Temat 3

**Zadanie 1.** Silnia

**while True**:  
 given\_number = int(input(**"Podaj liczbę: "**))  
 **if** given\_number == 0 **or** given\_number == 1:  
 print(1)  
 **break  
 else**:  
 factorial = 1;  
 **for** n **in** range(1, given\_number + 1):  
 factorial \*= n  
 print(factorial)

Tworzę pętlę **while**, która ma działać dopóki jej nie przerwę. Wykorzystuje fakt, że pętla działa dopóki wartość wyrażenia jest prawdziwa, a prawda zawsze jest prawdziwa. (xD)

Definiuje zmienną **given\_number** i wczytuje jej wartość z klawiatury.

Jeżeli podana liczba to 0 lub 1 wypisuje wynik równy 1 i przerywam główną pętlę tym samym kończąc działanie programu.

W innym przypadku:

Definiuję zmienną **factorial**. Będziemy w niej przechowywać iloczyn **n** kolejnych liczb.

Tworzę pętlę **for**, **n** przybiera wartości całkowite z zakresu **[1,given\_number+1)**, czyli **1,2,…,given\_number**.

W każdym obiegu pętli, mnożę wynik razy **n**.

**factorial** jest silnią podanego numeru. Wypisuje.

**Zadanie 2.** Tabliczka Mnożenia

*Na początku parę przydatnych faktów:*

***print(x1,x2,…,xn)*** *wypisuje stringa zakończonego znakiem końca linii (znak biały, bez reprezentacji graficznej), dzięki któremy komputer „wie”, że kolejne znaki ma wypisać w następnej linijce. W tym zadaniu jest to niepożądane.*

***print(x1,x2,…,xn, end=””)*** *wypisuje stringa bez wstawiania jakiegokolwiek dodatkowego znaku. Argument end=”’ pozwala określić co znajdzie się na końcu linii.*

***print(„{:8d}.format(integer))*** *wyrównuje zmienną całkowitą do prawej, ostatnia cyfra jest na 8 pozycji od ostatniego tekstu.*

***print(„{:xd}.format(integer))*** *wyrównuje zmienną całkowitą do prawej, ostatnia cyfra jest na x pozycji od ostatniego tekstu.*

print("Liczba na dziewięciu pozycjach:{:9d}".format(5) ) wypisze:

Liczba na dziewięciu pozycjach: 5

*Zwróć uwagę, że „5” jest dziewiątym znakiem po „:”. W przykładzie na upelu mylące jest to, że po dwukropku jest wpisana spacja, przez co wydaje się, że 5 jest na dziesiątej pozycji.*

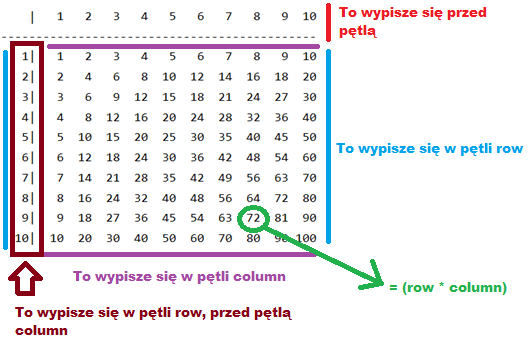
print("Liczba na dziewięciu pozycjach:{:9d}".format(53) ) wypisze:

Liczba na dziewięciu pozycjach: 53

„3” jest na 9 pozycji.

Przejdźmy do kodu:

print(**"{:4s}"**.format(**" |"**), end=**''**)  
**for** x **in** range(1, 11):  
 print(**"{:4d}"**.format(x), end=**''**)  
print(**""**)  
print(**"---------------------------------------------"**)  
**for** row **in** range(1, 11):  
 print(**"{:4d}"**.format(row),**'|'**,sep=**''**, end=**''**)  
  
 **for** column **in** range(1, 11):  
 print(**"{:4d}"**.format(column \* row), end=**''**)  
 print(**""**)



Na początku wypisuje pierwsze puste pole tabeli (lewy górny róg)

Następnie tworzę pętlę **for**, wypisującą kolejne liczby **od 1 do 10**, każda z wyrównana do czwartej pozycji względem poprzedniego znaku.

Wypisuje „pustego” stringa. De facto służy to tylko przejściu do kolejnej linii, print dodaje znak końca linii do pustego stringa.

Wypisuje kreskę z myślników (xD). Kończy się ona znakiem końca linii.

Tworzę pętlę **for** ze zmienną **row**, przyjmującą wartości od **[1;10]**. Są to numery kolejnych wierszy.

Wypisuje numer wiersza i **„|”**, nierozdzielone żadnym znakiem **(sep=””)**. Używam **(end=””)** aby pozbyć się znaku końca linii.

Tworzę pętlę zagnieżdżoną **for**, zmienna **column** określa kolejne kolumny w wierszu **row**. Wypisuje iloczyn wiersza i kolumny, wyrównany do czwartej pozycji od ostatniego znaku, bez końca linii.

Po wypisaniu całego wiersza wypisuje znak końca linii (tak jak na początku, **print(„”)**) i rozpoczynam wypisywanie kolejnego wiersza.

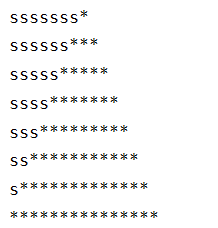
Po wypisaniu całości tabeli program wychodzi z pętli, kończąc działanie.

**Zadanie 3.** Choinka

*Na potrzeby tłumaczenia spacje zastąpiłem literą s. Niżej znajduje się wersja poprawna.*

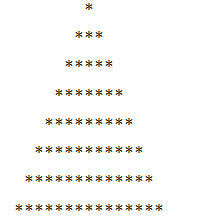
*Zauważmy, że:*

*W pierwszym rzędzie jest 1 gwiazdka, w każdym kolejnym o 2 więcej (kolejne liczby nieparzyste), zatem ilość gwiazdek w linijce to 2n - 1, gdzie n to numer rzędu.*

*W pierwszym rzędzie gwiazdka jest na pozycji której numer jest równy ilości rzędów choinki. W każdym kolejnym jest o 1 znak bliżej lewej strony.*

*Zatem w każdym rzędzie przed gwiazdkami pojawia się liczba spacji wyrażona: ilość\_rzędów – numer\_aktualnego\_rzędu*

Przejdźmy do kodu:

rows=int(input(**"Podaj liczbe rzędów: "**))  
**for** current\_row **in** range(1, rows + 1):  
 **for** \_ **in** range(rows - current\_row):  
 print(**" "**, end=**""**)  
 **for** \_ **in** range(2 \* current\_row - 1):  
 print(**"\*"**, end=**""**)  
 print(**""**)

Na początku definiuje zmienną **rows** i przypisuje jej wartość podaną przez użytkownika. Jest to liczba rzędów („pięter” choinki).

Tworzę pętlę **for**, **current\_row** oznacza numer rzędu i przyjmuje wartości całkowite z zakresu **[1;rows+1)** czyli **1,2,…,rows**.

Tworzę pętlę **for** wypisującą spacje. Zgodnie z początkowym spostrzeżeniem funkcja   
**print(„ ”, end=””)** powinna wykonać się **rows – current\_row** razy.

Po wypisaniu spacji czas na gwiazdki.

Tworzę pętlę **for** wypisującą gwiazdki. Zgodnie z początkowym spostrzeżeniem funkcja   
**print(„\*”, end=””)** powinna wykonać się **2 \* current\_row – 1 razy**.

Po wypisaniu spacji i gwiazdek przechodzę (**print(„”)**) do następnej linijki. Po wypisaniu **rows** rzędów program kończy działanie.

**Zadanie 4.** 100 zł

*Zauważmy następujące fakty:*

* ***Dwuzłotówek*** *może być minimalnie* ***0****, a maksymalnie* ***50*** *(bo gdyby było więcej to sama ich wartosć przekroczy 100zł).*
* ***Pięciozłotówek*** *może być minimalnie* ***0****, a maksymalnie* ***20***
* ***Dziesięciozłowtówek*** *może być minimalnie* ***0****, a maksymalnie* ***10***

*Zgodnie z poleceniem musimy sprawdzić wszystkie kombinacje, czyli 11\*21\*51. Nie jest to rozwiązanie optymalne. Już przy 100 zł i 3 rodzajach nominałów jest 11781 operacji do wykonania. Przy 10000zł i tych samych nominałach operacji jest 10,017,008,001.*

Przejdźmy do kodu:

found\_solutions=0  
**for** num\_of\_tens **in** range(11):  
 **for** num\_of\_fives **in** range(21):  
 **for** num\_of\_twos **in** range(51):  
 **if** num\_of\_twos \* 2 + num\_of\_fives \* 5 + num\_of\_tens \* 10 == 100:  
 found\_solutions+=1  
print(found\_solutions)

Definiuje zmienną **found\_solutions**, jest to liczba znalezionych rozwiązań. Na razie oczywiście jest równa **0**.

Tworzę trzy pętle **for**:

1. Pierwsza iteruje po ilości dziesięciozłotówek, **num\_of\_tens** zwiększa się o 1 gdy pętla num\_of\_fives wykona się 21 razy **[0;20]** (wykonuje się tyle razy w każdym obiegu pierwszej pętli).   
   *Pętla* ***num\_of\_twos*** *wykonuje się 21\*51 razy w jednym obiegu pierwszej pętli.*
2. Druga iteruje po ilości pięciozłotówek, num\_of\_fives zwiększa się o 1 gdy pętla **num\_of\_twos** wykona się 51 razy **[0;50]**
3. Trzecia iteruje po ilości dwuzłotówek i sprawdzamy w niej warunek: wartość monet powinna być równa **100** (dlatego mnożymy ilości monet przez ich wartość).  
   Jeśli dla danej kombinacji wartość wynosi **100** do zmiennej **found\_solutions** dodajemy **1**, kolejne rozwiązanie zostało znalezione. W innym przypadku nie robimy nic.  
   *Ta część kodu wykona się 11\*21\*51 razy, przyjmując wszystkie możliwe kombinacje.*

**Zadanie 5.** Liczby pierwsze

*Oczekiwany algorytm nazywa się Sitem Eratostenesa, jest dobrze opisany na angielskiej wiki i np. tu*

[*https://eduinf.waw.pl/inf/alg/001\_search/0011.php*](https://eduinf.waw.pl/inf/alg/001_search/0011.php)

*Idea jest taka:*

1. *Żeby znaleźć liczby pierwsze z zakresu [2;a] tworzymy tablicę tych liczb.*
2. *Korzystamy z zasady, że liczba pierwsza dzieli się tylko przez samą siebie i przez 1. Zatem nie może być wielokrotnością żadnej z poprzednich liczb.*
3. *Bierzemy pierwszą liczbę (2) i „wykreślamy” jej wszystkie wielokrotności (oczywiście bez niej samej).*
4. *Bierzemy kolejną, nieskreśloną liczbę i „wykreślamy” jej wszystkie wielokrotności.*
5. *Po wykonaniu tego dla wszystkich liczb, w tablicy pozostaną same liczby pierwsze*.

[*https://upload.wikimedia.org/wikipedia/commons/b/b9/Sieve\_of\_Eratosthenes\_animation.gif*](https://upload.wikimedia.org/wikipedia/commons/b/b9/Sieve_of_Eratosthenes_animation.gif)

*Dobrze obrazuje to ten gif.*

Algorytm jest dość wolny (złożoność czasowa wersji nieoptymalizowanej O(n2), złożoność pamięciowa O(n)). Optymalizując go możemy skorzystać z faktu, że dla liczby n wystarczy sprawdzić czy dzieli się przez liczby mniejsze bądź równe , aby ocenić czy jest pierwsza. Większe dzielniki są czynnikami sprawdzonych już wcześniej działań.

Implementując algorytm możemy stworzyć listę liczb i usuwać elementy z listy (jako operacja wykreślania), ale jest to mocno niezalecane. W innych językach (np. C++) operacja usunięcia z listy wymaga „przesunięcia” dalszych elementów. Zastosowanie tej techniki zwiększa złożoność czasową do O(n3).

Lepszym pomysłem jest utworzenie tablicy przechowującej wartości typu bool wielkości n+1 zawierającej początkowo same wartości **True**. Liczba oznaczona True jest aktywna/nieskreślona. Odwoływać do niej będziemy się przez indeks *(****tablica[liczba]*** *zwraca informacje czy liczba jest skreślona).*

Początkowo tabela wygląda tak. Nie przejmujemy się tab[0] i tab[1], są w pamięci ale program w żaden sposób ich nie wykorzystuje. Rozmiar tabeli to n+1.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … | n-1 | n |
| Wartość: | True | True | True | True | True | True | True | True | True | True | True | … | True | True |

Przejdźmy do kodu:

**import** math  
  
**def** sieve(number):  
 prime\_list = [**True for** \_ **in** range(number + 1)]  
 **for** prime **in** range(2, int(math.sqrt(number))):  
 **if** prime\_list[prime]:  
 **for** to\_check **in** range(prime + 1, number + 1):  
 **if** to\_check % prime == 0:  
 prime\_list[to\_check] = **False  
  
 for** n **in** range(2, number + 1):  
 **if** prime\_list[n]:  
 print(n, end=**" "**)

Importuje moduł **math**.

Tworzę funkcję **sieve** przyjmującą jako argument liczbę, będziemy szukali liczb pierwszych w zakresie **[2;number].**

Tworzę listę **prime\_list**, wypełnioną wartościami **True**, o długości **number+1** *(aby można było się odwołać do* ***prime\_list[number]****)*

Tworzę pętlę **for**, **prime** jest kolejnymi liczbami z zakresu **[2, number].**

*Uwaga! Wzięcie zakresu od 0 lub 1 spowoduje wykreślenie wszystkich liczb.*

Jeśli liczba jest wykreślona (**prime\_list[prime] == False**) nic się nie dzieje w danym obiegu pętli

Jeśli liczba nie jest wykreślona **(prime\_list[prime] == True**) przechodzę do **for**’a wykreślającego jej wielokrotności.

*Sprawdzanie tego warunku nie jest obowiązkowe, sito zadziała i bez niego, ale wykona zbędne operacje.*

W wewnętrznej pętli **for**, zmienna **to\_check** przyjmuje wartości z zakresu **[prime + 1, number]**. Wpisanie samego **prime** spowoduje wykreślenie liczby pierwszej **prime** z tablicy, bo na pewno dzieli się przez samą siebie.

Sprawdzamy, czy sprawdzana liczba dzieli się przez liczbę **prime**. Jeśli tak, oznacza to, że jest jej wielokrotnością, więc wykreślamy ją z tablicy (**prime\_list[to\_check] = False**)

Po wykonaniu się pętli **prime**, w tablicy **prime\_list** tylko liczby pierwsze są oznaczone **True** *(oraz 0 i 1, ale je pomijamy również dalej)*

Tworzę pętlę **for**, **n** przyjmuje wartości całkowite z zakresu **[2;number]**. Dla każdego **n** sprawdzamy czy **prime\_list[n] == True**. Jeśli tak, wypisujemy **n**. W innym przypadku nic się nie dzieje.

Wynikiem działania funkcji jest wypisanie liczb pierwszych z zakresu **[2;number]**

Alternatywa:

W folderze zamieszczam też algorytm oparty na **teście Millera-Rabina**. Jest to algorytm probabilistyczny- wykorzystujący losowość, zwraca informację, że liczba jest na pewno złożona lub prawdopodobnie pierwsza. Przy ilości testów dążącej do nieskończoności, prawdopodobieństwo błędu spada do 0. W praktyce niewielka liczba testów daje poprawny wynik.

Algorytm korzysta z Małego Twierdzenia Fermata i potęgowania modularnego. Warto to ogarnąć przed próbą zrozumienia go. Jest dobrze opisany w książce „*Wprowadzenie do algorytmów*”.

Zaletą jest jego złożoność czasowa- w przypadku sprawdzania wszystkich liczb z zakresu **[2;n]** wynosi

O(n \* log3n), zakładając z góry ustaloną liczbę testów każdej liczby. Wykorzystując Szybką Transformatę Fouriera redukujemy złożoność do O(n\* log2n \* log log n \* log log log n).

Największą przewagą nad **Sitem Erastotenesa** jest wykorzystanie pamięci. Sito musi korzystać z tablicy o n długości (wersja bez optymalizacji), zaś **test Millera-Rabina** operuje na samej sprawdzanej zmiennej.

Polecam przetestować obie implementacje dla liczb 100000 i 1000000.