Zestaw 2

1. Mamy listę, która może zawierać różne typy, na przykład inną listę, ale również **krotkę**, **słownik**. Dodaj element o kolejnej wartości w **najbardziej zagnieżdżonej liście**. Poprzez zwiększenie poziomu zagnieżdżenia rozumiemy wejście do kolejnego zagnieżdżenia – listy, krotki, słownika. W słowniku jeśli wartością będzie lista, to jej elementy należy traktować jak jeszcze bardziej zagnieżdżone. Napisz program, który zrobi to uniwersalnie, dla dowolnego zagnieżdżenia. Dla [1, [2, 3], 4] mamy [1, [2, 3, 4], 4], dla [3, 4, [2, [1, 2, [7, 8], 3, 4], 3, 4], 5, 6, 7] powinno być [3, 4, [2, [1, 2, [7, 8, 9], 3, 4], 3, 4], 5, 6, 7]. Jeżeli największe zagnieżdżenie na danym poziomie się **powtórzy**, to należy dodać w obu zagnieżdżeniach, czyli dla [1, [3], [2], []] należy uzyskać [1, [3, 4], [2, 3], [1]]. Przykład bardziej złożony: [1, 2, [3, 4, [5, {'klucz': [5, 6], 'tekst': [1, 2]}], 5], 'hello', 3, [4, 5], (5, (6, (1, [7, 8])))]. Tutaj na takim samym, największym poziomie zagnieżdżenia, są listy będące wartościami w słowniku (listy [5, 6], [1, 2]) a także zagnieżdżona w krotkach (lista [7, 8]) i do to do nich powinien zostać dodany kolejny element. Zatem oczekiwane jest: [1, 2, [3, 4, [5, {'klucz': [5, 6, 7], 'tekst': [1, 2, 3]}], 5], 'hello', 3, [4, 5], (5, (6, (1, [7, 8, 9])))].

   Wymagania formalne Użyć plik `ZADANIE1/zadanie1.py` w repozytorium GitHub Classroom do uzupełnienia swoim kodem. Nie zmieniać nazwy funkcji. Testowane będzie działanie funkcji `dodaj_element(wejscie)`, która zwraca „wyjście” zmodyfikowane jak powyżej opisano.

   wejście: `[[], [[], []]]` **wyjście:** `[[], [[1], [1]]]`
   wejście: `[1, 2, 3, 4]` **wyjście:** `[1, 2, 3, 4, 5]`

2. Napisać program konwertujący liczby zapisane w systemie rzymskim (wielkimi literami `I`, `V`, `X`, `L`, `C`, `D`, `M`) na liczby arabskie w zakresie liczb `1-3999`, i odwrotnie. Proszę **skontrolować poprawność** danych wejściowy, zarówno w formacie arabskim (czyli, że podano wartość całkowitą z przedziału od `1` do `3999`), jak również w formacie rzymskim. Proszę spróbować napisać zwięzły kod, np. bez monstrualnych konstrukcji *if-else*. Uwaga: kontrolę poprawności danych należy zrealizować z pomocą mechanizmu zgłaszania i obsługi wyjątków. Związana z tym część kodu jest pozostawiona w pliku, który należy pobrać z repozytorium i uzupełnić. W Pythonie wyjątki służą do obsługi błędów, które mogą wystąpić podczas wykonywania programu. Wyjątki umożliwiają reagowanie na różne rodzaje błędów w sposób kontrolowany i precyzyjny, zamiast przerywać program.
   - **zgłaszanie wyjątku** – jeśli w programie napotkamy błąd, możemy zgłosić wyjątek, używając instrukcji raise. Przykład:

     ```
     if not (1 <= liczba <= 3999):
         raise ValueError("Liczba musi być w zakresie 1-3999")
     ```

   - **obsługa wyjątków** – aby obsłużyć wyjątki i zapobiec przerwaniu działania programu, używamy bloku try-except. Możemy zdefiniować różne typy wyjątków w sekcji except, żeby obsługiwać konkretne błędy. Przykładowo, gdy rzymskie_na_arabskie("`IIII`") zgłosi ValueError, wyjątek zostanie przechwycony i wyświetlony komunikat „Błąd: Niepoprawny format liczby rzymskiej.”:

     ```
     try:
         wynik = rzymskie_na_arabskie("IIII")  # Niepoprawna liczba rzymska
     except ValueError as e:
         print(f"Błąd: {e}")
     ```

   - **przechwytywanie i dostęp do treści wyjątku** – możemy przechwycić treść wyjątku za pomocą as, jak w przykładzie powyżej (as e). Pozwala to na odczytanie szczegółów błędu.
   - **typy wyjątków** – Python oferuje wiele wbudowanych wyjątków. Najczęściej używane to:
     ValueError: gdy wartość ma nieprawidłowy format (np. liczba rzymska "`IIII`");
     TypeError: gdy typ danych jest niezgodny (np. dodawanie liczby całkowitej do łańcucha);
     IndexError: gdy indeks wykracza poza zakres listy.

- **obsługa wyjątków w testach** – w `pytest` możemy przetestować wyjątki za pomocą `pytest.raises`, co pozwala sprawdzić, czy dany kod zgłasza oczekiwany wyjątek. Przykładowo, blok with pytest.raises(ValueError): sprawdza, czy rzymskie_na_arabskie("ABCD") zgłasza ValueError. Jeśli wyjątek nie zostanie zgłoszony, test zakończy się niepowodzeniem.

```
import pytest

def test_konwersja_nieprawidłowe_symbole():
    with pytest.raises(ValueError):
        rzymskie_na_arabskie("ABCD")  # Niepoprawne znaki
```

Uwaga dotycząca konstrukcji „with”: instrukcja `with` w Pythonie jest używana do zarządzania kontekstem, czyli zasobami, które wymagają specjalnego sposobu otwierania i zamykania. Dzięki niej można bezpiecznie pracować z zasobami (np. plikami, połączeniami sieciowymi, blokami kodu z wyjątkami) bez konieczności ręcznego zamykania ich. Gdy tylko kod wewnątrz bloku with zostanie wykonany (niezależnie od tego, czy zakończył się pomyślnie, czy przez wyjątek), Python automatycznie wykonuje operacje czyszczące, np. zamyka plik lub kończy połączenie.

Wymagania formalne Użyć plik `ZADANIE2/zadanie2.py` w repozytorium GitHub Classroom do uzupełnienia swoim kodem. Nie zmieniać nazw funkcji i wykorzystać fragmenty kodu do zgłaszania wyjątków w przypadku niepoprawnych danych. Testowane będzie działanie funkcji `rzymskie_na_arabskie(rzymskie)` i `arabskie_na_rzymskie(arabskie)`, których liczba rzymska jest w postaci łańcucha znakowego `str`, a liczba arabska w postaci typu `int`.

wejście: `rzymskie_na_arabskie("MCMXCIV")` **wyjście:** `1994`
wejście: `arabskie_na_rzymskie(1994)` **wyjście:** `MCMXCIV`

3. Program ma za zadanie pobrać z publicznego API Wikipedii (język polski) sto losowych krótkich opisów artykułów i przeanalizować tekst każdego z nich. Dla każdego wylosowanego wpisu należy odczytać pola `title` (tytuł artykułu) oraz `extract` (jego streszczenie). Tytuły służą jedynie do wyświetlania bieżącego postępu - program w czasie działania ma wyświetlać aktualnie przetwarzane hasło w ramce o stałej szerokości, z lewej i prawej strony ograniczonej nawiasami kwadratowymi. Po zakończeniu losowań należy wypisać podsumowanie, które obejmuje łączną liczbę słów dłuższych niż trzy znaki, liczbę unikalnych słów oraz listę piętnastu najczęściej występujących wyrazów.

   W kodzie, częściowo przygotowanym, są funkcje do napisania. Funkcja `selekcja` ma z danego tekstu wydobywać wszystkie słowa złożone wyłącznie z liter (w tym polskich), zapisane małymi literami i dłuższe niż trzy znaki. Do wyszukiwania słów służy wyrażenie regularne `WORD_RE`, zdefiniowane w programie jako:

```
WORD_RE = re.compile(r"[^\W\d_]+", re.UNICODE)
```

   Wyrażenie to dopasowuje ciągi znaków, które są literami w rozumieniu Unicode, ale nie cyframi ani znakiem podkreślenia. Składnia `[^\W\d_]` oznacza: „dopasuj wszystko, co nie jest ani znakiem niewyrazowym (`\W`), ani cyfrą (`\d`), ani podkreśleniem (_)”. W efekcie otrzymujemy wyłącznie litery, również z polskimi znakami diakrytycznymi, a dzięki + na końcu - całe wyrazy, nie pojedyncze znaki. Flaga `re.UNICODE` sprawia, że klasy znaków (`\w`, `\W`, `\d`) rozumiane są zgodnie ze standardem Unicode, więc wyrażenie działa poprawnie dla polskich liter.

   W funkcji `ramka` należy sformatować napis tak, by mieścił się w ramce o ustalonej szerokości (`width`, domyślnie 80). Treść powinna być wyśrodkowana w polu o szerokości `width-2`, otoczona nawiasami kwadratowymi. Jeśli tekst jest dłuższy niż `width-2`, należy go przyciąć do `width-3` i zakończyć znakiem wielokropka … (U+2026). Przykład: `ramka("Kot", width=10)` zwróci `[ Kot ]`. W funkcji `main` program ma w każdej iteracji pobierać tytuł artykułu, tworzyć ramkę z nazwą hasła (drukowaną w jednej linii

z użyciem `\r`, aby odświeżała się w miejscu) oraz analizować tekst opisu (`extract`), przekazując go do funkcji `selekcja`.

Po zakończeniu pętli należy wypisać na ekranie podsumowanie: ile wpisów zostało pobranych, ile łącznie słów (dłuższych niż trzy znaki) wystąpiło oraz ile z nich było unikalnych. Następnie należy wyświetlić piętnaście najczęściej pojawiających się słów w kolejności malejącej według częstości.

W tym zadaniu warto poznać klasę `Counter` z modułu `collections` (https://docs.python.org/3/library/collections.html#collections.Counter), która ułatwia zliczanie elementów w iterowalnych strukturach, takich jak listy, łańcuchy znaków czy krotki. Obiekt `Counter` działa podobnie do słownika, w którym klucze to unikalne elementy, a wartości to liczba ich wystąpień. Przykład użycia:

```
from collections import Counter
tekst = "abrakadabra"
licznik = Counter(tekst)
print(licznik)              # Counter({'a': 5, 'b': 2, 'r': 2, 'k': 1, 'd': 1})
print(licznik.most_common(2))# [('a', 5), ('b', 2)]
```

W naszym zadaniu obiekt `licznik` (`Counter`) posłuży do zliczania słów pojawiających się w streszczeniach artykułów Wikipedii. Na koniec wystarczy skorzystać z metody `most_common(15)`, aby wypisać piętnaście najczęstszych słów wraz z ich liczebnością.

Wymagania formalne Należy uzupełnić kod w pliku `ZADANIE3/zadanie3.py` w repozytorium GitHub Classroom, implementując funkcje `selekcja(text)` i `ramka(text, width)` oraz brakujące fragmenty w funkcji `main()`. Do pobierania danych wykorzystujemy bibliotekę `requests` i adres https://pl.wikipedia.org/api/rest_v1/page/random/summary. Każde wywołanie zwraca w formacie JSON losowy wpis zawierający m.in. pola `title` i `extract`. W programie zastosowano nagłówki HTTP (HEADERS) z identyfikatorem użytkownika, aby uniknąć odrzucenia żądań przez serwer. Pobieranie odbywa się w pętli z zabezpieczeniem na wypadek błędu lub przekroczenia limitu czasu (`timeout`). Gdy wystąpi wyjątek, program chwilowo się zatrzymuje (`time.sleep(0.1)`) i próbuje ponownie, aż do skutecznego pobrania stu rekordów. W repozytorium znajdują się również testy sprawdzające poprawność działania programu. Testy weryfikują zarówno pojedyncze funkcje (selekcja i ramka), jak i zachowanie całego programu w funkcji `main`. Dla `selekcja` sprawdzane jest, czy poprawnie wybiera tylko słowa złożone z liter, konwertuje je do małych liter i odrzuca te krótsze niż cztery znaki. Funkcja `ramka` jest testowana pod kątem prawidłowego centrowania, zachowania stałej szerokości oraz poprawnego przycinania z dodaniem znaku wielokropka. Testy `main()` nie korzystają z internetu - zamiast rzeczywistych zapytań do Wikipedii używają mechanizmu *monkeypatch* z biblioteki `pytest`, który podstawia sztuczne dane (fikcyjne odpowiedzi z polami `title` i `extract`). Dzięki temu możliwe jest sprawdzenie logiki programu, zliczania słów i formatowania wyników bez wykonywania faktycznych żądań sieciowych.

4. Przygotować program, który wylicza numerycznie wartość liczby pi metodą prostokątów, a jednocześnie pozwala zaobserwować różnicę w wydajności między klasycznym interpreterem Pythona z blokadą GIL, a nową wersją Pythona 3.14, w której wprowadzono tryb free-threaded. Jest to pierwsza w historii wersja języka, w której możliwe jest rzeczywiste równoległe wykonywanie wątków na wielu rdzeniach procesora, również w zadaniach obciążających CPU.

   W klasycznym Pythonie wszystkie wątki współdzieliły jeden globalny interpretator, chroniony przez Global Interpreter Lock (GIL). W rezultacie nawet jeśli program tworzył kilka wątków, tylko jeden z nich wykonywał kod Pythona w danej chwili. Rozwiązanie to ułatwiało zarządzanie pamięcią i licznikami referencji, lecz całkowicie blokowało równoległe przetwarzanie obliczeń numerycznych.

W wersji 3.14 pojawiła się nowa możliwość uruchamiania Pythona w trybie free-threaded, w którym GIL jest wyłączony. Dzięki temu kilka wątków może wykonywać kod bajtowy CPythona jednocześnie, a praca procesora może być rozłożona na wiele rdzeni.

W trybie free-threaded Python 3.14 zapewnia pełną zgodność z dotychczasowym modelem pamięci Pythona. Każdy obiekt jest nadal chroniony przez licznik referencji, a interpreter gwarantuje, że operacje o charakterze atomowym w języku (na przykład przypisanie pojedynczej referencji, inkrementacja licznika, zapis do pojedynczej zmiennej obiektowej) pozostają wewnętrznie bezpieczne. Python nie zapewnia natomiast automatycznej synchronizacji dla operacji złożonych, takich jak równoczesne modyfikacje list, słowników czy obiektów użytkownika z kilku wątków. Oznacza to, że proste przypisania i odczyty są zawsze spójne i odporne na uszkodzenie pamięci, ale jeśli dwa wątki jednocześnie modyfikują tę samą strukturę danych, należy wprowadzić synchronizację na poziomie aplikacji (np. `threading.Lock`). Interpreter nie dopuszcza sytuacji wyścigu, w której mogłoby dojść do uszkodzenia wewnętrznych struktur obiektów Pythona – każda taka operacja jest chroniona mikrolockiem lub sekcją krytyczną na poziomie C API.

Innymi słowy, Python gwarantuje bezpieczeństwo pamięci, ale nie gwarantuje semantycznej izolacji. Kod z błędnym współdzieleniem danych nie doprowadzi do awarii interpretera, lecz może produkować błędne wyniki, jeśli nie zastosuje się mechanizmów synchronizacji. To jest spójne z ogólną zasadą Pythona: interpreter chroni swoje struktury przed uszkodzeniem, ale nie przejmuje odpowiedzialności za poprawność logiki aplikacji wielowątkowej.

W praktyce oznacza to, że w zadaniu takim jak to – gdzie każdy wątek akumuluje wynik lokalnie i wykonuje jeden końcowy zapis do własnego slotu w liście wyników – nie potrzeba żadnych dodatkowych blokad. Każdy zapis dotyczy innego elementu listy i nie występuje współbieżny dostęp do tego samego obiektu. Python gwarantuje w tym wypadku całkowite bezpieczeństwo i deterministyczne działanie kodu, zarówno w klasycznym interpreterze z GIL, jak i w wersji free-threaded.

Na Windows i macOS Python.org udostępnia dwa gotowe instalatory: klasyczny i free-threaded (oznaczony literą `t` w nazwie pliku wykonywalnego). Na Linuxie dystrybucje zwykle dostarczają tylko klasyczny interpreter, natomiast tryb free-threaded trzeba zbudować samodzielnie lub pobrać z gałęzi developerskiej CPythona. Standardowy interpreter ma nadal aktywny GIL i może być uruchamiany poleceniem `py -3.14`. Wersja free-threaded instalowana jest dodatkowo, oznaczona literą „t”, i uruchamiana poleceniem `py -3.14t`. W obu przypadkach interpreter jest tym samym CPythonem, różni się jednak konfiguracją blokad wewnętrznych.

Popatrzmy jak można to sprawdzić, na przykładzie kodu (początek funkcji `main`).

```
print(f"Python: {sys.version.split()[0]}  (tryb bez GIL? {getattr(sys, '_is_gil_enabled', lambda: None)() is False})")
```

Mamy tu *f*-string, czyli formatowany napis, w którym fragmenty w nawiasach klamrowych są obliczane w locie. `sys.version` zwraca pełny opis wersji Pythona, na przykład coś w rodzaju `3.14.0 (tags/v3.14.0:...) [MSC v.1938 64 bit (AMD64)]`. Dzielimy ten napis na fragmenty metodą `.split()` po spacji i bierzemy element o indeksie `0`, więc dostajemy sam numer wersji. To daje czytelny prefiks raportu wydajności w stylu `Python: 3.14.0`.

Druga część nawiasów to pytanie „tryb bez GIL?” i po niej wyznaczana odpowiedź logiczna. Używamy funkcji `getattr(sys, '_is_gil_enabled', lambda: None)`. `getattr` próbuje pobrać z modułu `sys` atrybut `_is_gil_enabled`. W Pythonie 3.14 ta funkcja istnieje i zwraca wartość logiczną: `True` jeśli GIL

jest włączony, `False` jeśli interpreter działa w trybie free-threaded. W starszych wersjach takiej funkcji w ogóle nie było, dlatego podajemy trzeci argument do `getattr`: awaryjną funkcję `lambda: None`. Dzięki temu nawet w Pythonie 3.13 wywołanie nawiasów `(...)()` nie spowoduje błędu, tylko zwróci `None`. Całe wyrażenie kończy się porównaniem `is False`. Jeśli `_is_gil_enabled()` istnieje i zwróciło `False` - co oznacza uruchomienie interpretera bez GIL - wynik porównania będzie `True` i na ekran trafi „`tryb bez GIL? True`". Jeśli `_is_gil_enabled()` zwróci `True` - klasyczny interpreter z GIL - to porównanie da `False` i zobaczymy „`tryb bez GIL? False`". Jeśli funkcji nie ma i podstawiliśmy `lambda: None`, to `None is False` też jest `False`, więc w Pythonie 3.13 użytkownik dostanie logicznie spójną odpowiedź: nie jest to tryb bez GIL. Ten sposób jest odporny na wersję i nie wymaga if-ów zależnych od numeru wersji Pythona. Warto też zauważyć, że nazwa zaczyna się od podkreślenia. To konwencja sygnalizująca „API wewnętrzne". W 3.14 jest to oficjalnie dostępne, ale należy traktować to wywołanie jako pewne obecnie dostępne narzędzie diagnostyczne.

Druga linia zwraca liczbę logicznych jednostek wykonawczych widocznych dla procesu:

```
print(f"Liczba rdzeni logicznych CPU: {os.cpu_count()}")
```

To oznacza, że na maszynie z Hyper-Threadingiem lub SMT (Simultaneous Multithreading) dostaniemy liczbę wątków sprzętowych, niekoniecznie liczbę fizycznych rdzeni. W praktyce właśnie ta wartość jest sensowna jako maksymalna liczba wątków roboczych w tym zadaniu, bo odzwierciedla realną liczbę równoległych kontekstów wykonania, które system może przydzielić. W rzadkich sytuacjach `os.cpu_count()` może zwrócić `None` - na przykład jeśli platforma nie potrafi tego policzyć - dlatego w kodzie szablonu zazwyczaj stosuje się konstrukcję ochronną typu `os.cpu_count() or 4`, która gwarantuje sensowną wartość domyślną.

W tym zadaniu będziemy chcieli zmierzyć czasy dla różnych konfiguracji wątków i porównać je między klasycznym 3.14 a free-threaded 3.14t. Powyższe dwie linie tworzą nagłówek eksperymentu: zapisują, którą wersję interpretera uruchomiono i czy GIL jest aktywny, oraz ile logicznych jednostek ma do dyspozycji *scheduler*. Dzięki temu wynik czasowy można później zinterpretować poprawnie. Przykład: jeśli uruchomisz `py -3.14 zadanie4.py`, zobaczysz tryb bez GIL? False, a skala czasów dla 2 i 4 wątków będzie zbliżona do jednego wątku. Jeśli uruchomisz `py -3.14t zadanie4.py`, powinno pojawić się tryb bez GIL? True i czasy powinny spadać wraz ze wzrostem liczby wątków mniej więcej proporcjonalnie do liczby logicznych rdzeni zgłoszonych przez `os.cpu_count()`.

Wątki w Pythonie

W Pythonie wątki tworzy się z użyciem klasy `threading.Thread`. Obiekt wątku powstaje po wywołaniu konstruktora z parametrem `target`, który wskazuje funkcję mającą zostać wykonana, oraz `args`, które zawierają argumenty przekazywane do tej funkcji. Utworzenie obiektu wątku nie powoduje jeszcze rozpoczęcia jego pracy. Aby wątek faktycznie się uruchomił, należy wywołać metodę `start()`. W tym momencie interpreter tworzy wątek systemowy, przekazuje mu funkcję docelową i natychmiast wraca do głównego programu. Oznacza to, że `start()` nie blokuje wykonania programu, a wszystkie wątki działają równolegle (lub współbieżnie w wersjach z GIL).

Aby poczekać na zakończenie wątku, należy wywołać metodę `join()`. Metoda `join()` blokuje bieżący wątek (zazwyczaj wątek główny) do momentu, aż dany wątek zakończy działanie. W praktyce najczęściej stosuje się dwie pętle: jedną uruchamiającą wszystkie wątki poprzez `start()`, i drugą wywołującą `join()` dla każdego z nich, aby program zakończył się dopiero po ich ukończeniu.

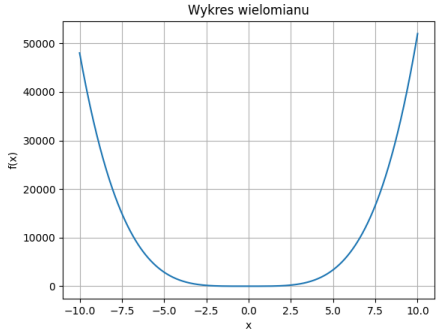
Należy przygotować program, który oblicza przybliżenie liczby π z wykorzystaniem numerycznej całki $\pi = \int_0^1 \frac{4}{1+x^2} dx$ z zastosowaniem reguły punktu środkowego. Wartość całki przybliżamy sumą $krok = \sum_{i=0}^{n-1} \frac{4}{1+x_i^2}$, gdzie $x_i = (i + 0.5) \cdot krok$, oraz $krok = 1/\text{LICZBA\_KROKOW}$.

Wymagania formalne W pliku `ZADANIE4/zadanie4.py` w repozytorium GitHub Classroom znajduje się szablon programu. Funkcja `policz_fragment_pi(pocz, kon, krok, wyniki, indeks)` ma obliczać częściową sumę dla przedziału `[pocz, kon)` i zapisywać wynik w `wyniki[indeks]`. Argument `wyniki` jest listą typu `list[float]` przekazywaną do wszystkich wątków, ponieważ w Pythonie wątki nie zwracają wyników w sposób bezpośredni. Każdy wątek powinien akumulować sumę lokalnie w zmiennej tymczasowej i dopiero po zakończeniu pętli przypisać ją do swojego miejsca w tej liście. Dzięki temu nie występuje problem współbieżnego dostępu do tych samych danych.

W funkcji `main()` należy przygotować zmienne i wielowątkowy przebieg obliczeń. Dla każdej liczby wątków z listy `LICZBY_WATKOW` trzeba:
- wyznaczyć zakresy `[pocz, kon)` tak, aby pokrywały cały przedział `[0, LICZBA_KROKOW)`,
- utworzyć listę obiektów `threading.Thread` z odpowiednimi argumentami,
- uruchomić wątki metodą `start()`,
- poczekać na ich zakończenie, czyli również wywołać w pętli na każdym wątku metodę `join()`,
- zsumować wartości częściowe i obliczyć końcowy wynik pi,
- zmierzyć czas obliczeń i obliczyć przyspieszenie względem czasu dla jednego wątku.

Do pomiaru czasu należy użyć funkcji `time.perf_counter()`. Celem zadania jest również porównanie czasu obliczeń w zależności od liczby wątków i sprawdzenie, jak zmienia się wydajność w klasycznym interpreterze z GIL i w nowym interpreterze free-threaded. Kod powinien być zgodny z załączonymi testami jednostkowymi i działać poprawnie zarówno w interpreterze klasycznym, jak i free-threaded.

5. Python jest językiem, w którym przy użyciu kodu o niewielkiej długości, ale z umiejętnym użyciem bibliotek, da się osiągnąć interesujące wyniki. Celem zadania jest użycie biblioteki do rysowania (`matplotlib`) oraz biblioteki `numpy`. Napisać prosty program, który pozwoli na podanie wielomianu funkcji `f(x)` jako danej wejściowej (łańcuch znakowy) oraz przedział `x` (od `x_min`, do `x_max`). Narysować ten wielomian za pomocą `plt.plot(x_val, y_val)` (gdzie `x_val` i `y_val` to będą, odpowiednio, tablica wygenerowana za pomocą `x_val = np.linspace(x_min, x_max, 200)`, a tablica `y_val` wyliczona z użyciem funkcji `eval()` dla wartości z tablicy `x_val`. Przykładowo, jeśli wpiszę na wejściu: `5*x**4 + 2*x**3 - x + 6` i podam `-10, 10`, to otrzymam:



Proszę, tak jak na rysunku przykładowym, dodać **podpis osi X i Y**, jakiś **tytuł**, oraz aktywować „grid lines”. Uwaga: program ma czytać i dekomponować dane wejściowe podane przez `input()` w formacie: najpierw przepis funkcji `f(x)`, następnie przecinek i dwie wartości będące przedziałem `x` oddzielone spacją, czyli przykładowo: `x**3 + 3*x + 1, -10 10`. Natomiast sprawdzana w testach będzie krotka z dwiema skrajnymi wartościami danej funkcji, czyli w tym przykładzie `(f(-10), f(10))`, czyli `(-1029.0, 1031.0)`.

Funkcja `eval()` jest potencjalnie niebezpieczna, poza tym ma ograniczone możliwości *parsowania*. Zatem jako uzupełnienie do powyższego, proszę również w kodzie spróbować zastosować bibliotekę SymPy ↳ https://pypi.org/project/sympy/ (instalujemy tradycyjnie, `pip install sympy`). Aby możliwa była identyfikacja zmiennej matematycznej we wzorze, np. `x`, najpierw należy określić jej nazwę za pomocą funkcji `symbols('x')`, a następnie dowolny łańcuch znakowy, reprezentujący matematyczny wzór, może zostać przekonwertowany do obiektu SymPy. W takiej postaci możliwe są matematyczne operacje. Można policzyć pochodną, albo całkę (oznaczoną i nieoznaczoną). Przykład:

```
>>> from sympy import symbols, sympify, integrate
>>> x = symbols('x')
>>> wzor = sympify("sin(x)**2 + x**2 + 3*x + 1") # to jest obiekt SymPy
>>> print(wzor)
x**2 + 3*x + sin(x)**2 + 1
>>> wzor.diff() # pochodna
2*x + 2*sin(x)*cos(x) + 3
>>> integrate(wzor) # całka nieoznaczona
x**3/3 + 3*x**2/2 + 3*x/2 - sin(x)*cos(x)/2
>>> integrate("4/(1 + x*x)", (x, 0, 1)) # całka oznaczona
pi
```

W kolejnym kroku trzeba użyć funkcję `lambdify()`, konwertującą wyrażenie symboliczne na funkcję numeryczną, którą wyliczamy wartości `y` z podanych wartości `x`, dokładnie tak samo jak powyżej, zatem można również ją narysować. Przykład użycia (możliwy inny backend niż numpy):

```
>>> funkcja = lambdify(x, wzor, 'numpy')
>>> funkcja(1.23)
7.091185141766646
```

Zatem, nie wypisując nic więcej, proszę w programie narysować również – z tego samego `input()` funkcję „opracowaną” za pomocą SymPy.

Wymagania formalne Użyć plik `ZADANIE5/zadanie5.py` w repozytorium GitHub Classroom do uzupełnienia swoim kodem. Nie zmieniać nazw funkcji, w kodzie są pomocne fragmenty kodu pokazujące co mają zwracać funkcje. Testowane będzie działanie funkcji `rysuj_wielomian(wejscie)`, która zwraca parę wartości (pierwsza i ostatnia komórka) return `y_val[0]`, `y_val[-1]`. Natomiast program będzie też uruchomiony i oceniony wizualnie, czy również `rysuj_wielomian_sympy(wejscie)` skutkuje narysowaniem takiego wykresu – może być bardziej skomplikowany niż wielomian, przykład jest w pliku, ale można poeksperymentować. Ciekawostka: liczby wyjściowe porównywane są z określoną dokładnością, pozwala na to jedna z metod: `pytest.approx((-27.0, 1.0), abs=0.1)`
wejście: `x**3 - 3*x**2 + 3*x - 1, -2 2` **wyjście:** `(-27.0, 1.0)`