Zadanie 1

Sprawdź, czy suma dwóch liczb zmiennoprzecinkowych (zarówno w pojedynczej jak i podwójnej precyzji) jest zawsze równa oczekiwanej matematycznie wartości. Wyjaśnij, dlaczego odpowiedź może być błędna przy bezpośrednim porównaniu liczb zmiennoprzecinkowych.

PYTHON

```python
import numpy as np

def suma_pojedyncza_precyzja(a, b):
    a32 = np.float32(a)
    b32 = np.float32(b)
    return np.float32(a32 + b32)

poj = suma_pojedyncza_precyzja(0.1, 0.2)

print(poj)

def suma_podwojna_precyzja(a: float, b: float) -> float:
    return a + b

pod = suma_podwojna_precyzja(0.1, 0.2)

print(pod)

print("Czy równe?: ", poj == pod)
```

```
0.3
0.30000000000000004
Czy równe?:  False
```

dlaczego odpowiedź może być błędna przy bezpośrednim porównaniu liczb zmiennoprzecinkowych

Komputery przechowują liczby zmiennoprzecinkowe w formacie binarnym (standard IEEE 754).

Niektóre liczby dziesiętne, takie jak:

- 0.1
- 0.2

- 0.3

nie mają dokładnej reprezentacji w systemie binarnym, podobnie jak 1/3 nie ma skończonego zapisu w systemie dziesiętnym.

W efekcie:

- 0.1 jest zapisywane jako liczba bardzo bliska 0.1
- 0.2 również
- ich suma daje 0.30000000000000004 zamiast dokładnie 0.3

To jest **błąd reprezentacji (rounding error)**.

Dlaczego bezpośrednie porównanie jest błędne?

Porównanie:

`suma == 0.3`

sprawdza **dokładną równość bitową**, a nie matematyczną równość w sensie „wystarczająco blisko".

Z powodu minimalnych błędów zaokrągleń wynik może się różnić o bardzo małą wartość (np. 5e-17), więc porównanie zwróci `False`.

Poprawny sposób porównywania

Zamiast `==` należy używać porównania z tolerancją:

`import math math.isclose(suma, 0.3)`

lub dla NumPy:

`np.isclose(suma32, np.float32(0.3))`

Można też ręcznie sprawdzić:

`abs(suma - 0.3) < 1e-9`

Wniosek

Suma dwóch liczb zmiennoprzecinkowych **nie zawsze** jest dokładnie równa wartości matematycznej.
Bezpośrednie porównanie (`==`) może dawać błędne wyniki.
Należy stosować porównania z tolerancją (`isclose`).

Zadanie 2

Sprawdź, co się stanie, gdy dodasz bardzo dużą liczbę do bardzo małej, a następnie odejmiesz dużą liczbę od wyniku.

```python
a = 10000000000000000
b = 0.000000000000001

wynik = (a + b) - a

print(wynik)
```

```
0.0
```

Matematycznie

Mamy:

a = 10000000000000000
b = 0.000000000000001

czyli:

a = 10^16
b = 10^-15

Matematycznie:

(10^16 + 10^-15) - 10^16 = 10^-15

Powinniśmy dostać:

0.000000000000001

Co dzieje się w praktyce?

Wynik będzie:

0.0

Dlaczego?

1. Ograniczona precyzja `float64`

Liczby zmiennoprzecinkowe typu `float` w Pythonie mają:

- 64 bity
- około 15–16 cyfr znaczących

Liczba:

10000000000000000

ma już **16 cyfr znaczących**.

To oznacza, że:

- cała dostępna precyzja jest zużyta na zapis tej dużej liczby,
- nie ma już miejsca na zapis bardzo małej zmiany rzędu `10^-15`.

2. Co dzieje się przy dodawaniu?

Komputer próbuje policzyć:

10000000000000000 + 0.000000000000001

Ale różnica rzędów wielkości wynosi:

10^16 vs 10^-15

To różnica **31 rzędów wielkości**.

Mała liczba jest tak niewyobrażalnie mała względem dużej, że:

> jej wpływ mieści się poza zakresem precyzji mantysy

W efekcie:

a + b ≈ a

czyli w pamięci:

10000000000000000 + 0.000000000000001
= 10000000000000000

3. Odejmowanie

Skoro w pamięci mamy:

(a + b) = a

to:

(a + b) - a = 0

Co to pokazuje?

- Bardzo małe liczby mogą całkowicie „zniknąć” przy dodawaniu do bardzo dużych.
- Operacje na liczbach zmiennoprzecinkowych nie są dokładne przy dużych różnicach skali.
- To przykład **utraty precyzji** wynikającej z ograniczonej liczby cyfr znaczących.

Kluczowa intuicja!

To tak, jakbyś próbował dodać:

10 000 000 000 000 000,00

- 0,000000000000001

Przy dokładności do grosza ta zmiana po prostu nie istnieje.

Zadanie 3

Wyświetl liczbę zmiennoprzecinkową jako float i double z precyzją do 20 miejsc po przecinku.

PYTHON

```python
import numpy as np

liczba = 0.1

# float - pojedyncza precyzja
liczba_f = np.float32(liczba)

# double - podwójna precyzja
liczba_d = float(liczba)

print("float: ", format(liczba_f, ".20f"))
print("double: ", format(liczba_d, ".20f"))
```

```
float:  0.10000000149011611938
double:  0.1000000000000000555
```

- `float32` ma mniejszą precyzję - szybciej pojawia się błąd.
- `float64` jest dokładniejszy, ale też nie przechowuje 0.1 idealnie.
- 20 miejsc po przecinku pokazuje rzeczywistą reprezentację w pamięci.

Zadanie 4

Oblicz 0,3 · 3 + 0,1 i porównaj wynik z jego wartościami zaokrąglonymi do dołu i do góry (floor i ceil).

PYTHON

```python
import math

wynik = 0.3 * 3 + 0.1

print("Wynik zwykły: ", wynik)
print("Wynik zaokrąglony w dół: ", math.floor(wynik))
print("Wynik zaokrąglony w górę: ", math.ceil(wynik))
```

```
Wynik zwykły:  0.9999999999999999
Wynik zaokrąglony w dół:  0
Wynik zaokrąglony w górę:  1
```

> Wynik nie jest równy dokładnie 1 z powodu błędu reprezentacji binarnej liczb 0.3 i 0.1 w standardzie IEEE 754.

Zadanie 5

Oblicz różnicę między 1,0000001 i 1,0000000 oraz między 1,0000002 i 1,0000001. Wyjaśnij, dlaczego wyniki mogą się różnić od teoretycznej różnicy.

PYTHON

```python
wynik1 = 1.0000001 - 1.0000000
wynik2 = 1.0000002 - 1.0000001

print("wynik1: ", wynik1)
print("wynik2: ", wynik2)
```

```
wynik1:  1.0000000005838672e-07
wynik2:  9.999999983634211e-08
```

Dlaczego wyniki różnią się od teoretycznej różnicy?

Dzieje się tak z powodu **błędów reprezentacji liczb zmiennoprzecinkowych w systemie binarnym (IEEE 754)**.

Liczby nie są przechowywane dokładnie

Liczby:

1.0000001
1.0000002

nie mają idealnej reprezentacji binarnej. W pamięci są zapisane jako **najbliższe możliwe przybliżenia**.

Czyli komputer faktycznie odejmuje:

(1.0000001 + mały_błąd1) - (1.0000000 + mały_błąd2)

Odejmowanie liczb bliskich sobie

Tutaj odejmujemy liczby prawie równe:

1.0000001 - 1.0000000

To powoduje zjawisko zwane:

utratą cyfr znaczących (catastrophic cancellation)

Podczas odejmowania:

- duże wspólne cyfry się „kasują”,
- pozostaje bardzo mała różnica,
- a względny wpływ błędu zaokrąglenia rośnie.

Dlatego wyniki:

1.0000000005838672e-07
9.999999983634211e-08

różnią się minimalnie od siebie i od idealnego `1e-7`.

Podsumowanie

To są liczby bardzo bliskie `1e-7`, więc obliczenia są poprawne - różnice wynikają z reprezentacji zmiennoprzecinkowej.

Wyniki różnią się od teoretycznej wartości, ponieważ liczby zmiennoprzecinkowe nie są przechowywane dokładnie w systemie binarnym. Podczas odejmowania liczb bardzo bliskich sobie dochodzi do utraty cyfr znaczących, co powoduje, że niewielkie błędy reprezentacji stają się widoczne w wyniku.

Zadanie 6

Podziel liczbę 1,0 przez 0,0 i liczbę 0,0 przez 0,0. Sprawdź, co zwrócą te operacje.

PYTHON

```python
wynik1 = 1.0 / 0.0
wynik2 = 0.0 / 0.0

print("wynik dzielenia pierwszego: ", wynik1)
print("wynik dzielenia drugiego: ", wynik2)
```

```
Traceback (most recent call last):
  File "zad6.py", line 1, in <module>
    wynik1 = 1.0 / 0.0
             ~~~~^~~~
ZeroDivisionError: float division by zero
----------------------------------------
Traceback (most recent call last):
  File "zad6.py", line 2, in <module>
    wynik2 = 0.0 / 0.0
             ~~~~^~~~
ZeroDivisionError: float division by zero
```

Mimo że liczby zmiennoprzecinkowe są zgodne ze standardem IEEE 754, Python **celowo zgłasza wyjątek**, aby zapobiec dalszym obliczeniom na niepoprawnych wartościach.

W Pythonie dzielenie przez zero dla liczb zmiennoprzecinkowych powoduje zgłoszenie wyjątku `ZeroDivisionError`. Jednak zgodnie ze standardem IEEE 754 operacja 1.0 / 0.0 powinna zwrócić nieskończoność (∞), a 0.0 / 0.0 wartość NaN (Not a Number). W bibliotekach takich jak NumPy zachowanie jest zgodne z IEEE 754 i zamiast wyjątku zwracane są wartości `inf` oraz `nan`.

Co mówi standard IEEE 754?

W czystej arytmetyce IEEE 754 (np. w NumPy):

Operacja	Wynik
`1.0 / 0.0`	`+∞` (infinity)
`-1.0 / 0.0`	`-∞`
`0.0 / 0.0`	`NaN` (Not a Number)

Jak to sprawdzić w NumPy?

```python
import numpy as np

print(np.float64(1.0) / np.float64(0.0))
print(np.float64(0.0) / np.float64(0.0))
```

Wtedy otrzymasz:

```
inf
nan
```

(plus ostrzeżenie RuntimeWarning zamiast wyjątku)

Różnica

- 🔹 Python (czysty `float`) → zgłasza `ZeroDivisionError`
- 🔹 NumPy → zwraca `inf` lub `nan` zgodnie z IEEE 754

Zadanie 7

Oblicz maszynowy epsilon dla typów float i double i porównaj wyniki. Wyjaśnij, czym jest maszynowy epsilon i jak wpływa na dokładność obliczeń komputerowych.

Maszynowy epsilon - definicja

Maszynowy epsilon (ε_{mach}) to najmniejsza dodatnia liczba ε, taka że:

$$1 + \varepsilon > 1$$

w arytmetyce zmiennoprzecinkowej danego typu.

Oznacza to najmniejszą wartość, którą komputer jest w stanie „zauważyć” przy dodaniu do 1.
Wyznacza on granicę precyzji reprezentacji liczb w komputerze.

Obliczenie maszynowego epsilon w Pythonie

Maszynowy epsilon można wyznaczyć algorytmicznie, zmniejszając wartość ε tak długo, aż suma $1 + \varepsilon$ przestanie być większa od 1.

```python
import numpy as np

def epsilon_float32():
    eps = np.float32(1.0)
    while np.float32(1.0) + eps / np.float32(2.0) > np.float32(1.0):
        eps = eps / np.float32(2.0)
    return eps

def epsilon_float64():
    eps = 1.0
    while 1.0 + eps / 2.0 > 1.0:
        eps = eps / 2.0
    return eps

eps32 = epsilon_float32()
eps64 = epsilon_float64()

print("Epsilon float32:", eps32)
print("Epsilon float64:", eps64)
```

```
Epsilon float32: 1.1920929e-07
Epsilon float64: 2.220446049250313e-16
```

Typ	ε_{mach}
float32	$\approx 1.19 \cdot 10^{-7}$
float64	$\approx 2.22 \cdot 10^{-16}$

Porównując wartości:

$$\frac{1.19 \cdot 10^{-7}}{2.22 \cdot 10^{-16}} \approx 10^{9}$$

Oznacza to, że typ **float64 jest około miliard razy dokładniejszy niż float32**.

Wpływ maszynowego epsilon na dokładność obliczeń

1. Ogranicza liczbę cyfr znaczących

- float32 → około 7 cyfr znaczących
- float64 → około 15–16 cyfr znaczących

2. Określa maksymalny względny błąd pojedynczej operacji

Każda operacja arytmetyczna wprowadza błąd rzędu ε_{mach}.

3. Wpływa na stabilność obliczeń

- Przy odejmowaniu liczb bardzo bliskich sobie może dojść do utraty cyfr znaczących.
- Przy dużej liczbie operacji błędy mogą się kumulować.

4. Wyznacza granicę rozróżnialności liczb

Jeśli dodamy do 1 liczbę mniejszą niż ε_{mach}, wynik nadal będzie równy 1, ponieważ zmiana jest poniżej granicy precyzji.

Wniosek

Maszynowy epsilon:

- dla **float32** wynosi około $\mathbf{1.19 \cdot 10^{-7}}$
- dla **float64** wynosi około $\mathbf{2.22 \cdot 10^{-16}}$

Im mniejszy epsilon, tym większa dokładność obliczeń komputerowych.
Dlatego typ double (float64) zapewnia znacznie większą precyzję niż float (float32).

Zadanie 8

Sumuj liczbę 0,0001 w pętli 1.000.000 razy i porównaj wynik z wynikiem uzyskanym przez mnożenie 1.000.000 przez 0,0001. Wyjaśnij, dlaczego mogą wystąpić różnice.

```python
# liczba iteracji
n = 1_000_000
wartosc = 0.0001

# Sumowanie w pętli
suma_petla = 0.0
for _ in range(n):
    suma_petla += wartosc

# Mnożenie
suma_mnozenie = n * wartosc

# Różnica
roznica = suma_petla - suma_mnozenie

print("Wynik sumowania w pętli:", suma_petla)
print("Wynik mnożenia:", suma_mnozenie)
print("Różnica:", roznica)
```

```
Wynik sumowania w pętli: 100.00000000219612
Wynik mnożenia: 100.0
Różnica: 2.1961170659778873e-09
```

Dlaczego występuje różnica?

1. 0.0001 nie ma dokładnej reprezentacji binarnej

Liczba `0.0001` nie jest zapisywana idealnie w systemie binarnym — jest przechowywana jako przybliżenie.

2. Akumulacja błędu

W pętli wykonujemy milion operacji dodawania:

suma = suma + przybliżona_wartość

Każde dodanie wprowadza bardzo mały błąd zaokrąglenia.
Po 1 000 000 iteracjach te małe błędy się sumują.

To zjawisko nazywamy:

> **akumulacją błędu numerycznego**

3. Dlaczego mnożenie daje dokładniejszy wynik?

Mnożenie wykonuje:

n * wartosc

czyli jedną operację zamiast miliona, więc błąd pojawia się tylko raz, a nie milion razy.

Wniosek

- Sumowanie wielu małych liczb może prowadzić do akumulacji błędów.
- Wynik obliczony w pętli może różnić się od wyniku mnożenia.
- To pokazuje, że operacje zmiennoprzecinkowe nie są dokładne matematycznie.
- Im więcej operacji, tym większe ryzyko narastania błędu.

Zadanie 9

Oblicz sumę odwrotności liczb od 1 do 1.000.000 w kolejności rosnącej i malejącej. Porównaj wyniki.

PYTHON

```python
n = 1_000_000

# 1️⃣ Sumowanie w kolejności rosnącej (od 1 do 1_000_000)
suma_rosnaco = 0.0
for i in range(1, n + 1):
    suma_rosnaco += 1.0 / i

# 2️⃣ Sumowanie w kolejności malejącej (od 1_000_000 do 1)
suma_malejaco = 0.0
for i in range(n, 0, -1):
    suma_malejaco += 1.0 / i

# Różnica
roznica = suma_rosnaco - suma_malejaco

print("Suma rosnąco:", suma_rosnaco)
print("Suma malejąco:", suma_malejaco)
print("Różnica:", roznica)
```

Matematycznie obie sumy powinny być identyczne:

$$\sum_{i=1}^{1,000,000} \frac{1}{i}$$

Jest to tzw. **milionowy wyraz szeregu harmonicznego**.

W praktyce otrzymujemy:

```
Suma rosnąco: 14.392726722864989
Suma malejąco: 14.392726722865772
Różnica: -7.833733661755105e-13
```

Dlaczego wyniki się różnią?

1. Ograniczona precyzja liczb zmiennoprzecinkowych

Liczby typu `float` mają ograniczoną liczbę cyfr znaczących (około 15–16).

2. Akumulacja błędu zaokrągleń

Każde dodawanie wprowadza bardzo mały błąd.
Ponieważ wykonujemy **milion operacji**, błędy się kumulują.

3. Znaczenie kolejności dodawania

To kluczowe.

🔹 Sumowanie rosnąco (1 → 1 000 000)

Na początku dodajemy duże liczby (1, 1/2, 1/3...).
Na końcu bardzo małe liczby (np. 1/1 000 000).

Małe składniki dodawane do dużej sumy mogą zostać częściowo „zgubione" z powodu ograniczonej precyzji.

🔹 Sumowanie malejąco (1 000 000 → 1)

Najpierw dodajemy bardzo małe liczby.
Suma jest jeszcze mała, więc precyzja względna jest lepsza.

Dopiero później dodawane są większe składniki.

➞ Ta metoda daje zwykle **dokładniejszy wynik**.

Wniosek

- Matematycznie kolejność sumowania nie ma znaczenia.
- W arytmetyce zmiennoprzecinkowej ma znaczenie.

- Sumowanie od najmniejszych do największych wartości jest numerycznie stabilniejsze.
- Różnice wynikają z ograniczonej precyzji i akumulacji błędów zaokrągleń.

Zadanie 10

Niech

$$f(x) = \sqrt{x^2 + 1} - 1$$

oraz

$$g(x) = \frac{x^2}{\sqrt{x^2 + 1} + 1}.$$

Łatwo zauważyć, że $g = f$. Oblicz i porównaj wartości funkcji g i f dla:

$$x = 8^{-1},\ 8^{-2},\ 8^{-3},\ \ldots$$

Łatwo zauważyć, że algebraicznie:

$$g(x) = f(x)$$

ponieważ:

$$\sqrt{x^2 + 1} - 1 = \frac{(\sqrt{x^2 + 1} - 1)(\sqrt{x^2 + 1} + 1)}{\sqrt{x^2 + 1} + 1} = \frac{x^2}{\sqrt{x^2 + 1} + 1}$$

Obliczenia w Pythonie

PYTHON

```python
import math

def f(x):
    return math.sqrt(x**2 + 1) - 1

def g(x):
    return x**2 / (math.sqrt(x**2 + 1) + 1)

print(f"{'x':>12} {'f(x)':>20} {'g(x)':>20} {'różnica':>20}")

for k in range(1, 11):
    x = 8**(-k)
    fx = f(x)
    gx = g(x)
    print(f"{x:12.5e} {fx:20.15e} {gx:20.15e} {(fx-gx):20.15e}")
```

```
          x                 f(x)                  g(x)              różnica
1.25000e-01 7.782218537318641e-03 7.782218537318706e-03 -6.505213034913027e-17
1.56250e-02 1.220628628286757e-04 1.220628628287590e-04 -8.328027937404281e-17
1.95312e-03 1.907346813823096e-06 1.907346813826566e-06 -3.469446951953614e-18
2.44141e-04 2.980232194360610e-08 2.980232194360612e-08 -1.323488980084844e-23
3.05176e-05 4.656612873077393e-10 4.656612871993190e-10 1.084202172485504e-19
3.81470e-06 7.275957614183426e-12 7.275957614156956e-12 2.646977960169689e-23
4.76837e-07 1.136868377216160e-13 1.136868377216096e-13 6.462348535570529e-27
5.96046e-08 1.776356839400250e-15 1.776356839400249e-15 1.577721810442024e-30
7.45058e-09 0.000000000000000e+00 2.775557561562891e-17 -2.775557561562891e-17
9.31323e-10 0.000000000000000e+00 4.336808689942018e-19 -4.336808689942018e-19
```

Co się stanie?

Dla większych wartości x obie funkcje dadzą niemal identyczne wyniki.

Jednak gdy x staje się bardzo małe (np. 8^{-8}, 8^{-9}, 8^{-10}):

- wartości f(x) zaczynają tracić dokładność,
- g(x) pozostaje stabilne numerycznie.

Dlaczego?

Dla bardzo małych x:

$$\sqrt{x^2+1} \approx 1$$

W funkcji:

$$f(x) = \sqrt{x^2+1} - 1$$

odejmujemy dwie prawie równe liczby:

$$1.000000000000\ldots - 1$$

Powoduje to zjawisko:

> **katastrofalnej utraty cyfr znaczących (catastrophic cancellation)**

W wyniku tracimy znaczną część dokładności.

Natomiast funkcja:

$$g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$$

nie zawiera odejmowania prawie równych liczb, więc jest znacznie stabilniejsza numerycznie.

Wniosek

- Algebraicznie: (f(x) = g(x))
- Numerycznie: dla małych x funkcja g(x) daje dokładniejsze wyniki.
- Powód: w f(x) występuje utrata cyfr znaczących.
- Jest to klasyczny przykład niestabilności numerycznej.

Dla bardzo małych x funkcja g(x) powinna być używana zamiast f(x).