Zestaw 3

1. Celem zadania jest przygotowanie programu, który dla podanego n wyznacza wartość n-tego wyrazu ciągu Fibonacciego trzema sposobami: (1) iteracyjnie, (2) metodą macierzową (szybkie potęgowanie macierzy 2×2), (3) z użyciem biblioteki gmpy2. W funkcji main() należy wykonać pomiar czasu każdej metody oraz zapisać trzy wyniki do trzech plików tekstowych.

(1) Ciąg Fibonacciego jest zdefiniowany rekurencyjnie: $F_0 = 0$, $F_1 = 1$, a dla $n \geq 2$: $F_n = F_{n-1} + F_{n-2}$. Zależność ta ma wiele interpretacji numerycznych i geometrycznych. W tym zadaniu skupimy się na obliczeniach efektywnych.
Biblioteka gmpy2 dostarcza zoptymalizowanych funkcji arytmetycznych wysokiej precyzji, opartych na bibliotece GMP. Poza funkcją fib(n), która błyskawicznie zwraca n-ty wyraz ciągu Fibonacciego, znajdują się tam m.in. funkcje do obliczeń modularnych (powmod, invert), sprawdzania pierwszości (is_prime), oraz obsługi dużych liczb zmiennoprzecinkowych i ułamków (mpz, mpq, mpfr).

(2) Formalizm macierzowy (można poczytać na https://pl.wikipedia.org/wiki/Ci%C4%85g_Fibonacciego)
Podstawowe równanie macierzowe dla ciągu Fibonacciego ma postać:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Macierz ta przesuwa kolejne wyrazy ciągu o jeden krok:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} F_2 \\ F_1 \end{bmatrix}$$

Indukcyjnie otrzymujemy zależność ogólną:

$$A^n = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Zatem F_n to element o indeksie [0][1] w macierzy A^n.

Aby uniknąć n mnożeń macierzy, stosujemy szybkie potęgowanie: jeśli n jest parzyste, $A^n = \left(A^{n/2}\right)^2$, jeśli nieparzyste, $A^n = A \times A^{n-1}$. Dzięki temu złożoność obliczeniowa spada z O(n) do O(log n). Przykładowo, przeanalizujmy postępowanie dla n = 5. W obliczeniach będziemy korzystać również z macierzy jednostkowej, zwyczajowo oznaczanej I (wielka litera i). Niech na początku mamy takie obiekty: macierz R = I oraz macierz M = A, oraz n = 5. Zapisujemy binarnie wartość n: 101_2. Najmłodszy bit to 1, zatem uwzględniamy A^1, pierwszy krok obliczeń to R = R * M (czyli I * A = A). Następnie M = M * M (czyli A^2). Kolejny krok, dzielimy n przez 2, poprzez przesunięcie bitowe: n >>= 1 dostajemy n = 2, binarnie 10_2. Najmłodszy bit 0, nic nie mnożymy do R, wykonujemy kolejne mnożenie M = M * M i teraz M = A^4. Kolejne przesunięcie bitowe n >>= 1 daje n = 1, czyli bit 1. Stąd mnożenie R = R * M = A * A^4 = A^5, tu mamy już wynik: $A^5 = \begin{bmatrix} 8 & 5 \\ 5 & 3 \end{bmatrix}$, zatem n-ty element to pozycja $[0][1] = 5$. Podobnie dla innych n, jeśli n jest parzyste (na przykład n = 6), to binarnie mamy 110_2, zatem najmłodszy bit 0. Nie mnożymy do R, tylko przechodzimy (na początek R = I, M = A) do liczenia M = M * M = A^2, następnie n = 3 (bo 6 >> 1), czyli binarnie 11_2, zatem ostatni bit 1 i mnożymy R = R * M, czyli tutaj R = I * A^2, oraz M = M * M = A^4, wreszcie n = 1 (bo 3 >> 1) i ostatni bit 1, więc R = R * M, tutaj R = A^2 * A^4 = A^6. Mamy macierz, a w niej wynik, 6-ty wyraz ciągu równy 8.

(3) Biblioteka gmpy2 (należy zainstalować, np. pip install gmpy2) jest interfejsem do bardzo szybkiej biblioteki GMP (GNU Multiple Precision Arithmetic Library), służącej do obliczeń na dużych liczbach całkowitych i zmiennoprzecinkowych. Oprócz funkcji fib(n), która efektywnie wyznacza n-ty wyraz ciągu Fibonacciego, gmpy2 udostępnia m.in. typy liczbowe mpz, mpq, mpfr (liczby całkowite, ułamki, liczby zmiennoprzecinkowe wysokiej precyzji), szybkie potęgowanie modularne (powmod), odwracanie modularne (invert), testy pierwszości (is_prime), generowanie liczb losowych w zadanych zakresach, funkcje trygonometryczne i specjalne działające na liczbach wysokiej precyzji. Dzięki temu świetnie nadaje się do porównywania własnych implementacji algorytmów numerycznych z wersją „biblioteczną", napisaną w zoptymalizowanym C.

Wymagania formalne Użyć plik `ZADANIE1/zadanie1.py`. Program szkieletowy ma następujące funkcje: `def fib_iter(n: int) -> int`, dla punktu (1) ma zwracać F_n obliczone iteracyjnie. Funkcja pomocnicza `def mat_mul(A: tuple[int, int, int, int], B: tuple[int, int, int, int]) -> tuple[int, int, int, int]`, w celu prostego przemnożenia macierzy 2×2 zapisanych jako krotki (a,b,c,d). Wreszcie, funkcja `def mat_pow(M: tuple[int, int, int, int], n: int) -> tuple[int, int, int, int]`, która ma implementować szybkie potęgowanie macierzy (w środku definicja macierzy jednostkowej `R = (1,0,0,1)`, procedura jak opisano powyżej. Realizacja punktu (2) poprzez wywołanie `def fib_matrix(n: int) -> int`, która zwraca F_n metodą macierzową. Program `main()` dodatkowo wywołuje `gmpy2.fib(n)` jako realizacja punktu (3).

Wyniki (dla przykładowo wartości `n = 1_000_000`) należy zapisać do osobnych plików: fib_iter.txt, fib_matrix.txt, fib_gmpy2.txt (po jednym wyniku w każdym pliku, wynik jest długości `208988`, a plik wielkości ponad 204 KiB), sprawdzić, czy wszystkie trzy metody zwracają ten sam wynik liczbowy. Ponieważ wynika jest bardzo dużą wartością `int` (a tym samym, po konwersji, bardzo długim łańcuchem znakowym), potrzebne jest wyłączenie limitu zamiany int → str tak, żeby `len(str(…))` nie zgłaszało wyjątku. W kodzie należy (po zaimportowaniu modułu sys) dopisać:

```
if hasattr(sys, "set_int_max_str_digits"):
       sys.set_int_max_str_digits(0)
```

[**zadanie za 1 pkt**]

2. Zadanie polega na przetworzeniu danych o liniach tramwajowych w Krakowie, wykonaniu analizy statystycznej oraz stworzeniu interaktywnej mapy sieci tramwajowej za pomocą biblioteki `folium` (więcej informacji np. tutaj ↳ https://medium.com/datasciencearth/map-visualization-with-folium-d1403771717).
W pliku `linie_tramwajowe.json` znajdują się dane o 17 liniach tramwajowych w Krakowie wraz z przystankami i ich współrzędnymi GPS (stan na listopad 2025). Struktura danych:

```
{
  "opis": "Linie tramwajowe w Krakowie z przystankami i współrzędnymi GPS (stan: 2025)",
  "liczba_linii": 17,
  "linie": [
    {
      "linia": "1",
      "przystanki": [
        {"nazwa": "Wzgórza Krzesławickie", "lat": 50.094815, "lon": 20.065271},
        {"nazwa": "Jarzębiny", "lat": 50.0920145, "lon": 20.0611215},
        ...
      ]
    },
    ...
  ]
}
```

W języku Python czytanie danych w formacie JSON wykonywane jest z pomocą modułu `json`:

```
import json
with open('linie_tramwajowe.json', 'r', encoding='utf-8') as file:
    data = json.load(file)
```

Dostęp do danych:

`data['linie']` - lista wszystkich linii tramwajowych
`data['linie'][0]['linia']` - numer pierwszej linii (np. "1")
`data['linie'][0]['przystanki']` - lista przystanków dla pierwszej linii
`data['linie'][0]['przystanki'][0]['nazwa']` - nazwa pierwszego przystanku
`data['linie'][0]['przystanki'][0]['lat']` - szerokość geograficzna przystanku
`data['linie'][0]['przystanki'][0]['lon']` - długość geograficzna przystanku

Napisz funkcję `process_tram_data(input_file)`, która wczyta dane z pliku `input_file` (`linie_tramwajowe.json`) i wypisze finalnie na ekranie informacje w formacie: linia X: Y (gdzie Y to liczba przystanków), posortowane po liczbie przystanków (malejąco), a na końcu całkowitą liczbę unikalnych przystanków (przystanki mogą być współdzielone przez różne linie).

Nastepnie, napisz funkcję `create_tram_map(input_file, output_map_file)`, która wczytuje dane z pliku `input_file` (jak poprzednio, `linie_tramwajowe.json`) i stworzy interaktywną mapę używając biblioteki `folium`:

```
import folium
# Stwórz mapę wycentrowaną na Krakowie
m = folium.Map(location=[50.06, 19.95], zoom_start=12, tiles="cartodbpositron")
```

Następnie, narysuje trasy tramwajowe jako linie (PolyLine):
- każda linia ma inny kolor (użyj listy kolorów)
- dodaj tooltip z numerem linii

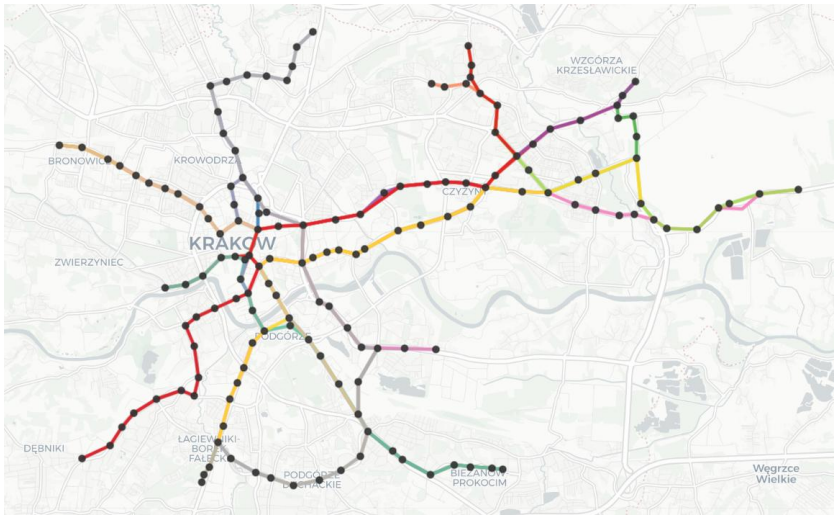
Przykład:

```
folium.PolyLine(
    coordinates,  # lista krotek (lat, lon)
    color=color,
    weight=3,
    opacity=0.9,
    tooltip=f"Linia {line_number}"
).add_to(m)
```

Oznacz przystanki jako punkty (`CircleMarker`):
Tooltip zawierający nazwę przystanku i listę linii, które przez niego przejeżdżają
Przykład: "Rondo Mogilskie – linie: 3, 4, 5, 9, 10, 14, 20, 50, 52"
Jeśli chodzi o paletę kolorów, to można użyć takie coś:

```
kolory = [
    "#e41a1c", "#377eb8", "#4daf4a", "#984ea3", "#ff7f00",
    "#a65628", "#f781bf", "#999999", "#66c2a5", "#fc8d62",
    "#8da0cb", "#e78ac3", "#a6d854", "#ffd92f", "#e5c494",
    "#b3b3b3"
]
```

następnie cyklicznie wybierać kolory z tej puli
`color = kolory[i % len(kolory)]`
Zapisz mapę do pliku HTML o nazwie `mapa_tramwaje.html`: `m.save(output_map_file)`
Przykładowy wygląd mapy:



Wymagania formalne Użyć plik `ZADANIE2/zadanie2.py` w repozytorium GitHub Classroom do uzupełnienia swoim kodem. Funkcja `process_tram_data(input_file)` ma zwrócić słownik ze statystykami: `{numer_linii: liczba_przystanków}` (np. `{1: 28, 3: 24, ...}`) oraz liczbę unikalnych przystanków (jako `int`). Do repozytorium należy wysłać koniecznie również plik z mapą, czyli `mapa_tramwaje.html` [**zadanie za 2 pkt**].

3. Python jest naturalnym językiem do pozyskiwania danych, ich badania i wizualizacji. Celem tego zadania będzie zapoznanie się z bibliotekami `yfinance` i `pandas`. Pierwszy moduł dostarcza dane z notowań giełdowych różnych walorów (z serwisu Yahoo Finance). Zainstalujemy: `yfinance`, `pandas`, `matplotlib`. Należy również zainstalować i uruchomić Jupyter Notebook (na przykład `pip install notebook`, uruchomienie `python -m notebook`), gdyż bardzo ułatwi to poznanie wybranych funkcji i danych. Obsługa Notebooka jest intuicyjna, uruchomienie pola z kodem to skórt Shift-Enter, a jeśli zmienimy jakiś fragment wcześniejszego kodu i chcemy, aby wykonały się wszystkie polecenia poniżej, to z menu u góry wybieramy: Run > Run Selected Cell and All Below.
Zacznijmy od następującej sekwencji komend:
- zróbmy import bibliotek (jak na rysunku powyżej)

```
[1]: import yfinance as yf
     import pandas as pd
     import matplotlib.pyplot as plt
```

```
[2]: tickers = ['BTC-USD']
```

```
[5]: data = yf.download(tickers, start = '2025-11-01', auto_adjust=False)
```

```
[*********************100%***********************]  1 of 1 completed
```

```
[6]: data.head()
```

[6]:

Price	Adj Close	Close	High	Low	Open	Volume
Ticker	BTC-USD	BTC-USD	BTC-USD	BTC-USD	BTC-USD	BTC-USD
Date						
2025-11-01	110064.015625	110064.015625	110574.898438	109372.953125	109558.625000	25871668762
2025-11-02	110639.625000	110639.625000	111167.312500	109523.453125	110064.429688	34284209459
2025-11-03	106547.523438	106547.523438	110764.914062	105336.359375	110646.906250	72852006359
2025-11-04	101590.523438	101590.523438	107264.882812	98962.062500	106541.421875	110967184773
2025-11-05	103891.835938	103891.835938	104534.703125	98989.914062	101579.234375	77584934804

- w drugim polu stwórzmy jednoelementową listę z nazwą interesującego waloru; może to być symbol giełdowy firmy (np. `'AAPL'` to Apple notowane na giełdzie NASDAQ), albo notowanie wybranej kryptowaluty np. `'BTC-USD'` to notowanie Bitcoin do dolara, dane pochodzą od zewnętrznych agregatorów. Opcja `auto_adjust` dotyczy cen automatycznie skorygowanych o dywidendy i splity akcji, co w przypadku walorów kryptowalut nie ma znaczenia (stąd `False`).
- kolejna linia to pobranie danych za pomocą funkcji `download()`; jako argumenty można podać również datę końcową `end='2025-11-20'`, interwał czasowy (domyślnie 1 dzień), `interval='1h'` i inne opcje, np. dotyczące wielowątkowego pobierania danych. Zamiast daty można użyć argumentu `period='1d'` czyli okresu czasu wstecz (uwaga, zaczyna się o północy). Funkcja `download()` zwraca dane w formacie `pandas.DataFrame`, który jest bardzo powszechnym formatem do agregacji, selekcji i wszelkiego rodzaju przeliczeń na stabularyzowanych danych. Przećwiczmy na początek kilka selekcji:

```
data.head() # pierwsze 5 wierszy lub data.head(10) to będzie 10 wierszy
data.tail(8) # ostatnie 8 wieszy lub 5 jeśli bez argumentu
data[1:3] # selekcja wiersza numer 1 i 2 (pamiętajmy, zaczynamy od 0)
data.sample() # losowy wiersz lub losowa liczba wierszy – zobacz co będzie jeśli wpisać np. 100
```

```
[7]: data.head()
     data.tail(8)
     data[1:3]
     data.sample()
```

[7]:

Price	Adj Close	Close	High	Low	Open	Volume
Ticker	BTC-USD	BTC-USD	BTC-USD	BTC-USD	BTC-USD	BTC-USD
Date						
2025-11-13	99697.492188	99697.492188	104005.492188	97988.71875	101674.148438	101546815416

Generalnie widzimy, że tabela z danymi składa się z identyfikatora (tutaj jest to Date), można go użyć, jak również indeksu porządkowego, np. drugi wiersz to będzie (patrz rysunek):

```
[8]: data.loc['2025.11.01']
```

```
[8]: Price      Ticker
     Adj Close  BTC-USD    1.100640e+05
     Close      BTC-USD    1.100640e+05
     High       BTC-USD    1.105749e+05
     Low        BTC-USD    1.093730e+05
     Open       BTC-USD    1.095586e+05
     Volume     BTC-USD    2.587167e+10
     Name: 2025-11-01 00:00:00, dtype: float64
```

```
[9]: data.iloc[1]
```

```
[9]: Price      Ticker
     Adj Close  BTC-USD    1.106396e+05
     Close      BTC-USD    1.106396e+05
     High       BTC-USD    1.111673e+05
     Low        BTC-USD    1.095235e+05
     Open       BTC-USD    1.100644e+05
     Volume     BTC-USD    3.428421e+10
     Name: 2025-11-02 00:00:00, dtype: float64
```

To co widzimy pod nazwami, to kolumny z danymi ceny otwarcia Open (początek danego interwału czasowego, tutaj konkretnego dnia), najwyższa cena w danym odcinku czasu High, najniższa Low, cena zamknięcia Close, skorygowaną cenę zamknięcia Adj Close (ma znaczenie w przypadku akcji, uwzględnia dywidendę, splity akcji), wreszcie wolumen transakcyjny Volume (w tym przypadku wyrażony w USD i dotyczy tylko handlu pary BTC-USD, dla akcji będzie to liczba jednostek). Ograniczmy się do ceny zamknięcia oraz wolumenu, będzie to selekcja: `data[['Close', 'Volume']]`, zwróćmy uwagę na zagnieżdżoną listę w liście, ale możliwe jest też podanie jednej wartości kolumny `data['Close']`. Jeśli chcemy ograniczyć zakres czasowy (czyli po indeksie) tylko do części listopada 2025, napiszemy: `data.loc['2025-11-01':'2025-11-10', ['Close', 'Volume']]`. Te operacje są podstawowymi sposobami selekcji na typie `pandas.DataFrame`. Oczywiście można jawnie skopiować część danych do nowej zmiennej:
`close_prices = pd.DataFrame(data['Close'])`,
albo dwie zmienne:
`cv_prices = pd.DataFrame(data[['Close', 'Volume']])`.

```
[13]: cv_prices['Volume']/cv_prices['Close']
```

[13]:

Ticker	BTC-USD
Date	
2025-11-01	2.350602e+05
2025-11-02	3.098728e+05
2025-11-03	6.837513e+05
2025-11-04	1.092299e+06
2025-11-05	7.467857e+05
2025-11-06	6.311149e+05

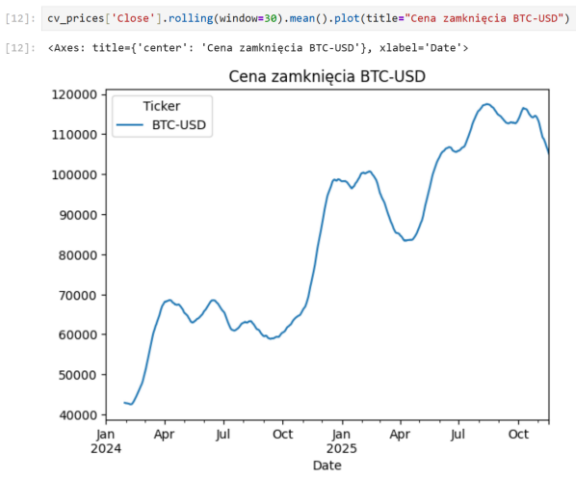
Teraz możemy zastosować na wybranym przedziale danych mnóstwo funkcji. Zacznijmy od najprostszej, czyli sumy: `cv_prices['Volume'].sum()`. Wiemy jednak, że jest to cały wolumen w USD, a jeśli chcemy policzyć liczbę jednostek BTC, przynajmniej w przybliżeniu, należy podzielić wolumen przez cenę. Pandas pozwala na macierzowe dzielenie odpowiednich elementów, spróbujmy (jak na rysunku obok).

Następnie, `(cv_prices['Volume']/cv_prices['Close']).sum()`. W podobny sposób, już bez dzielenia, tylko operując na zmiennej `'Close'` czyli cenie zamknięcia, można pozyskać za pomocą funkcji: `min()` – cenę minimalną, `max()` – cenę maksymalną, `mean()` – cenę średnią. Do dyspozycji jest dużo więcej. Możemy obliczyć średnią kroczącą w zadanym przedziale czasowym (jeden z prostych indykatorów analizy technicznej, gdzie obserwuje się kiedy różne średnie kroczące przecinają się).

Wczytajmy więcej danych (można się cofnąć do właściwej komórki w notatniku), np. od początku roku 2024:
`data = yf.download(tickers, start = '2024-01-01', auto_adjust=False)`
I teraz, na przykład, dla okna 30 jednostek (tutaj dni):
`cv_prices['Close'].rolling(window=30).mean()`.

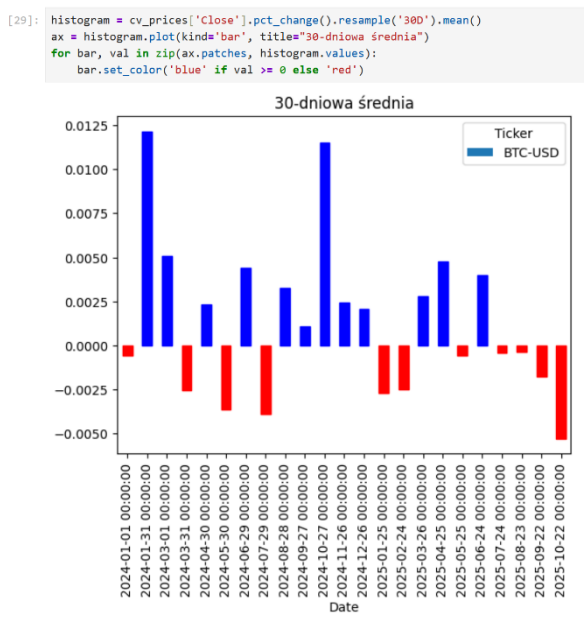
Można wynik przypisać do kolejnej zmiennej, a można też, zwłaszcza w środowisku interaktywnym, jakim jest Jupyter Notebook, od razu narysować wykres (Pandas korzysta tutaj z Matplotlib) – efekt widzimy na poniższym rysunku. W makrze należałoby wywołać dodatkowo `plt.show()`.
W komórce, dokładamy po prostu plot z opcją title:

```
cv_prices['Close'].rolling(window=30).mean().plot(title="Cena zamknięcia BTC-USD")
```

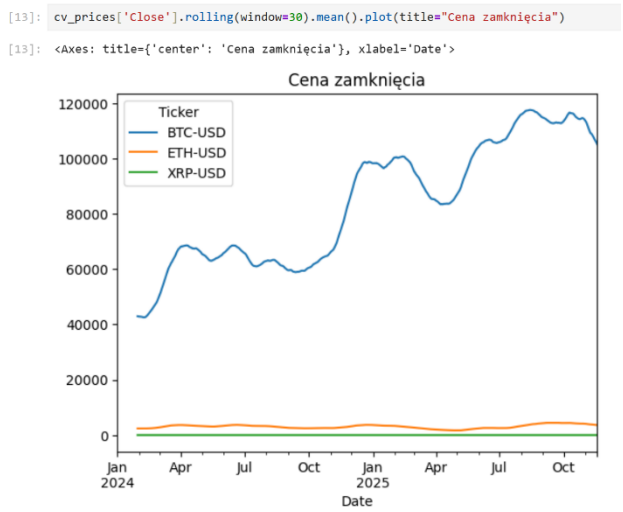


Procentową zmianę możemy uzyskać: `cv_prices['Close'].pct_change()`.
Przedział jednego dnia może być za mały, zatem można „rebinować" taki histogram, a do tego biblioteka Pandas pozwala na wybór dowolnej strategii, jak ma być policzona nowa zawartość. Może to być suma (funkcja `sum()`), lub – minimum, maksimum, średnia, mediana – np. `cv_prices['Close'].resample('30D').median()` oznacza połączenie 30 dni w jedną komórkę, której wartością będzie mediana ceny z 30 dni. Polecam przyjrzeć się poniższemu przykładowi, gdzie dokonano połączenia komórek, wpisania wartości średniej. Następnie `kind='bar'` oznacza wybór rodzaju wykresu, jako słupkowego histogramu, a pętla na końcu to pokolorowanie słupków, gdzie ax.patches to lista wszystkich narysowanych kształtów (tutaj słupków) na wykresie.
Komendy do komórek notatnika:

```
histogram = cv_prices['Close'].pct_change().resample('30D').mean()
ax = histogram.plot(kind='bar', title="30-dniowa średnia")
for bar, val in zip(ax.patches, histogram.values):
    bar.set_color('blue' if val >= 0 else 'red')
```



Zanim przejdziemy do formalnej części zdania, bardzo polecam przerobić te same ćwiczenia, gdy zdefiniujemy więcej symboli, np. `tickers = ['BTC-USD', 'ETH-USD', 'XRP-USD']`, wszystkie kolejne selekcje będą miały trzy kolumny, a wszystko będzie się działo całkowicie automatycznie! Na przykład, po wykonaniu selekcji `cv_prices = pd.DataFrame(data[['Close','Volume']])`, wykresy ceny zamknięcia:

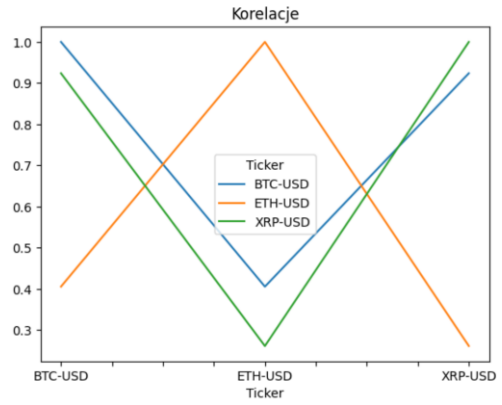


Mając kilka zmiennych (np. notowania trzech kryptowalut), z łatwością można policzyć korelacje pomiędzy parami kryptowalut. Do tego celu służy funkcja `corr()`, która domyślnie używa współczynników korelacji Pearsona (czyli znormalizowana wersja macierzy kowariancji), gdzie doskonała korelacja wynosi +1, a antykorelacja –1. Zatem otrzymujemy taką tabelę:

	BTC-USD	ETH-USD	XRP-USD
BTC-USD	1	0.x	0.y
ETH-USD	0.x	1	0.z
XRP-USD	0.y	0.z	1

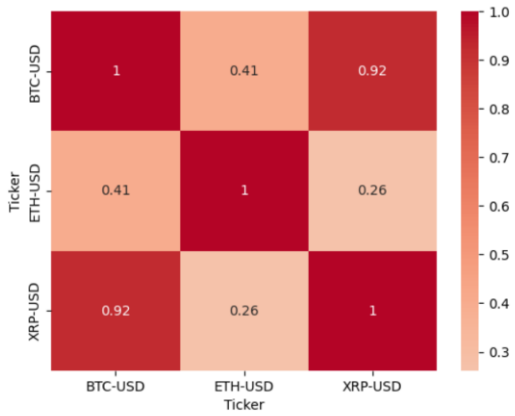
Wartości diagonalne wynoszą oczywiście 1, a pozadiagonalne pokazują korelację pomiędzy parami kryptowalut. Tabela (macierz) jest symetryczna, bo korelacja BTC-ETH jest taka sama dla pola ETH-BTC. Wykonując rysunek, dostaniemy nieco mylący „wykres", który nie powinniśmy rysować łącząc punkty liniami ciągłymi:

```
[14]: cv_prices['Close'].corr().plot(title="Korelacje")
[14]: <Axes: title={'center': 'Korelacje'}, xlabel='Ticker'>
```



Bardziej sensowne byłoby tu narysowanie „szachownicy" odzwierciedlającej współczynnik korelacji. Tutaj warto zainteresować się udoskonaloną wersją myplotlib, a mianowicie biblioteką `seaborn` (pamiętajmy wcześniej ją zainstalować). Argument `annot=True` powoduje wyświetlenie wartości liczbowej współczynnika korelacji. Resztę pozostawimy bez komentarza:

```
[16]: import seaborn as sns
[17]: sns.heatmap(cv_prices['Close'].corr(), annot=True, cmap="coolwarm", center=0)
[17]: <Axes: xlabel='Ticker', ylabel='Ticker'>
```

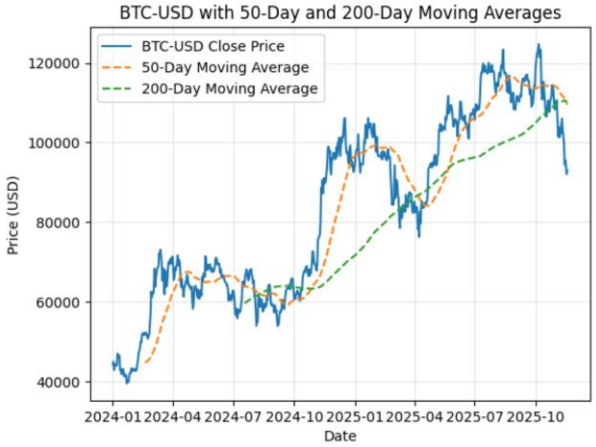


Po zapoznaniu się z tymi możliwościami, należy napisać program (prototypować rozwiązanie można w Jupyter Notebook, ale finalny program `.py` musi zwracać dane, które będą podlegać testowaniu), który:

a) zwróci daty (listę z datami jako łańcuchy znakowe, w formacie `'2024-01-01'`) przecięcia się dwóch średnich kroczących (MA – moving average), jednej 50-dniowej, drugiej 200-dniowej (jest to tzw. „golden cross"), dla ułatwienia, proszę znaleźć takie dni dla danych od początku roku 2024 do 2025-11-20. Oczywiście obie średnie współistnieją dopiero po 200 dniach, obecnie są takie dwa przypadki, ale należy zwrócić wynik w oparciu o bieżące wartości. Zatem po pozyskaniu danych:
`btc_data = yf.download('BTC-USD', start='2024-01-01', end='2025-11-20', auto_adjust=False)`
policzyłbym np.

`btc_data['50-day MA'] = btc_data['Close'].rolling(window=50).mean(),`
tak samo dla średniej 200-dniowej,
`btc_data['200-day MA'] = btc_data['Close'].rolling(window=200).mean()`
następnie szukał gdzie ich różnica zmienia znak i jest mniejsza niż określona wartość progowa – tu trzeba trochę poeksperymentować. Warto najpierw sobie to narysować, na przykład:

```
plt.plot(btc_data.index, btc_data['Close'], label='BTC-USD Close Price')
plt.plot(btc_data.index, btc_data['50-day MA'], label='50-Day Moving Average', linestyle='--')
plt.plot(btc_data.index, btc_data['200-day MA'], label='200-Day Moving Average', linestyle='--')
plt.title('BTC-USD with 50-Day and 200-Day Moving Averages')
plt.xlabel('Date')
plt.ylabel('Price (USD)')
plt.legend()
plt.grid(True, alpha=0.3)
```



Uwaga: należy wypisać wszystkie daty, coś typu: `['2024-01-01', '2024-01-02', '2024-01-03']`.

b) zwróci największą liczbę BTC które sprzedano/kupiono w którymś z dni w badanym zakresie czasu, czyli od początku roku, argument `start='2024-01-01'`, do dnia `end='2025-11-20'`, domyślnym interwałem jest 1d. Reszta operacji jest oczywista, trzeba podzielić wolumen przez cenę. Proszę wynik wydrukować po zrzutowaniu do typu `int`.

Wymagania formalne Użyć plik `ZADANIE3/zadanie3.py` w repozytorium GitHub Classroom do uzupełnienia swoim kodem. Nie zmieniać nazw, testowane będą funkcje `find_crossovers()` i `calculate_total_btc_traded()`, gdzie pierwsza zwraca listę znalezionych dat przecięć średnich kroczących, a druga zwraca liczbę typu int maksymalnego wolumenu handlowanych dziennie BTC, od początku roku 2024 i do dnia 2025-11-20.
Ciekawostka: „Cryptocurrency Prices by Market Cap" w tym 24h Volume: https://www.coingecko.com/
[**zadanie za 2 pkt**]