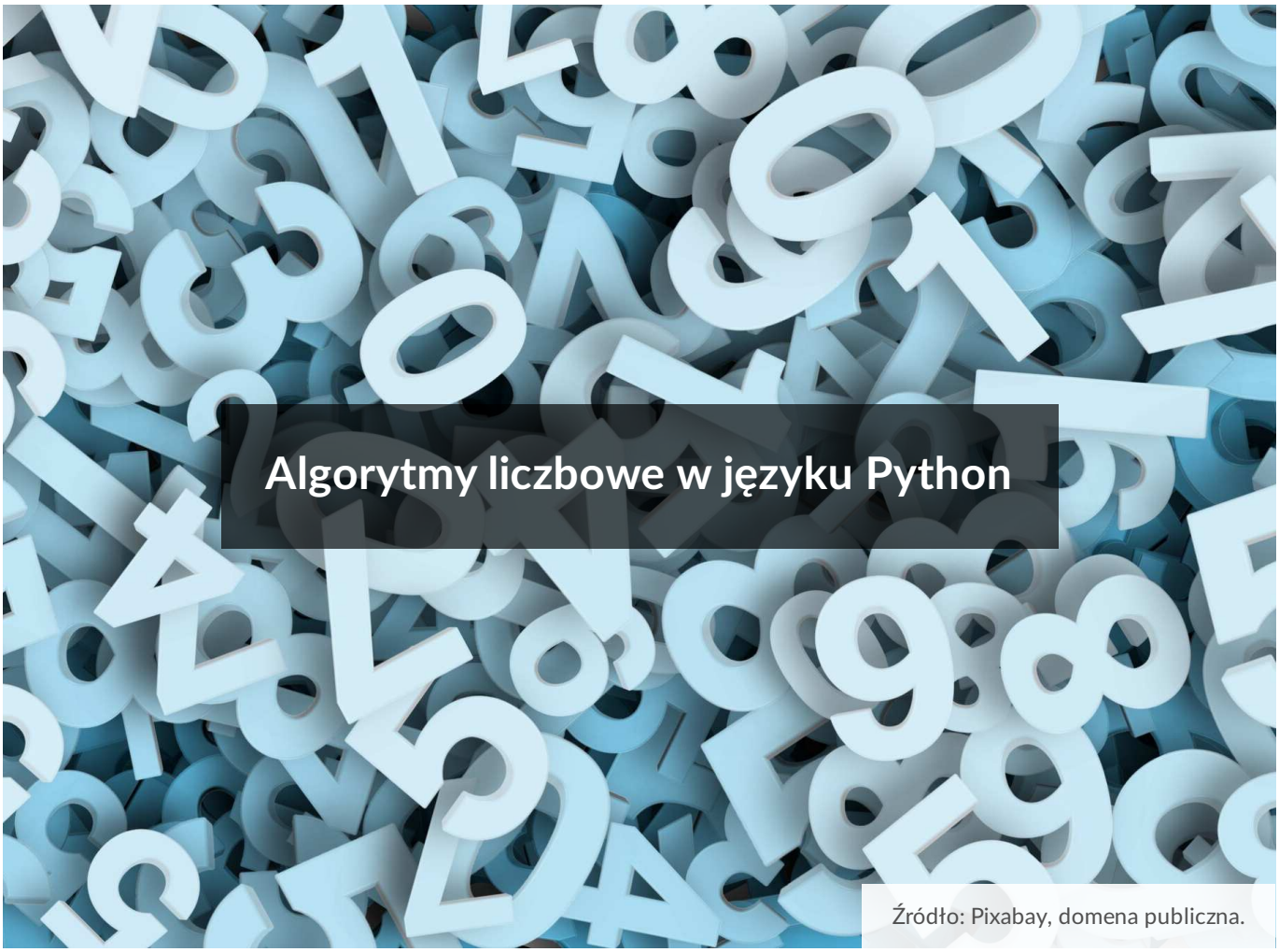
Algorytmy liczbowe w języku Python



Źródło: Pixabay, domena publiczna.

Liczby zaprzyjaźnione, bliźniacze, doskonałe oraz pierwsze to liczby spełniające określone warunki. To, czy dana liczba należy do wybranego zbioru, weryfikujemy za pomocą odpowiedniego algorytmu liczbowego. W e-materiale Algorytmy liczbowe poznaliśmy podstawowe informacje na ten temat.

Teraz natomiast dowiemy się, jak powinna wyglądać implementacja wybranych algorytmów liczbowych w języku Python.

Ciekawi cię, jak wyglądają implementacje w innych językach programowania? Znajdziesz je w e-materiałach:

- Algorytmy liczbowe w języku C++,
- Algorytmy liczbowe w języku Java.

Więcej zadań? Przejdź do Algorytmy liczbowe – zadania maturalne.

> **Twoje cele**
>
> - Przedstawisz algorytm szukania liczb bliźniaczych wśród liczb pierwszych.
> - Scharakteryzujesz sposób na otrzymanie liczby doskonałej.
> - Sprawdzisz, czy dwie liczby są zaprzyjaźnione.
> - Użyjesz różnych struktur danych w języku Python.

Przeczytaj

Liczby pierwsze i bliźniacze

Definicja: Liczby pierwsze bliźniacze

Liczby pierwsze bliźniacze to kolejne **liczby pierwsze**, których różnica wynosi 2.

Przykładem takich liczb są pary: 71 i 73 lub 137 i 139.

Przykład 1

Przygotujmy kod, który – wśród liczb pierwszych – pozwoli odnaleźć liczby bliźniacze. W tym przypadku przygotujemy funkcję obliczającą liczby pierwsze w przedziale $\langle 2, n \rangle$ metodą Eratostenesa (więcej na jej temat znajdziesz w e-materiale Sito Eratostenesa). Następnie zbadamy, które spośród znalezionych liczb pierwszych spełniają warunki liczb bliźniaczych. Działanie naszego programu sprawdzimy, szukając liczb bliźniaczych do liczby `80`.

Specyfikacja:

Dane:

- `n` – górna granica przedziału; liczba naturalna, większa od 2

Wynik:

Program ma znaleźć i wypisać pary liczb bliźniaczych w zadanym przedziale.

Zaczynamy od zaimplementowania funkcji wyszukującej wszystkie liczby pierwsze w przedziale $\langle 2, n \rangle$. Tworzymy definicję naszej funkcji. Następnie przygotowujemy dwie listy: `lista_liczb`, która będzie przechowywała znalezione liczby pierwsze, oraz `lista`, która będzie służyła do przechowywania wykreślonych liczb (wartość `True` oznacza, że liczba nie została jeszcze wykreślona).

```
1 def sito_eratostenesa(n):
2     lista_liczb = []
3     lista = [True] * (n + 1)
```

Następnie tworzymy pętlę, której zadaniem będzie znajdowanie najmniejszej niewykreślonej liczby w danym momencie. Będzie ona działała w przedziale $\langle 2, \sqrt{n} \rangle$. W celu obliczenia pierwiastka, importujemy z biblioteki `math` funkcję `sqrt`. Jeżeli

obecnie sprawdzana liczba nie była jeszcze wykreślona, to zaczynamy wykreślanie wszystkich jej wielokrotności z tablicy `lista`. W tym celu tworzymy pętlę odpowiadającą za wyznaczanie kolejnych wielokrotności liczby `indeks`. Swoje działanie będzie zaczynała od jej dwukrotności i zwiększała wartość o wartość sprawdzanej liczby.

```
from math import sqrt

def sito_eratostenesa(n):
    lista_liczb = []
    lista = [True] * (n + 1)

    for indeks in range(2, int(sqrt(n)) + 1):
        if lista[indeks]:
            for indeks_2 in range(2 * indeks, n + 1, indeks):
                lista[indeks_2] = False
```

Kolejnym krokiem będzie odszukanie liczb pierwszych, czyli tych, które nie zostały wykreślone z listy `lista`. W tym celu tworzymy pętlę, która zapisuje te liczby do listy `lista_liczb`. Na końcu zwracamy nasz wynik, czyli listę liczb pierwszych znalezionych w podanym przedziale.

```
from math import sqrt

def sito_eratostenesa(n):
    lista_liczb = []
    lista = [True] * (n + 1)

    for indeks in range(2, int(sqrt(n)) + 1):
        if lista[indeks]:
            for indeks_2 in range(2 * indeks, n + 1, indeks):
                lista[indeks_2] = False

    for element in range(2, n + 1):
        if lista[element]:
            lista_liczb.append(element)

    return lista_liczb
```

Przejdźmy teraz do wyszukiwania liczb bliźniaczych. W tym celu, za pomocą zaimplementowanej funkcji, wyszukamy najpierw wszystkie liczby pierwsze

z przedziału ⟨ 2 , n ⟩.

```
liczby_pierwsze = sito_eratostenesa(80)
print('Liczby pierwsze:', liczby_pierwsze)
# Wypisze:
# Liczby pierwsze: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
```

Tworzymy listę `liczby_blizniacze`, która będzie przechowywała znalezione pary liczb bliźniaczych. Następnie, za pomocą pętli, sprawdzamy każde dwie kolejne znalezione liczby pierwsze z tablicy `liczby_pierwsze`.

```
liczby_pierwsze = sito_eratostenesa(80)
print('Liczby pierwsze:', liczby_pierwsze)

liczby_blizniacze = []
for indeks in range(1, len(liczby_pierwsze)):
    x0 = liczby_pierwsze[indeks - 1]
    x1 = liczby_pierwsze[indeks]
    if x1 - x0 == 2:
        liczby_blizniacze.append((x0, x1))

print('Liczby blizniacze:', liczby_blizniacze)
# Wypisze:
# Liczby blizniacze: [(3, 5), (5, 7), (11, 13), (17, 19), (29,
```

Oto kod całego programu:

```
from math import sqrt

# Definicja funkcji
def sito_eratostenesa(n):
    lista_liczb = []
    lista = [True] * (n + 1)

    for indeks in range(2, int(sqrt(n)) + 1):
        if lista[indeks]:
            for indeks_2 in range(2 * indeks, n + 1, indeks):
                lista[indeks_2] = False

```

```
    for element in range(2, n + 1):
        if lista[element]:
            lista_liczb.append(element)

    return lista_liczb


# Wykonanie funkcji
liczby_pierwsze = sito_eratostenesa(80)

print('Liczby pierwsze:', liczby_pierwsze)

liczby_blizniacze = []
for indeks in range(1, len(liczby_pierwsze)):
    x0 = liczby_pierwsze[indeks - 1]
    x1 = liczby_pierwsze[indeks]
    if x1 - x0 == 2:
        liczby_blizniacze.append((x0, x1))

print('Liczby blizniacze:', liczby_blizniacze)
```

Wynik wywołania kodu będzie następujący:

```
Liczby pierwsze: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 4
Liczby blizniacze: [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31
```

Liczby doskonałe

Definicja: Liczby doskonałe

Liczby doskonałe to liczby naturalne, które są jednocześnie sumą wszystkich swoich dzielników właściwych.

Przykładem takiej liczby jest 28, ponieważ spełnia ona przedstawiony warunek:
$28 = 14 + 7 + 4 + 2 + 1$.

Ciekawostka

Jedną z możliwości odnajdywania liczb doskonałych jest sposób opisany przez Euklidesa w IX księdze *Elementów*.

W celu znalezienia liczby doskonałej należy obliczyć sumy kolejnych potęg liczby 2, a jeśli taka liczba okaże się liczbą pierwszą, wówczas trzeba pomnożyć ją przez ostatni składnik

dodawania – w ten sposób otrzymamy liczbę doskonałą.

Przykład:

Badamy kolejno wyrazy:

- $1 + 2^1 = 3$ - jest liczbą pierwszą, czyli $3 \cdot 2^1 = 6$ jest liczbą doskonałą;
- $1 + 2^1 + 2^2 = 7$ - jest liczbą pierwszą, czyli $7 \cdot 2^2 = 28$ jest liczbą doskonałą;
- $1 + 2^1 + 2^2 + 2^3 = 15$ - nie jest liczbą pierwszą;
- $1 + 2^1 + 2^2 + 2^3 + 2^4 = 31$ - jest liczbą pierwszą, czyli $31 \cdot 2^4 = 496$ jest liczbą doskonałą;
- itd.

Przykład 2

Napiszmy program, który wypisze listę liczb doskonałych w przedziale $\langle 0, n \rangle$. Działanie naszego programu przetestujemy dla `n = 10000`.

Specyfikacja:

Dane:

- `n` - górna granica zakresu; liczba naturalna

Wyniki:

Program ma znaleźć i wypisać liczby doskonałe z podanego przedziału.

Zacznijmy od zdefiniowania funkcji pomocniczej `suma_dzielnikow(liczba)`, która będzie zwracała sumę dzielników właściwych danej liczby. W tym celu, zaczynając od 1, sprawdzamy, czy kolejne liczby są dzielnikami liczby `liczba` (wynik operacji modulo, operator `%`, będzie równy `0`). Jeżeli tak, to dodajemy do sumy wartość tego dzielnika.

```
def suma_dzielnikow(liczba):
    suma = 0
    dzielnik = 1

    while dzielnik <= liczba / 2:
        if liczba % dzielnik == 0:
            suma += dzielnik
        dzielnik += 1

    return suma
```

Następnie stworzymy funkcję `czy_doskonala(liczba)`, która będzie odpowiadała na pytanie, czy dana liczba jest liczbą doskonałą. W tym celu sprawdzamy, czy `liczba` jest równa sumie dzielników tej liczby.

```
def czy_doskonala(liczba):
    if liczba == suma_dzielnikow(liczba):
        return True
    return False
```

Na samym końcu tworzymy pętlę, która będzie szukała liczb doskonałych w zadanym przedziale. Sprawdzamy kolejne liczby - jeżeli spełniają warunki, zapisujemy je w liście wynikowej i wyświetlamy nasze rozwiązanie.

```
n = 10000

doskonale = []
for liczba in range(1, n):
    if czy_doskonala(liczba):
        doskonale.append(liczba)

print('Liczby doskonałe:', doskonale)
```

Oto kod całego programu:

```
# Obliczanie sumy dzielnikow wlasciwych
def suma_dzielnikow(liczba):
    suma = 0
    dzielnik = 1

    while dzielnik <= liczba / 2:
        if liczba % dzielnik == 0:
            suma += dzielnik
        dzielnik += 1

    return suma

# Sprawdzanie, czy dana liczba jest liczba doskonala
def czy_doskonala(liczba):
    if liczba == suma_dzielnikow(liczba):
```

```
        return True
    return False

# Kod wyszukujący liczby doskonale
n = 10000

doskonale = []
for liczba in range(1, n):
    if czy_doskonala(liczba):
        doskonale.append(liczba)

# Wyswietlanie wynikow
print('Liczby doskonałe:', doskonale)
```

Wynik wywołania kodu będzie następujący:

```
Liczby doskonałe: [6, 28, 496, 8128]
```

Ciekawostka

Oprócz liczb, które znaleźliśmy, kolejnymi liczbami doskonałymi są liczby: 33 550 336, 8 589 869 056, 137 438 691 328.

Wszystkie znane dotychczas liczby doskonałe są parzyste. Nie udało się jednak do tej pory udowodnić, że liczby doskonałe nieparzyste nie istnieją.

Dla zainteresowanych

Zaczynamy od kodu z Przykładu 2.

Możemy sprawdzić, ile czasu zabiera naszemu komputerowi policzenie kolejnych liczb i zwizualizować te czasy za pomocą modułu matplotlib.

W tym celu, na samej górze pliku, importujemy odpowiednią bibliotekę do tworzenia wykresów (`matplotlib.pyplot`) i ustawiamy jej alias `plt`. Potrzebna nam będzie również funkcja `perf_counter()` z biblioteki `time`, która umożliwi dokładny pomiar czasu.

```
import matplotlib.pyplot as plt
from time import perf_counter
```

Na początku przygotowujemy dwie tablice: `X`, która będzie przechowywała wartości na osi OX oraz `czasy`, która będzie przechowywała czas, jaki upłynął od rozpoczęcia wykonywania obliczeń, do ukończenia sprawdzania każdej liczby, czy jest ona liczbą doskonałą.

```
X = []
czasy = []
```

Rozpoczynamy pomiar, zapisując w zmiennej `czas_startu` czas rozpoczęcia wykonywania obliczeń. Ponieważ zależy nam na dokładności obliczeń, używamy do tego funkcji `perf_counter()` z biblioteki `time`.

```
X = []
czasy = []
czas_startu = perf_counter()
```

Następnie modyfikujemy pętlę, która odpowiadała za sprawdzanie kolejnych liczb, czy są liczbami doskonałym. Po sprawdzeniu każdej liczby dodajemy do utworzonych tablic wartości: do tablicy `X` dodajemy liczbę, którą sprawdziliśmy, a do tablicy `czasy` - czas, jaki upłynął od rozpoczęcia obliczeń.

```
X = []
czasy = []
czas_startu = perf_counter()

for liczba in range(1, n):
    if czy_doskonala(liczba):
        doskonale.append(liczba)

    ile = perf_counter() - czas_startu
    X.append(liczba)
    czasy.append(ile)
```

Ostatnim krokiem będzie wyświetlenie wykresu. W kolejnych krokach używamy funkcji:

- `plt.plot()` - rzutujemy na wykres kolejne punkty o współrzędnych z tablic `X` oraz `czasy`,
- `plt.xlabel()` - ustawiamy opis osi X,
- `plt.ylabel()` - ustawiamy opis osi Y,

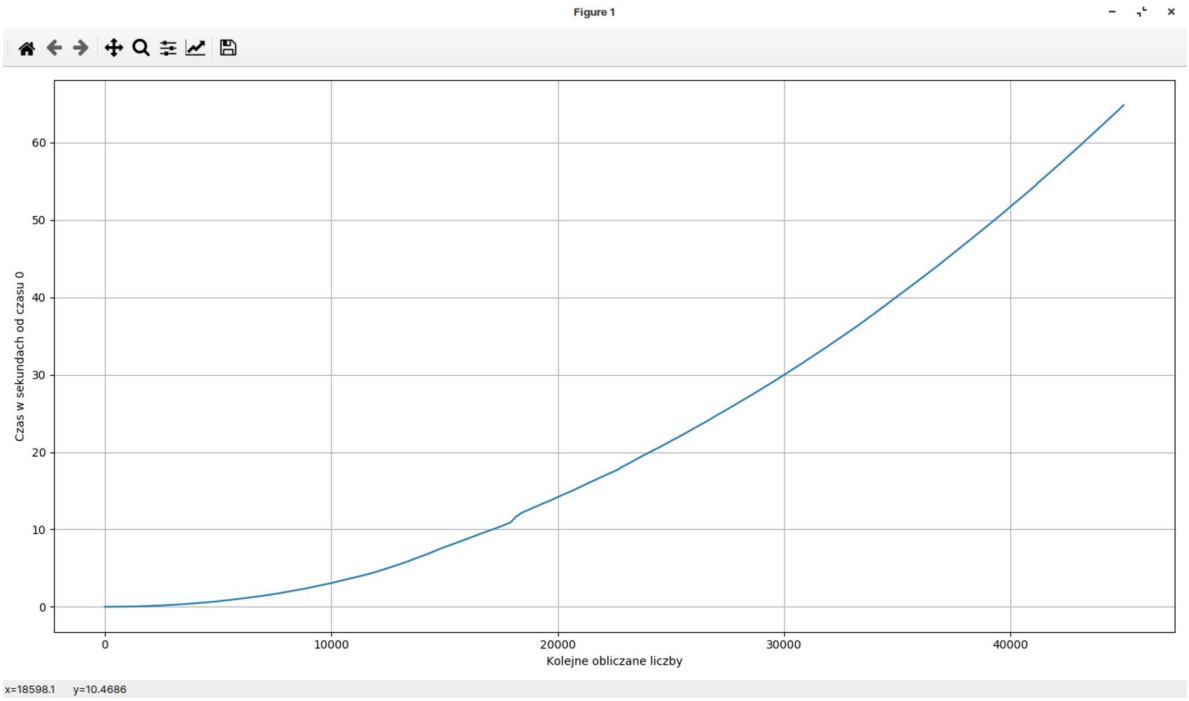
- `plt.grid()` - włączamy wyświetlanie siatki na wykresie,
- `plt.show()` - wyświetlamy wykres.

```
plt.plot(X, czasy)
plt.xlabel('Kolejne obliczane liczby')
plt.ylabel('Czas w sekundach od czasu 0')
plt.grid(True)
plt.show()
```

Oto kod całego programu:

```
import matplotlib.pyplot as plt
from time import perf_counter

# Obliczanie sumy dzielnikow wlasciwych
def suma_dzielnikow(liczba):
    suma = 0
    dzielnik = 1

    while dzielnik <= liczba / 2:
        if liczba % dzielnik == 0:
            suma += dzielnik
        dzielnik += 1

    return suma

# Sprawdzanie, czy dana liczba jest liczba doskonala
def czy_doskonala(liczba):
    if liczba == suma_dzielnikow(liczba):
        return True
    return False

n = 45000
doskonale = []

X = []
czasy = []
czas_startu = perf_counter()

for liczba in range(1, n):
```

```
30      if czy_doskonala(liczba):
31          doskonale.append(liczba)
32
33      ile = perf_counter() - czas_startu
34      X.append(liczba)
35      czasy.append(ile)
36
37
38  # Wyswietlanie wykresu
39  plt.plot(X, czasy)
40  plt.xlabel('Kolejne obliczane liczby')
41  plt.ylabel('Czas w sekundach od czasu 0')
42  plt.grid(True)
43  plt.show()
```



Możemy zauważyć, jak długo trwa wyszukiwanie ostatniej liczby.

Słownik

dzielnik właściwy

każdy dzielnik danej liczby, który jest od niej mniejszy

matplotlib

biblioteka służąca do przedstawienia obrazów złożonych z punktów o współrzędnych `x` oraz `y` (wykresów, histogramów, rozkładów itp.); moduł `matplotlib` nie jest dostępny

> w standardowej instalacji języka Python –należy go zainstalować, korzystając z mechanizmu `pip`

Schemat interaktywny

Liczby zaprzyjaźnione

> **Definicja: Liczby zaprzyjaźnione**
>
> Liczby zaprzyjaźnione to para liczb naturalnych: pierwsza jest sumą wszystkich dzielników właściwych drugiej liczby, a druga - sumą dzielników właściwych pierwszej.
>
> Jedną z takich liczb jest para: 220 i 284. Wynika to z następującego faktu:
>
> - $220 = 1 + 2 + 4 + 71 + 142$ (dzielniki 284)
> - $284 = 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110$ (dzielniki 220)

Polecenie 1

Napisz program, który sprawdzi, czy podane liczby są liczbami zaprzyjaźnionymi. Jego działanie przetestuj dla pary liczb 220 oraz 284.

Specyfikacja:

Dane:

- `liczba_1` – sprawdzana liczba; liczba naturalna
- `liczba_2` – sprawdzana liczba; liczba naturalna

Wynik:

Program wyświetla komunikat informujący użytkownika, czy podane liczby są liczbami zaprzyjaźnionymi.

```
liczba_1 = 220
liczba_2 = 284

# Tutaj dodaj własny kod. Użyj funkcji
# print()
```

```
```

Polecenie 2

Porównaj swoje rozwiązanie z prezentacją przedstawiającą zapis programu w języku Python, który sprawdza, czy dana para liczb to liczby zaprzyjaźnione.

1

Na początku definiujemy wartości sprawdzanych liczb.

```
1 liczba_1 = 220
2 liczba_2 = 284
```

2

Przygotowujemy funkcję, która będzie obliczała dzielniki właściwe liczby podanej w parametrze. W tym celu dodajemy do listy `lista` kolejne liczby, które są dzielnikami zadanej liczby.

```
1 def
  dzielniki_wlasciwe(liczba)
  :
2     if liczba == 0:
3         return None
4
```

```
    lista = []
    polowka = liczba // 2

    for i in range(1, polowka + 1):
        if liczba % i == 0:

lista.append(i)

    return lista
```

3

Następnie obliczamy dzielniki pierwsze zadanych liczb:

```
dzielniki_1 = dzielniki_wlasciwe(liczba_1)
dzielniki_2 = dzielniki_wlasciwe(liczba_2)

```

4

Przygotowujemy funkcję, która będzie sumowała elementy listy podanej jako parametr.

```
def sumuj(tablica):
    suma = 0

    for liczba in tablica:
        suma += liczba

    return suma
```

5

W kolejnym kroku obliczamy sumę dzielników liczb:

```
suma_1 = sumuj(dzielniki_1)
suma_2 = sumuj(dzielniki_2)
```

6

Sprawdzamy, czy pierwsza liczba jest równa sumie dzielników drugiej liczby oraz czy druga liczba jest równa sumie dzielników pierwszej liczby:

```
if liczba_1 == suma_2 and liczba_2 == suma_1:
    pass
```

7

Jeśli oba te warunki są spełnione jednocześnie (operator logiczny and), wypisujemy odpowiedni komunikat, podając obliczone wartości:

```
if liczba_1 == suma_2 and liczba_2 == suma_1:
    print('Podane liczby sa liczbami zaprzyjaznionymi')
    print('Liczba', liczba_1, 'to suma dzielników', dzielniki_2, 'liczby', liczba_2)
    print('Liczba', liczba_2, 'to suma dzielników', dzielniki_1, 'liczby', liczba_1)
```

8

W przeciwnym wypadku informujemy, że sprawdzane liczby nie spełniają warunków liczb zaprzyjaźnionych.

```
if liczba_1 == suma_2 and liczba_2 == suma_1:
    print('Podane liczby sa liczbami zaprzyjaznionymi')
    print('Liczba', liczba_1, 'to suma dzielników', dzielniki_2, 'liczby', liczba_2)
    print('Liczba', liczba_2, 'to suma dzielników', dzielniki_1, 'liczby', liczba_1)

else:
    print('Niestety, to nie są liczby zaprzyjaźnione')
```

9

Gotowy kod wygląda następująco:

```
liczba_1 = 220
liczba_2 = 284

def dzielniki_wlasciwe(liczba):
    if liczba == 0:
        return None

    lista = []
    polowka = liczba // 2
```

```
10
11      for i in range(1,
    polowka + 1):
12          if liczba % i ==
    0:
13
    lista.append(i)
14
15      return lista
16
17  dzielniki_1 =
    dzielniki_wlasciwe(liczba_
    1)
18  dzielniki_2 =
    dzielniki_wlasciwe(liczba_
    2)
19
20  def sumuj(tablica):
21      suma = 0
22
23      for liczba in tablica:
24          suma += liczba
25
26      return suma
27
28  suma_1 =
    sumuj(dzielniki_1)
29  suma_2 =
    sumuj(dzielniki_2)
30
31  if liczba_1 == suma_2 and
    liczba_2 == suma_1:
32      print('Podane liczby
    sa liczbami
    zaprzyjaznionymi')
33      print('Liczba',
    liczba_1, 'to suma
    dzielników', dzielniki_2,
    'liczby', liczba_2)
34      print('Liczba',
    liczba_2, 'to suma
    dzielników', dzielniki_1,
    'liczby', liczba_1)
35
36  else:
```

```
    print('Niestety, to nie są liczby zaprzyjaźnione')
```

10 Wynik działania programu:

```
Podane liczby sa liczbami zaprzyjaznionymi
Liczba 220 to suma dzielników [1, 2, 4, 71, 142] liczby 284
Liczba 284 to suma dzielników [1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110] liczby 220
```

Ćwiczenie 1

Znasz już pojęcie **liczby doskonałej** (suma jej dzielników właściwych jest równa tej liczbie). Wyróżnić możemy również **liczby nadmiarowe** (suma dzielników właściwych danej liczby jest większa od tej liczby) oraz **liczby deficytowe** (suma dzielników właściwych danej liczby jest mniejsza od tej liczby).

Przykład liczby **doskonałej** to liczba **6**, ponieważ:

$$6 = 1 + 2 + 3$$

Przykład liczby **deficytowej** to liczba **3**, ponieważ:

$$3 > 1$$

Przykład liczby **nadmiarowej** to liczba **12**, ponieważ:

$$12 < 1 + 2 + 3 + 4 + 6$$

Wykorzystując schemat interaktywny, przygotuj program, który przypisze daną liczbę do odpowiedniej kategorii (liczby doskonałe, nadmiarowe lub deficytowe).

Przetestuj działanie programu dla liczby 27.

Specyfikacja problemu:

Dane:

- n – liczba naturalna

Wynik:

- `wynik` – komunikat `Liczba nadmiarowa`, `Liczba deficytowa` lub `Liczba doskonała`

Sprawdź się

Pokaż ćwiczenia: 

Ćwiczenie 1

Liczba narcystyczna to n-cyfrowa liczba naturalna, która jest równa sumie swoich cyfr podniesionych do potęgi n. Napisz funkcję `czy_narcystyczna(liczba)`, sprawdzającą, czy dana liczba jest liczbą narcystyczną. Działanie swojego programu przetestuj dla liczby `370`.

W celu podniesienia liczby do potęgi, użyj operatora `**`, np. polecenie: a `**` b podniesie liczbę a do potęgi b.

Przykład:

Kolejne liczby narcystyczne: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407 ...

- 6 jest liczbą narcystyczną, ponieważ: $6^1 = 6$
- 153 jest liczbą narcystyczną, ponieważ: $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$

Specyfikacja:

Dane:

- `liczba` – sprawdzana liczba; liczba naturalna

Wynik:

Program wypisze komunikat `Podana liczba jest narcystyczna!` lub `Podana liczba nie jest narcystyczna`

Twoje zadania

1. Zdefiniowanie funkcji `czy_narcystyczna()`, sprawdzającej, czy podana liczba spełnia warunki liczby narcystycznej.

Ćwiczenie 2

Zdefiniuj funkcję `czy_liczby_zaprzyjaznione(lista)`, sprawdzającą, które pary z listy par liczb, zapisanych jako krotki, to liczby zaprzyjaźnione.

Przetestuj działanie swojego programu dla list:

- `[(220, 284), (33, 451), (128, 363)]` - wynik: `[True, False, False]`
- `[(33, 451), (220, 284), (128, 363)]` - wynik: `[False, True, False]`

Specyfikacja:

Dane:

- `lista` – pary liczb do sprawdzania; lista krotek dwuelementowych

Wynik:

Funkcja zwraca listę wartości typu `bool`, prezentującą, które z par liczb to liczby zaprzyjaźnione.

Twoje zadania

1. Program sprawdza, które z par liczb spełniają warunki liczb zaprzyjaźnionych.

Ćwiczenie 3

Napisz program wyszukujący liczby bliźniacze z podanego zakresu. Program powinien wypisywać tylko te pary liczb bliźniaczych, w których jedna lub obie liczby kończą się cyfrą 3. Każdą parę liczb wypisuj w nowej linii, oddzielając liczby w parze znakiem spacji. Przetestuj swój program dla przedziału $\langle 2, 100 \rangle$.

Specyfikacja:

Dane:

- a - początek sprawdzanego przedziału; liczba naturalna
- b - koniec sprawdzanego przedziału; liczba naturalna

Wynik:

Program wypisze w kolejnych liniach pary liczb bliźniaczych, z przedziału $\langle a, b \rangle$, takich, że chociaż jedna z nich kończy się cyfrą 3. Pary powinny być oddzielone znakiem spacji.

Twoje zadania

1. Program ma wyszukiwać pary liczb bliźniaczych w przedziale $\langle 2, 100 \rangle$. Wypisywane powinny być te pary, w których jedna lub dwie z liczb kończą się cyfrą 3.

Dla nauczyciela

Autor: Adam Jurkiewicz
Przedmiot: Informatyka

Temat: Algorytmy liczbowe w języku Python

Grupa docelowa:

Szkoła ponadpodstawowa, liceum ogólnokształcące, technikum, zakres podstawowy i rozszerzony

Podstawa programowa:

Cele kształcenia – wymagania ogólne

I. Rozumienie, analizowanie i rozwiązywanie problemów na bazie logicznego i abstrakcyjnego myślenia, myślenia algorytmicznego i sposobów reprezentowania informacji.

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera oraz innych urządzeń cyfrowych: układanie i programowanie algorytmów, organizowanie, wyszukiwanie i udostępnianie informacji, posługiwanie się aplikacjami komputerowymi.

Treści nauczania – wymagania szczegółowe

I. Rozumienie, analizowanie i rozwiązywanie problemów.

Zakres podstawowy. Uczeń:

1) planuje kolejne kroki rozwiązywania problemu, z uwzględnieniem podstawowych etapów myślenia komputacyjnego (określenie problemu, definicja modeli i pojęć, znalezienie rozwiązania, zaprogramowanie i testowanie rozwiązania).

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera i innych urządzeń cyfrowych.

Zakres podstawowy. Uczeń:

1) projektuje i programuje rozwiązania problemów z różnych dziedzin, stosuje przy tym: instrukcje wejścia/wyjścia, wyrażenia arytmetyczne i logiczne, instrukcje warunkowe, instrukcje iteracyjne, funkcje z parametrami i bez parametrów, testuje poprawność programów dla różnych danych; w szczególności programuje algorytmy z punktu I.2);

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

3) sprawnie posługuje się zintegrowanym środowiskiem programistycznym przy pisaniu, uruchamianiu i testowaniu programów;

Kształtowane kompetencje kluczowe:

- kompetencje cyfrowe;
- kompetencje osobiste, społeczne i w zakresie umiejętności uczenia się;
- kompetencje matematyczne oraz kompetencje w zakresie nauk przyrodniczych, technologii i inżynierii.

Cele operacyjne (językiem ucznia):

- Przedstawisz algorytm szukania liczb bliźniaczych wśród liczb pierwszych.
- Scharakteryzujesz sposób na otrzymanie liczby doskonałej.
- Sprawdzisz, czy dwie liczby są zaprzyjaźnione.
- Użyjesz różnych struktur danych w języku Python.

Strategie nauczania:

- konstruktywizm;
- konektywizm.

Metody i techniki nauczania:

- dyskusja;
- rozmowa nauczająca z wykorzystaniem multimedium i ćwiczeń interaktywnych;
- ćwiczenia praktyczne.

Formy pracy:

- praca indywidualna;
- praca w parach;
- praca w grupach;
- praca całego zespołu klasowego.

Środki dydaktyczne:

- komputery z głośnikami, słuchawkami i dostępem do internetu;
- zasoby multimedialne zawarte w e-materiale;
- tablica interaktywna/tablica, pisak/kreda;
- oprogramowanie dla języka Python 3 (lub nowszej wersji), w tym PyCharm lub IDLE.

Przebieg lekcji

Przed lekcją:

1. Uczniowie przypominają sobie algorytmy liczbowe zaprezentowane w e-materiale teoretycznym.
2. Chętne lub wybrane osoby przygotowują prezentacje dotyczące liczb o ciekawych właściwościach.
3. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Algorytmy liczbowe w języku Python". Uczniowie zapoznają się z treściami w sekcji „Przeczytaj" w kontekście programowania.

Faza wstępna:

1. **Rozpoznanie wiedzy uczniów**. Chętna lub wybrana osoba przypomina poznane algorytmy liczbowe.
2. Nauczyciel wyświetla uczniom temat, wskazuje cele zajęć oraz ustala z uczestnikami zajęć kryteria sukcesu.

Faza realizacyjna:

1. Nauczyciel sprawdza przygotowanie uczniów do lekcji.
2. Uczniowie prezentują przygotowane przez siebie przyklady liczb o ciekawych właściwościach.
3. Uczniowie na podstawie zaprezentowanych wystąpień przygotowują mapę myśli podsumowującą informacje o liczbach o ciekawych właściwościach.
4. **Praca z tekstem.** Nauczyciel wyświetla zawartość sekcji „Przeczytaj". Na forum klasy uczniowie analizują przedstawione w niej rozwiązania Przykładów 1, 2 i 3. Nauczyciel odpowiada na ew. pytania.
5. **Praca z multimedium.** Nauczyciel wyświetla zawartość sekcji „Schemat interaktywny". Uczniowie w parach rozwiązują Problem 1. Porównują swoje implementacje z prezentacją. Następnie indywidualnie wykonują Ćwiczenie 1 z sekcji „Schemat interaktywny".
6. Chętna lub wybrana osoba prezentuje swoje rozwiązanie. Nauczyciel omawia je i wskazuje ew. błędy.
7. **Ćwiczenie umiejętności**. Uczniowie wykonują Ćwiczenia 1–2 z sekcji „Sprawdź się". Chętna lub wybrana osoba prezentuje swoje rozwiązanie. Nauczyciel omawia je i wskazuje ew. błędy.

Faza podsumowująca:

1. Nauczyciel ponownie wyświetla na tablicy temat i cele lekcji zawarte w sekcji „Wprowadzenie". W kontekście ich realizacji następuje omówienie ewentualnych problemów z rozwiązaniem ćwiczeń z sekcji „Sprawdź się".

Praca domowa:

1. Uczniowie wykonują Ćwiczenie 3 z sekcji „Sprawdź się”.

Materiały pomocnicze:

- Oficjalna dokumentacja techniczna dla języka Python 3 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania PyCharm lub IDLE.

Wskazówki metodyczne:

- Treści w sekcji „Przeczytaj” można wykorzystać jako podsumowanie i utrwalenie wiedzy uczniów.