotnego:

zmienna_1, zmienna_2,...,zmienna_n=wartość_1, wartość_2,...,wartość_n

równoważnej ciągowi instrukcji:

zmienna_1 =wartość_1
zmienna_2=wartość_2
…
zmienna_n=wartość_n

przy czym najpierw obliczane są wszystkie wartości po prawej stronie znaku = a dopiero wtedy ma miejsce przypisanie.

```
File  Edit  Format  Run  Options  Window  Help
print('generowanie ciągu Fibonacciego o elementach mniejszych od n')
n=int(input('n='))
a,b=0,1
while a<n:
    print(a, end=' ')
    a,b=b,a+b
print()
```

Zapis: a,b=b, a+b

oznacza, że najpierw obliczana jest wartość a+b, później modyfikowane b; przy zastosowaniu „zwykłych" instrukcji przypisania konieczne było wprowadzenie dodatkowej zmiennej c.

```
========================== RESTART: F:\PYTHON\fibo1.py ======
generowanie ciągu Fibonacciego o elementach mniejszych od n
n=2000
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
>>>
```

Wynik obliczeń jest taki sam jak przy zastosowaniu pierwszego algorytmu.

Przykład 6

„Gra" podobna do Black Jacka: program odczytuje liczby i sumuje je do momentu, w którym suma stanie się większa lub równa 21. Gdyby na wejściu pojawiło się 0, program zatrzymuje działanie nawet, jeśli suma wczytanych liczb jest równa mniejsza niż 21.

Przetestuj wprowadzony skrypt i zauważ, że człon **else** jest realizowany tylko wtedy, gdy działanie pętli **while** zakończone jest „normalnie", czyli gdy ***warunek*** po **while** okazał się mieć wartość **False**. Gdy działanie pętli zostało przerwane przez **break**, człon **else** jest pomijany.

```
File  Edit  Format  Run  Options  Window  Help
suma = 0
a = int(input())
while a != 0:
    suma += a
    if suma >= 21:
        print('Końcowa suma jest równa ', suma)
        break
    a = int(input())
else:
    print('Końcowa suma <= 21 i jest równa', suma, '.')
```

Zadania dodatkowe

Napisz, uruchom i przetestuj skrypty:

- **licz_pplus.py**, który wyznacza iloczyn kolejnych liczb dodatnich parzystych;. obliczanie iloczynu zakończyć, gdy iloczyn ten przekroczy wartość 1000
- **sred_n_uj.py,** który wprowadza liczby do momentu napotkania liczby ujemnej i wyznacza średnią wprowadzonych liczb nieujemnych.

- **szereg.py**, który oblicza wartość $\mathbf{e^x}$, korzystając z rozwinięcia w szereg potęgowy z dokładnością **eps**:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

i porównuje uzyskaną wartość z math.exp(x).

- **pierw_kw.py,** który dla zadanej liczby a>0 wyznacza przybliżoną wartość jej pierwiastka kwadratowego, zgodnie ze wzorem:

$$x_{n+1} = 0.5\left(x_n + \frac{a}{x_n}\right) \qquad n=0,1,2\ldots$$

obliczenia zakończ, gdy:

$$|x_{n+1} - x_n| < \varepsilon$$

liczba ε oznacza dokładność obliczeń; przyjmij $x_0=1$

- **równ_iter.py,** który rozwiązuje metodą iteracji prostej równanie:

$$x - \cos(x) = 0$$

dla znanych wartości przybliżenia początkowego x_0 (dowolna liczba) oraz dokładności obliczeń ε; kolejne przybliżenia rozwiązania należy wyznaczać z zależności:

$$x_{n+1} = \cos(x_n)$$