# The Bank Simulator

Marcin Grabysz

Styczeń 2021

## 1 Uruchomienie programu

Symulator banku zacznie pracę po uruchomieniu modułu `bank_simulator.py`, na przykład poprzez wpisanie w wierszu poleceń komendy:

```
python3 bank_simulator.py
```

Można uruchomić symulator podając opcjonalny argument `--load [path]`, gdzie path to ścieżka pliku zawierającego odpowiednie dane. W takim wypadku bank od początku symulacji będzie posiadał pewną ilość klientów obciążonych odpowiednimi kredytami. Poprawne dane do wykorzystania zawiera przykładowy plik `example_csv_data.txt`.

Symulator daje możliwość udzielania pożyczek nowym klientom, udzielania kolejnych pożyczek już obecnym na liście klientom, przeglądanie bieżących danych oraz posuwanie symulacji do przodu o dowolną (mniejszą niż 10 tysięcy) ilość miesięcy. Po uruchomieniu programu i wybraniu działania, należy postępować zgodnie z poleceniami wyświetlanymi w terminalu.

## 2 Opis modułów

### 2.1 `bank_simulator`

Główny moduł zawierający funkcję `main()` powodującą uruchomienie symulatora. W zależności od tego, czy podany został argument LOAD, funkcja powoduje wczytanie (lub nie) danych z pliku. W przypadku, gdy w pliku zawarte są niepoprawne dane, funkcja zwraca odpowiedni komunikat.

Aby umożliwić uruchomienie funkcji `main()` z opcjonalnymi argumentami, moduł `bank_simulator` importuje biblioteki `argparse` i `sys`.

## 2.2 bank_classes

Moduł zawiera implementację klas: `Bank()`, `Loan()` oraz `Client()`; wyjątków: `InvalidValueError`, `InvalidRateError`, `InvalidInstallmentsError`, `InvalidNameError`, `NoBudgetError`, `ToBigInstallmentsError, YearOutOfRangeError`; oraz funkcji sprawdzających poprawność danych.

**Bank()** obiekt klasy `Bank` stanowi serce programu. Przechowuje informacje o klientach i ich pożyczkach. Posiada metody umożliwiające zwiększanie (zmniejszanie) budżetu, dodawanie nowych klientów oraz pożyczek, spłacanie pożyczek oraz zmieniania bieżącej daty symulacji. Bank posiada również metody informacyjne, zwracające słowniki z bieżącymi danymi dotyczącymi budżetu, klientów lub pożyczek. Bank przechowuje jedynie informacje o klientach, którzy są mu dłużni jakieś pieniądze - po spłacie ostatniej raty, klient jest bezpowrotnie usuwany z listy.

**Loan()** obiekt klasy `Loan` przechowuje informacje o danej pożyczce, takie jak: całkowita wartość, oprocentowanie, ilość pozostałych rat, ilość spłaconych rat, wysokość jednej raty. Podstawową metodą obiektu tej klasy jest funkcja `payment(self)`, która zwraca odpowiednią kwotę (wysokość jednej raty) w zależności od tego, czy rata jest ostania czy nie (patrz: Uwagi).

**Client()** obiekt klasy `Client` przechowuje podstawowe informacje o kliencie - imię oraz ID. ID jest niepowtarzalne dla każdego klienta i służy wskazaniu na konkretny obiekt tej klasy. Imię ma jedynie funkcję informacyjną i może się powtarzać.

Parametry: budget (budżet banku), value (wartość pożyczki), rate (oprocentowanie) są obiektami klasy Decimal. Pozwala to wykonywać na nich precyzyjne obliczenia zwracające wynik oczekiwany przez użytkownika przyzwyczajonego do pracy z liczbami dziesiętnymi.

## 2.3 test_bank_classes

Moduł zawiera testy jednostkowe sprawdzające poprawność metod i funkcji zawartych w module `bank_classes`.

## 2.4 bank_io

Moduł zawierający funkcje służące komunikacji z użytkownikiem.

Funkcje opisane słowem `info` formatują dane dostarczone przez metody klasy `Bank()` i zwracają je w przyjaznym dla użytkownika, gotowym do wyświetlenia formacie

Funkcje `greeting()` i `game_over()` zwracają komunikaty wyświetlane na początku i końcu symulacji

Funkcje `'take_correct...'` służą pobieraniu poprawnych danych od użytkownika.

Funkcje `load_from_file()` oraz `read_from_csv()` umożliwiają czytanie danych z pliku .txt.

## 2.5 test_bank_io

Moduł zawiera testy jednostkowe sprawdzające poprawność funkcji czytającej dane z pliku .txt

Zrezygnowałem z testowania wielu funkcji z modułu `bank_io`, które zwracają wielolinijkowe napisy. Układ takiego stringa prościej jest sprawdzić i ocenić "ocznie" pod kątem estetycznym, natomiast poprawność zawartych danych jest sprawdzana testami modułu `bank_classes`.

## 2.6 bank_interface

Moduł zawiera implementację klasy `Interface()` która odpowiada za interfejs użytkownika.

## 2.7 example_csv_data.txt

Plik zawiera przykładowe dane, z którymi można uruchomić symulator (opcjonalnie). Format danych zawartych w pliku:

```
name,value,rate,installments\n
```

gdzie: name - nazwa klienta, value - całkowita wartość pożyczki, rate - oprocentowanie pożyczki, installments - liczba rat, w których pożyczka ma zostać spłacona.

Aby bank udzielił drugiej pożyczki klientowi, który już wziął jeden kredyt, parametr 'name' należy zastąpić słowem '[previous]' w nawiasach kwadratowych. W takim wypadku pożyczka o odpowiednich parametrach zostanie udzielona ostatniemu utworzonemu klientowi.

### 2.8 `opis_zadania.txt`

Opis zadania otrzymany na rozpoczęcie projektu

# 3 Uwagi

Podczas pracy natrafiłem na możliwy błąd, wynikający z zaokrąglania. Przykładowo, klient ma do spłaty 100 PLN w trzech ratach. Ponieważ może posługiwać się wartościami nie większymi niż jeden grosz, jedna rata wynosi 33.33 PLN. Aby uniknąć straty w wysokości 1 grosza, przyjąłem założenie, że ostatnia rata jest obliczana na inny sposób tak, aby nadrobić wynikłą różnicę. W tym przypadku trzecia rata wyniesie nie 33.33 a 33.34 PLN. W zależności od ilości rat, które zostały do zapłacenia, metoda klasy `Loan self.payment()` zwraca odpowiednią wartość.

Obliczanie wartości pojedynczej raty odbywa się z precyzją 28 miejsc po przecinku, jednak wartość jest zaokrąglana do dwóch miejsc po przecinku przed odjęciem jej od budżetu. W założeniu, bank i klienci mogą posługiwać się wartościami nie mniejszymi niż jeden grosz.

Początkowo, spłacanie pożyczek było obarczone ryzykiem jeszcze jednego, mało uchwytnego (ale niezwykle ciekawego) błędu, który wynikł w fazie końcowego testowania. W przypadku, gdy liczba rat była bardzo duża w stosunku do wartości pożyczki, błąd wynikły z zaokrąglania mógł z powodować, że klient spłacał pożyczkę wcześniej i dalsze płatności powodowały naliczenie mu ujemnego długu. Analiza tego problemu doprowadziła mnie do wniosku, że nawet przy najmniej sprzyjających okolicznościach (błąd zaokrąglenia równy 0,5 grosza; większy nie jest możliwy) błąd na pewno nie wystąpi, jeżeli spełniony zostanie warunek:

$$I^2 < 200 * V$$

Gdzie: I - (Installments) - liczba rat, V - (Value) - wartość pożyczki.

Aby zapobiec temu problemowi, przyjęte zostały (całkiem racjonalne) założenia, że:

- Minimalna wartość pożyczki to 100 PLN
- Liczba rat nie może być większa niż pewna wartość, wyliczana na podstawie podanego V

Przyjęte założenia nie powinny przeszkodzić w używaniu symulatora w "racjonalny" sposób. Proszę zauważyć, że dla ekstremalnie małej pożyczki 100 PLN, maksymalna liczba rat to aż 141 (co przekłada się na ponad 11 lat spłacania)

Po osiągnięciu roku 10000 symulator podnosi odpowieni wyjątek, wyświetla komunikat o tym, że nawet Cyberpunk nie był aż tak opóźniony i kończy działanie. Nie potrafiłem w inny sposób obsłużyć tej niedoskonałości klasy date z biblioteki datetime.