PYTHON
ZORIENTOWANY OBIEKTOWO

PROGRAMOWANIE GIER I GRAFICZNYCH INTERFEJSÓW UŻYTKOWNIKA

IRV KALB

Helion

no starch press

Tytuł oryginału: Object-Oriented Python: Master OOP by Building Games and GUIs

Tłumaczenie: Agnieszka Górczyńska

ISBN: 978-83-283-9406-3

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/pytzor
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

1

Proceduralny styl programowania w Pythonie

W KSIĄŻKACH WPROWADZAJĄCYCH DO PROGRAMOWANIA ZWYKLE STOSUJE SIĘ STYL PROCEDURALNY, W KTÓRYM TO PROGRAM ZOSTAJE PODZIELONY NA WIELE FUNKCJI (NAZYWANYCH RÓWNIEŻ PROCEDURAMI lub podprocedurami). Dane są przekazywane do funkcji, z których każda wykonuje jedno lub więcej obliczeń, a następnie zwraca wynik. Ta książka została poświęcona innemu paradygmatowi, nazywanemu *programowaniem zorientowanym obiektowo*, który pozwala programistom na zastosowanie zupełnie innego podejścia podczas tworzenia oprogramowania. W programowaniu zorientowanym obiektowo dane i kod można ze sobą łączyć w spójnych jednostkach, a tym samym uniknąć pewnych komplikacji, nierozerwalnie związanych ze stylem programowania proceduralnego.

W tym rozdziale przedstawię wiele podstawowych koncepcji Pythona poprzez utworzenie dwóch małych programów, opartych na różnych konstrukcjach tego języka. Pierwszy to prosta gra karciana o nazwie *Większa czy mniejsza*. Drugi to symulacja banku pozwalająca przeprowadzać operacje na jednym, dwóch lub wielu kontach. Oba te programy zostaną zbudowane z użyciem stylu proceduralnego — czyli z wykorzystaniem standardowych technik bazujących na danych i funkcjach. W dalszej części książki te programy będą tworzone ponownie, ale z użyciem technik programowania zorientowanego obiektowo. Celem tego

rozdziału jest pokazanie najczęściej występujących problemów podczas stosowania programowania proceduralnego. Gdy poznasz te problemy, z następnego rozdziału dowiesz się, jak programowanie zorientowane obiektowo pomaga w ich wyeliminowaniu.

Gra karciana

Pierwszy przykład to prosta gra karciana o nazwie *Większa czy mniejsza*. Gracz otrzymuje osiem losowo wybranych kart z talii, pierwsza z nich jest widoczna. Zadaniem gracza jest odpowiedź na pytanie, czy następna z wybranych kart ma większą czy mniejszą wartość niż aktualnie wyświetlona karta. Załóżmy, że aktualna karta ma wartość 3. Gracz przewiduje, że następna będzie miała większą wartość. Jeżeli się okaże, że miał rację, zdobywa punkty. Jeśli natomiast przewidywał, że następna karta będzie miała niższą wartość, oznacza to, że się pomylił, i traci pewną liczbę punktów.

Prawidłowa odpowiedź jest nagradzana 20 punktami, nieprawidłowa zaś — karana utratą 15 punktów. Jeżeli następna karta ma taką samą wartość jak obecna, uznaje się, że gracz się pomylił.

Przedstawienie danych

Program musi przedstawić talię 52 kart, które będą tworzyły listę. Jej każdy element będzie miał postać słownika, to będą pary klucz-wartość. W celu przedstawienia karty słownik wykorzysta trzy pary klucz-wartość: `'rank'`, `'suit'` i `'value'`. Klucz `'rank'` przechowuje nazwę karty (as, 2, 3, …, walet, dama, król). Wartość karty to liczba całkowita używana do porównywania kart (1, 2, 3, …, 10, 11, 12, 13). Przykładowo karta walet trefl zostanie przedstawiona za pomocą takiego słownika:

```
{'rank': 'walet', 'suit': 'trefl', 'value': 11}
```

Zanim gracz rozpocznie rundę, następuje utworzenie listy przedstawiającej talię kart i ułożenie kart w losowo wybranej kolejności. W tym programie karty nie są pokazane w sposób graficzny, więc za każdym razem, gdy gracz wybierze opcję „większa” lub „mniejsza”, program pobiera słownik karty z listy talii, a następnie wyświetla graczowi kartę i jej kolor. Dalej program porównuje wartość nowej karty z poprzednią i wyświetla odpowiedni komunikat na podstawie wyboru dokonanego przez gracza.

Implementacja

Na listingu 1.1. przedstawiłem kod źródłowy omawianej tutaj gry karcianej.

UWAGA *Przypominam, że kod źródłowy wszystkich przykładowych programów omówionych w książce jest dostępny pod adresem https://ftp.helion.pl/przyklady/pytzor.zip. Możesz go pobrać i uruchomić albo samodzielnie wpisać kod źródłowy.*

Listing 1.1. Gra karciana w Pythonie utworzona z zastosowaniem podejścia proceduralnego

Plik: HigherOrLowerProcedural.py

```
# HigherOrLower.
import random

# Stałe przedstawiające karty.
SUIT_TUPLE = ('pik', 'kier', 'trefl', 'karo')
RANK_TUPLE = ('as', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'walet',
'dama', 'król')

NCARDS = 8

# Przekazanie talii. Wartością zwrotną funkcji jest losowo wybrana karta z talii.
def getCard(deckListIn):
    thisCard = deckListIn.pop() # Pobranie jednej karty z góry talii i jej zwrot.
    return thisCard

# Przekazanie talii. Wartością zwrotną funkcji jest talia, w której karty są ułożone losowo.
def shuffle(deckListIn):
    deckListOut = deckListIn.copy()  # Utworzenie kopii talii początkowej.
    random.shuffle(deckListOut)
    return deckListOut

# Kod główny programu.
print('Witaj w grze Większa czy mniejsza.')
print('Musisz odgadnąć, czy następna wyświetlona karta będzie miała wartość
↪większą czy
mniejszą od aktualnej karty.')
print('Jeżeli zgadniesz, zdobywasz 20 punktów. W przeciwnym razie tracisz
↪15 punktów.')
print('Na początek masz 50 punktów.')
print()

startingDeckList = []
for suit in SUIT_TUPLE:  ❶.
    for thisValue, rank in enumerate(RANK_TUPLE):
        cardDict = {'rank':rank, 'suit':suit, 'value':thisValue + 1}
        startingDeckList.append(cardDict)

score = 50

while True:  # Możliwych jest kilka rund gry.
    print()
    gameDeckList = shuffle(startingDeckList)
```

```
    currentCardDict = getCard(gameDeckList)  ❷
    currentCardRank = currentCardDict['rank']
    currentCardValue = currentCardDict['value']
    currentCardSuit = currentCardDict['suit']
    print('Pierwsza widoczna karta to:', currentCardRank + ' of ' +
    ↪currentCardSuit)
    print()

    for cardNumber in range(0, NCARDS):  # Jedna gra z tych wielu kart. ❸
        answer = input('Jaka będzie następna karta: większa czy mniejsza niż' +
                       currentCardRank + currentCardSuit + '? (wpisz w lub m: ')
        answer = answer.casefold()  # Wymuszenie użycia małych liter.
        nextCardDict = getCard(gameDeckList)  ❹
        nextCardRank = nextCardDict['rank']
        nextCardSuit = nextCardDict['suit']
        nextCardValue = nextCardDict['value']
        print('Następna karta to:', nextCardRank nextCardSuit)

        if answer == 'w':  ❺
            if nextCardValue > currentCardValue:
                print('Masz rację, karta miała większą wartość.')
                score = score + 20
            else:
                print('Niestety karta nie miała większej wartości.')
                score = score - 15

        elif answer == 'm':
            if nextCardValue < currentCardValue:
                score = score + 20
                print('Masz rację, karta miała niższą wartość.')

            else:
                score = score - 15
                print('Niestety karta nie miała niższej wartości.')

        print('Twój wynik:', score)
        print()
        currentCardRank = nextCardRank
        currentCardValue = nextCardValue  # Nie potrzebujemy bieżącej karty.

    goAgain = input('Naciśnij ENTER, aby zagrać ponownie. "q" kończy grę: ')  ❻
    if goAgain == 'q':
        break

print('Żegnaj!')
```

Działanie programu rozpoczyna się od utworzenia listy przedstawiającej talię kart ❶. Każda karta to słownik przechowujący informacje o figurze, kolorze i wartości. W trakcie każdej rundy gry z talii zostaje pobrana pierwsza karta, a jej komponenty są zapisywane w zmiennych ❷. W wypadku siedmiu kolejnych kart gracz musi przewidzieć, jaką wartość będzie miała następna karta: wyższą czy

niższą od bieżącej ❸. Program pobiera następną kartę z talii, a jej komponenty zapisuje w drugim zbiorze zmiennych ❹. Gra porównuje odpowiedź udzieloną przez gracza z wartością nowej karty, a następnie na podstawie wyniku przyznaje lub odejmuje punkty graczowi ❺. Po zakończeniu rundy gracz może zdecydować o rozpoczęciu nowej rundy lub o zakończeniu gry ❻.

Ten program przedstawia wiele elementów dotyczących programowania, zwłaszcza w Pythonie: używanie zmiennych, poleceń przypisania, funkcji i ich wywołań, konstrukcji `if-else`, poleceń wyświetlających dane, pętli `while`, a także korzystanie z list, ciągów tekstowych i słowników. W książce przyjąłem założenie, że nie masz problemu ze zrozumieniem sposobu działania tego programu. Jeżeli którykolwiek fragment w omawianym programie sprawia Ci trudność lub wydaje się niejasny, to przed przejściem dalej prawdopodobnie dobrze byłoby zajrzeć do innego źródła dotyczącego podstaw programowania w Pythonie.

Kod wielokrotnego użycia

Skoro mamy do czynienia z grą karcianą, to jest oczywiste, że działanie kodu polega na tworzeniu i operowaniu symulowaną talią kart do gry. Jeżeli postanowisz utworzyć inną grę karcianą, byłoby doskonale móc ponownie użyć utworzonego wcześniej kodu odpowiedzialnego za obsługę talii i kart.

W programie proceduralnym często trudno jest zidentyfikować wszystkie fragmenty kodu powiązane z jedną częścią programu, np. w omawianym przykładzie są to talia i karty. W programie pokazanym na listingu 1.1. kod obsługi talii składa się z dwóch krotek stałych, dwóch funkcji, pewnego kodu głównego, który tworzy globalną listę przedstawiającą początkową talię 52 kart i kolejną globalną listę przedstawiającą talię kart używaną podczas gry. Zwróć uwagę, że nawet w tak małych programach jak tutaj omawiany dane i operujący nimi kod nie muszą być ze sobą blisko zgrupowane.

Dlatego też ponowne użycie w innym programie kodu przeznaczonego do obsługi talii i kart nie jest wcale takie łatwe. W rozdziale 12. powrócę do tego programu i pokażę, jak styl programowania zorientowanego obiektowo pozwala na znacznie łatwiejsze ponowne wykorzystanie kodu.

Symulacja konta bankowego

W drugim przykładzie proceduralnego stylu programowania pokażę kilka wariantów programu symulującego funkcjonalność banku. W każdej kolejnej wersji programu będzie się pojawiała nowa funkcjonalność. Warto w tym miejscu dodać, że te programy nie są przeznaczone do użycia w środowisku produkcyjnym. Niepoprawne dane wejściowe lub ich nieprawidłowe użycie może prowadzić do błędów. Tutaj celem jest skoncentrowanie się na tym, jak kod współdziała z danymi dotyczącymi jednego lub większej liczby kont bankowych.

Na początek należy ustalić, jakie operacje klient chce przeprowadzać względem konta bankowego i jakie dane będą niezbędne do przedstawienia takiego konta.

Zawsze wymagane operacje i dane

Lista operacji, jakie klient chce przeprowadzać względem konta bankowego, obejmuje takie pozycje:

- utworzenie (konta),
- wpłata środków,
- wypłata środków,
- sprawdzenie salda.

Zapoznaj się teraz z minimalną listą danych niezbędnych do przedstawienia konta bankowego:

- imię i nazwisko klienta,
- hasło,
- wysokość salda.

Zwróć uwagę, że wszystkie operacje są wyrażone za pomocą słów akcji (czasowniki), a wszystkie elementy danych to rzeczowniki. Zdefiniowane, rzeczywiste konto bankowe ma znacznie większą liczbę dostępnych operacji i będzie wymagało dodatkowych danych (takich jak adres posiadacza konta, numer telefonu, numer pesel itd.). Jednak w celu zachowania czytelności przykładu rozpoczynamy od czterech przedstawionych akcji i trzech elementów danych. Ponadto, by zachować prostotę i skoncentrować się na interesujących nas aspektach, wszystkie kwoty będą wyrażane w pełnych złotych. Trzeba również dodać, że w rzeczywistej aplikacji banku hasła nie są przechowywane w postaci zwykłego tekstu (nieszyfrowane), jak ma to miejsce w omawianych tutaj przykładach.

Implementacja 1. — pojedyncze konto bez funkcji

W początkowej wersji naszej implementacji (zobacz listing 1.2.) mamy obsługę pojedynczego konta.

Listing 1.2. Symulacja banku obsługującego tylko jedno konto

Plik: Bank1_OneAccount.py

```
# Wersja proceduralna.
# Bank — wersja 1.
# Tylko jedno konto.

ccountName = 'Janek'  ❶
accountBalance = 100
accountPassword = 'soup'
```

```
while True:
    print()  ❷
    print('Wybierz opcję b, aby wyświetlić saldo')
    print('Wybierz opcję d, aby dokonać wpłaty')
    print('Wybierz opcję w, aby dokonać wypłaty')
    print('Wybierz opcję s, aby wyświetlić informacje o koncie')
    print('Wybierz opcję q, aby zakończyć działanie programu')
    print()

    action = input('Co chcesz teraz zrobić? ')
    action = action.lower()  # Wymuszenie użycia małych liter.
    action = action[0]  # Użycie po prostu pierwszej litery.
    print()

    if action == 'b':
        print('Wyświetl saldo:')
        userPassword = input('Proszę podać hasło: ')
        if userPassword != accountPassword:
            print('Hasło jest nieprawidłowe.')
        else:
            print('Wysokość salda wynosi:', accountBalance)

    elif action == 'd':
        print('Wpłata środków:')
        userDepositAmount = input('Proszę podać kwotę wpłaty: ')
        userDepositAmount = int(userDepositAmount)
        userPassword = input('Proszę podać hasło: ')

        if userDepositAmount < 0:
            print('Kwota wpłaty musi być wartością dodatnią!')

        elif userPassword != accountPassword:
            print('Hasło jest nieprawidłowe.')

        else:  # OK.
            accountBalance = accountBalance + userDepositAmount
            print('Wysokość salda po operacji wynosi:', accountBalance)

    elif action == 's':  # Wyświetlenie informacji o koncie.
        print('Informacje:')
        print('       Imię', accountName)
        print('       Saldo:', accountBalance)
        print('       Hasło:', accountPassword)
        print()

    elif action == 'q':
        break

    elif action == 'w':
        print('Wypłata środków:')
        userWithdrawAmount = input('Proszę podać kwotę wypłaty: ')
        userWithdrawAmount = int(userWithdrawAmount)
```

```
        userPassword = input('Proszę podać hasło: ')

        if userWithdrawAmount < 0:
            print('Kwota wypłaty musi być wartością dodatnią.')

        elif userPassword != accountPassword:
            print('Hasło do tego konta jest nieprawidłowe.')

        elif userWithdrawAmount > accountBalance:
            print('Kwota wypłaty nie może być większa od wysokości salda.')

        else:  # OK.
            accountBalance = accountBalance - userWithdrawAmount
            print('Wysokość salda po operacji wynosi:', accountBalance)

print('Gotowe')
```

Działanie programu rozpoczyna się od inicjalizacji trzech zmiennych przeznaczonych do przedstawienia danych jednego konta bankowego ❶. Następnie zostaje wyświetlone menu dostępnych operacji ❷. Główny kod programu działa bezpośrednio na zmiennych globalnych zawierających dane konta.

W omawianym przykładzie wszystkie akcje zostały zdefiniowane na poziomie głównym. W kodzie nie ma żadnych funkcji. Ten program działa świetnie, choć może się wydawać nieco długi. W wypadku dłuższych programów typowo stosowane podejście polega na przeniesieniu powiązanych ze sobą fragmentów kodu do funkcji, a następnie wywołanie tych funkcji. To zostanie dokładniej omówione w następnej implementacji programu symulującego działanie banku.

Implementacja 2. — pojedyncze konto z funkcjami

W nowej wersji programu, przedstawionej na listingu 1.3., kod został podzielony na oddzielne funkcje, po jednej dla poszczególnych akcji. Ta symulacja również jest przeznaczona dla pojedynczego konta.

Listing 1.3. Utworzone z użyciem funkcji symulacja banku obsługującego tylko jedno konto

Plik: Bank2_OneAccountWithFunctions.py

```
# Wersja proceduralna.
# Bank — wersja 2.
# Tylko jedno konto.

accountName = ''
accountBalance = 0
accountPassword = ''

def newAccount(name, balance, password):  ❶
    global accountName, accountBalance, accountPassword
```

```
    accountName = name
    accountBalance = balance
    accountPassword = password

def show():
    global accountName, accountBalance, accountPassword
    print('       Imię', accountName)
    print('       Saldo:', accountBalance)
    print('       Hasło:', accountPassword)
    print()

def getBalance(password):  ❷
    global accountName, accountBalance, accountPassword
    if password != accountPassword:
        print('Hasło jest nieprawidłowe.')
        return None
    return accountBalance

def deposit(amountToDeposit, password):  ❸
    global accountName, accountBalance, accountPassword
    if amountToDeposit < 0:
        print('Kwota wpłaty musi być wartością dodatnią!')
        return None

    if password != accountPassword:
        print('Hasło jest nieprawidłowe.')
        return None

    accountBalance = accountBalance + amountToDeposit
    return accountBalance

def withdraw(amountToWithdraw, password):  ❹
    global accountName, accountBalance, accountPassword  ❺
    if amountToWithdraw < 0:
        print('Kwota wypłaty musi być wartością dodatnią.')
        return None

    if password != accountPassword:
        print('Hasło do tego konta jest nieprawidłowe.')
        return None

    if amountToWithdraw > accountBalance:
        print('Kwota wypłaty nie może być większa od wysokości salda.')
        return None

    accountBalance = accountBalance - amountToWithdraw  ❻
    return accountBalance

newAccount("Janek", 100, 'soup')  # Utworzenie konta.

while True:
    print()
```

```
    print('Wybierz opcję b, aby wyświetlić saldo')
    print('Wybierz opcję d, aby dokonać wpłaty')
    print('Wybierz opcję w, aby dokonać wypłaty')
    print('Wybierz opcję s, aby wyświetlić informacje o koncie')
    print('Wybierz opcję q, aby zakończyć działanie programu')
    print()

    action = input('Co chcesz teraz zrobić? ')
    action = action.lower()  # Wymuszenie użycia małych liter.
    action = action[0]  # Użycie po prostu pierwszej litery.
    print()

    if action == 'b':
        print('Wyświetl saldo:')
        userPassword = input('Proszę podać hasło: ')
        theBalance = getBalance(userPassword)
        if theBalance is not None:
            print('Wysokość salda wynosi:', theBalance)

    elif action == 'd':  ❼
        print('Wpłata środków:')
        userDepositAmount = input('Proszę podać kwotę wpłaty: ')
        userDepositAmount = int(userDepositAmount)
        userPassword = input('Proszę podać hasło: ')

        newBalance = deposit(userDepositAmount, userPassword)  ❽
        if newBalance is not None:
            print('Wysokość salda po operacji wynosi:', newBalance)

--- Wywołania do odpowiednich funkcji zostały usunięte. ---

print('Gotowe')
```

W tej wersji zdefiniowałem oddzielne funkcje dla poszczególnych operacji dostępnych dla konta bankowego — tworzenie ❶, sprawdzenie wysokości salda ❷, wpłata środków ❸ i wypłata środków ❹. Główny kod programu został również zmodyfikowany, aby zawierał wywołania różnych funkcji.

W efekcie program stał się znacznie czytelniejszy. Jeśli np. użytkownik wybierze opcję *d*, a tym samym wyrazi chęć wpłaty środków ❼, to kod wywoła funkcję `deposit()` ❸, co spowoduje przekazanie kwoty wpłaty i podanego przez klienta hasła do konta.

Jeżeli przeanalizujesz definicję dowolnej z tych funkcji — np. `withdraw()` — zauważysz użycie słowa kluczowego `global` ❺ w celu uzyskania dostępu (pobranie lub zdefiniowanie) zmiennych przedstawiających konto. W Pythonie polecenie `global` jest wymagane tylko wtedy, gdy w funkcji chcesz zmienić wartość zmiennej globalnej. Trzeba w tym miejscu podkreślić, że polecenia `global` użyłem tylko dlatego, aby jasno pokazać odwoływanie się funkcji do zmiennych globalnych, nawet jeśli są jedynie pobierane wartości.

Jedną z ogólnych zasad stosowanych w programowaniu jest to, że funkcje *nigdy* nie powinny modyfikować zmiennych globalnych. Funkcja powinna używać jedynie przekazanych jej danych, przeprowadzać obliczenia na podstawie tych danych i potencjalnie zwracać wynik lub wyniki. Wprawdzie funkcja `withdraw()` w tym programie działa, ale łamie tę zasadę przez modyfikację wartości zmiennej globalnej `accountBalance` ❻ i uzyskuje dostęp do wartości zmiennej globalnej `accountPassword`.

Implementacja 3. — dwa konta

W kolejnej wersji programu, przedstawionej na listingu 1.4., używane jest to samo podejście co wcześniej (zobacz listing 1.3.), ale została dodana możliwość obsługi dwóch kont.

Listing 1.4. Utworzona z użyciem funkcji symulacja banku obsługującego dwa konta

Plik: Bank3_TwoAccounts.py

```
# Wersja proceduralna.
# Bank — wersja 3.
# Dwa konta.

accountOName = ''
accountOBalance = 0
accountOPassword = ''
account1Name = ''
account1Balance = 0
account1Password = ''
nAccounts = 0

def newAccount(accountNumber, name, balance, password):
    global accountOName, accountOBalance, accountOPassword  ❶
    global account1Name, account1Balance, account1Password

    if accountNumber == 0:
        accountOName = name
        accountOBalance = balance
        accountOPassword = password
    if accountNumber == 1:
        account1Name = name
        account1Balance = balance
        account1Password = password

def show():
    global accountOName, accountOBalance, accountOPassword  ❷
    global account1Name, account1Balance, account1Password
    if accountOName != '':
        print('Konto 0')
        print('       Imię', accountOName)
        print('       Saldo:', accountOBalance)
```

```
        print('       Hasło:', account0Password)
        print()
    if account1Name != '':
        print('Konto 1')
        print('       Imię', account1Name)
        print('       Saldo:', account1Balance)
        print('       Hasło:', account1Password)
        print()

def getBalance(accountNumber, password):
    global account0Name, account0Balance, account0Password  ❸
    global account1Name, account1Balance, account1Password

    if accountNumber == 0:
        if password != account0Password:
            print('Hasło jest nieprawidłowe.')
            return None
        return account0Balance
    if accountNumber == 1:
        if password != account1Password:
            print('Hasło jest nieprawidłowe.')
            return None
        return account1Balance

--- Wywołania funkcji deposit() i withdraw() zostały usunięte. ---

--- Usunięcie kodu głównego odpowiedzialnego za wywołanie zdefiniowanych wcześniej funkcji. ---

print('Gotowe')
```

Nawet w wypadku zaledwie dwóch kont można dostrzec, że to podejście szybko wymknęło się spod kontroli. Przede wszystkim mamy po trzy zmienne globalne dla każdego konta — ❶, ❷, ❸. Ponadto każda funkcja ma konstrukcję `if` pozwalającej na wybór zestawu zmiennych globalnych, które będą odczytywane lub modyfikowane. Jeżeli zajdzie potrzeba dodania kolejnego konta, to oznacza konieczność zdefiniowania następnego zbioru zmiennych globalnych i kolejnych konstrukcji `if` we wszystkich funkcjach. Takie podejście jest po prostu niemożliwe do wykonania. Potrzebujemy innego sposobu obsługi dowolnej liczby kont.

Implementacja 4. — wiele kont z użyciem listy

Aby ułatwić obsługę wielu kont, program przedstawiony na listingu 1.5. wykorzystuje listy do przechowywania danych. W tej wersji programu zostały użyte trzy równorzędne listy: `accountNamesList`, `accountPasswordsList` i `accountBalances ↪List`.

Listing 1.5. Symulacja banku utworzona z użyciem list

Plik: Bank4_N_Accounts.py

```
# Wersja proceduralna.
# Bank — wersja 4.
# Dowolna liczba kont — implementacja wykorzystująca listy.

accountNamesList = []  ❶
accountBalancesList = []
accountPasswordsList = []

def newAccount(name, balance, password):
    global accountNamesList, accountBalancesList, accountPasswordsList
    accountNamesList.append(name)  ❷
    accountBalancesList.append(balance)
    accountPasswordsList.append(password)

def show(accountNumber):
    global accountNamesList, accountBalancesList, accountPasswordsList
    print('Konto', accountNumber)
    print('       Imię', accountNamesList[accountNumber])
    print('       Saldo:', accountBalancesList[accountNumber])
    print('       Hasło:', accountPasswordsList[accountNumber])
    print()

def getBalance(accountNumber, password):
    global accountNamesList, accountBalancesList, accountPasswordsList
    if password != accountPasswordsList[accountNumber]:
        print('Hasło jest nieprawidłowe.')
        return None
    return accountBalancesList[accountNumber]

--- Część funkcji została usunięta. ---

# Utworzenie dwóch przykładowych kont.
print("Numer konta bankowego Janka:", len(accountNamesList))  ❸
newAccount("Janek", 100, 'soup')

print("Numer konta bankowego Marysi:", len(accountNamesList))  ❹
newAccount("Marysia", 12345, 'nuts')

while True:
    print()
    print('Wybierz opcję b, aby wyświetlić saldo')
    print('Wybierz opcję d, aby dokonać wpłaty')
    print('Wybierz opcję n, aby utworzyć nowe konto')
    print('Wybierz opcję w, aby dokonać wypłaty')
    print('Wybierz opcję n, aby wyświetlić informacje o kontach')
    print('Wybierz opcję q, aby zakończyć działanie programu')
    print()
```

```
    action = input('Co chcesz teraz zrobić? ')
    action = action.lower()  # Wymuszenie użycia małych liter.
    action = action[0]  # Użycie po prostu pierwszej litery.
    print()

  if action == 'b':
       print('Wyświetl saldo:')
       userAccountNumber = input('Proszę podać numer konta: ')  ❺
       userAccountNumber = int(userAccountNumber)
       userPassword = input('Proszę podać hasło: ')
       theBalance = getBalance(userAccountNumber, userPassword)
       if theBalance is not None:
           print('Wysokość salda wynosi:', theBalance)

--- Kod pozostałej części interfejsu użytkownika został usunięty. ---

print('Gotowe')
```

Na początku programu następuje utworzenie trzech pustych list ❶. W celu utworzenia nowego konta odpowiednie wartości zostają przypisane poszczególnym listom ❷.

Skoro mamy do czynienia z wieloma kontami, korzystam z podstawowej koncepcji numeru konta bankowego. Gdy użytkownik tworzy konto, kod używa funkcji `len()` jednej z list i zwraca liczbę, która staje się numerem konta danego klienta ❸, ❹. Podczas tworzenia konta dla pierwszego klienta długość listy `accountNamesList` wynosi 0. Dlatego też pierwsze utworzone konto będzie miało numer 0. Następne konto będzie miało numer 1 itd. Następnie, podobnie jak w rzeczywistym banku, w celu wykonania dowolnej operacji po utworzeniu konta (np. wpłaty lub wypłaty środków) konieczne będzie podanie numeru konta ❺.

Jednak ten kod nadal korzysta z danych globalnych — teraz to są trzy globalne listy danych.

Wyobraź sobie umieszczenie tych danych w arkuszu kalkulacyjnym. Mogłyby mieć postać pokazaną w tabeli 1.1.

Tabela 1.1. Tabela przykładowych danych

Numer konta	Imię	Hasło	Saldo
0	Janek	**soup**	100
1	Marysia	**nuts**	3550
2	Bartek	**frisbee**	1000
3	Sara	**xxyyzz**	750
4	Henryk	**PW**	10 000

Dane są przechowywane w trzech globalnych listach Pythona, z których każda przedstawia kolumnę w tej tabeli. Po przyjrzeniu się np. wyróżnionej kolumnie można dostrzec, że wszystkie hasła zostały zgrupowane razem na jednej liście.

Nazwy użytkownika są grupowane na innej liście, podobnie jak wysokości salda przechowywane na trzeciej liście. Korzystając z tego podejścia, aby uzyskać dostęp do informacji o jednym z kont, trzeba je odczytać ze wszystkich trzech list i użyć do tego wartości indeksu.

Wprawdzie takie rozwiązanie działa, ale wydaje się wyjątkowo niewygodne. Dane nie są grupowane w żaden logiczny sposób; np. nie wydaje się właściwe, aby hasła wszystkich użytkowników były przechowywane razem. Ponadto, jeżeli chcesz dodać nowy atrybut do konta, np. adres lub numer telefonu, musisz utworzyć kolejną listę globalną i uzyskać do niej dostęp.

Zamiast tego chcemy grupowania informacji w sposób przypominający wiersz w tym samym arkuszu kalkulacyjnym, jak pokazałem w tabeli 1.2.

Tabela 1.2. Tabela naszych przykładowych danych

Numer konta	Imię	Hasło	Saldo
0	Janek	soup	100
1	Marysia	nuts	3550
2	Bartek	frisbee	1000
3	Sara	xxyyzz	750
4	Henryk	PW	10 000

W tym nowym podejściu każdy wiersz przedstawia dane powiązane z jednym numerem konta bankowego. Dane pozostały bez zmian, ich grupowanie natomiast jest znacznie naturalniejszym sposobem na przedstawienie konta.

Implementacja 5. — lista słowników kont

W celu zaimplementowania ostatniego podejścia potrzebujemy nieco bardziej złożonej struktury danych. W tej wersji następuje utworzenie listy kont, z których każde (poszczególne elementy list) jest słownikiem o takiej postaci:

```
{'name':<imię>, 'password':<hasło>, 'balance':<saldo>}
```

UWAGA *Za każdym razem, gdy przedstawiam wartość w nawiasie ostrym (<>), oznacza to, że należy ją zastąpić (łącznie z nawiasem) dowolnie wybraną wartością. W omawianym przykładzie <imię>, <hasło> i <saldo> to miejsca zarezerwowane, które należy zastąpić rzeczywistymi wartościami.*

Kod tej implementacji został zamieszczony na listingu 1.6.

Listing 1.6. Symulacja banku utworzona z użyciem listy słowników

Plik: Bank5_Dictionary.py

```
# Wersja proceduralna.
# Bank — wersja 5.
# Dowolna liczba kont — implementacja wykorzystująca listę słowników.

accountsList = []  ❶

def newAccount(aName, aBalance, aPassword):
    global accountsList
    newAccountDict = {'name':aName, 'balance':aBalance, 'password':aPassword}
    accountsList.append(newAccountDict)  ❷

def show(accountNumber):
    global accountsList
    print('Konto', accountNumber)
    thisAccountDict = accountsList[accountNumber]
    print('       Imię', thisAccountDict['name'])
    print('       Saldo:', thisAccountDict['balance'])
    print('       Hasło:', thisAccountDict['password'])
    print()

def getBalance(accountNumber, password):
    global accountsList
    thisAccountDict = accountsList[accountNumber]  ❸
    if password != thisAccountDict['password']:
        print('Hasło jest nieprawidłowe.')
        return None
    return thisAccountDict['balance']

--- Wywołania funkcji deposit() i withdraw() zostały usunięte. ---

# Utworzenie dwóch dodatkowych kont.
print("Numer konta bankowego Janka:", len(accountsList))
newAccount("Janek", 100, 'soup')

print("Numer konta bankowego Marysi:", len(accountsList))
newAccount("Marysia", 12345, 'nuts')

while True:
    print()
    print('Wybierz opcję b, aby wyświetlić saldo')
    print('Wybierz opcję d, aby dokonać wpłaty')
    print('Wybierz opcję n, aby utworzyć nowe konto')
    print('Wybierz opcję w, aby dokonać wypłaty')
    print('Wybierz opcję n, aby wyświetlić informacje o kontach')
    print('Wybierz opcję q, aby zakończyć działanie programu')
    print()

    action = input('Co chcesz teraz zrobić? ')
```

```
    action = action.lower()  # Wymuszenie użycia małych liter.
    action = action[0]  # Użycie po prostu pierwszej litery.
    print()

    if action == 'b':
        print('Wyświetl saldo:')
        userAccountNumber = input('Proszę podać numer konta: ')
        userAccountNumber = int(userAccountNumber)
        userPassword = input('Proszę podać hasło: ')
        theBalance = getBalance(userAccountNumber, userPassword)
        if theBalance is not None:
            print('Wysokość salda wynosi:', theBalance)

    elif action == 'd':
        print('Wpłata środków:')
        userAccountNumber= input('Proszę podać numer konta: ')
        userAccountNumber = int(userAccountNumber)
        userDepositAmount = input('Proszę podać kwotę wpłaty: ')
        userDepositAmount = int(userDepositAmount)
        userPassword = input('Proszę podać hasło: ')

        newBalance = deposit(userAccountNumber, userDepositAmount, userPassword)
        if newBalance is not None:
            print('Wysokość salda po operacji wynosi:', newBalance)

    elif action == 'n':
        print('Nowe konto:')
        userName = input('Jak masz na imię? ')
        userStartingAmount = input('Jakie jest saldo początkowe? ')
        userStartingAmount = int(userStartingAmount)
        userPassword = input('Podaj hasło do tego konta ')
        userAccountNumber = len(accountsList)
        newAccount(userName, userStartingAmount, userPassword)
        print('Numer nowego konta:', userAccountNumber)

--- Kod pozostałej części interfejsu użytkownika został usunięty. ---

print('Gotowe')
```

W tym podejściu wszystkie dane dotyczące konta klienta znajdują się w pojedynczym słowniku ❶. W celu wygenerowania nowego konta należy utworzyć słownik i dołączyć go do listy kont ❷. Każdemu kontu jest przypisana liczba (to zwykła liczba całkowita). To jest numer konta, który musi być podawany podczas przeprowadzania każdej operacji na tym koncie. Przykładowo klient podaje numer konta podczas dokonywania wpłaty środków, a funkcja `getBalance()` używa tego numeru jako indeksu na liście kont ❸.

Dzięki tym zmianom kod został nieco uporządkowany, a struktura danych stała się bardziej logiczna. Jednak każda funkcja w programie wciąż musi mieć dostęp do globalnej listy kont. Jak się dowiesz z następnego podrozdziału, umożliwienie funkcjom dostępu do danych wszystkich kont rodzi obawy dotyczące zapewnienia

bezpieczeństwa. W idealnej sytuacji poszczególne funkcje powinny mieć możliwość wpływania tylko na dane jednego konta.

Najczęstsze problemy z implementacją proceduralną

Przykłady zaprezentowane w rozdziale mają jeden wspólny problem: wszystkie dane, na których operują funkcje, są przechowywane w jednej lub więcej zmiennych globalnych. Z wymienionych tutaj powodów używanie wielu danych globalnych w połączeniu z programowaniem proceduralnym jest uznawane za błędną praktykę tworzenia kodu źródłowego.

1. Każda funkcja używająca i (lub) modyfikująca dane globalne nie może być w łatwy sposób wykorzystana w innym programie. Funkcja uzyskująca dostęp do danych globalnych działa na danych, które znajdują się na innym (wyższym) poziomie niż kod samej funkcji. W celu uzyskania dostępu do tych danych wspomniana funkcja wymaga polecenia `global`. Nie możesz tak po prostu wziąć funkcji operującej na danych globalnych i zastosować ją ponownie w innym programie. To jest możliwe jedynie w programie używającym podobnych danych globalnych.
2. Wiele programów proceduralnych ma ogromne kolekcje zmiennych. Z definicji zmienna globalna może być używana lub modyfikowana przez dowolny fragment kodu znajdujący się gdziekolwiek w programie. Operacje przypisania zmiennych globalnych są często porozrzucane w całym programie proceduralnym, zarówno w kodzie głównym, jak i w definicjach funkcji. Skoro wartość zmiennej może być zmieniona w dowolnym miejscu, niezwykle trudno jest debugować i konserwować programy utworzone w taki właśnie sposób.
3. Funkcje przeznaczone do używania danych globalnych zwykle mają dostęp do zbyt dużej ilości danych. Gdy funkcja używa globalnej listy, słownika lub innej globalnej struktury danych, wówczas ma dostęp do *wszystkich* znajdujących się w niej danych. Zwykle jednak funkcja powinna operować tylko na jednym fragmencie danych lub po prostu na ich niewielkiej części. Możliwość odczytywania i modyfikowania dowolnych danych w ogromnej strukturze może prowadzić do błędów, np. przypadkowego użycia lub nadpisania danych, które nie powinny być używane przez daną funkcję.

Rozwiązanie w stylu programowania zorientowanego obiektowo — pierwszy rzut oka na klasę

Na listingu 1.7. pokazałem podejście zorientowane obiektowo, łączące ze sobą cały kod i powiązane dane pojedynczego konta bankowego. W tym programie znalazło się wiele nowych koncepcji, a ich dokładnym omówieniem zajmę się począwszy od następnego rozdziału. Nie oczekuję, że w pełni zrozumiesz przedstawiony tutaj przykład. Zwróć jednak uwagę na połączenie kodu i danych w pojedynczym skrypcie (nazywanym *klasą*). Oto pierwsze zetknięcie z kodem programowania zorientowanego obiektowo.

Listing 1.7. Pierwszy przykład klasy w Pythonie

Plik: Account.py

```
# Klasa Account.

class Account():
    def __init__(self, name, balance, password):
        self.name = name
        self.balance = int(balance)
        self.password = password

    def deposit(self, amountToDeposit, password):
        if password != self.password:
            print('Podane hasło jest nieprawidłowe.')
            return None

        if amountToDeposit < 0:
            print('Kwota wpłaty musi być wartością dodatnią.')
            return None

        self.balance = self.balance + amountToDeposit
        return self.balance

    def withdraw(self, amountToWithdraw, password):
        if password != self.password:
            print('Hasło dla tego konta jest nieprawidłowe.')
            return None

        if amountToWithdraw < 0:
            print('Kwota wypłaty musi być wartością dodatnią.')
            return None

        if amountToWithdraw > self.balance:
            print('Kwota wypłaty nie może być większa od wysokości salda.')
            return None
```

```
        self.balance = self.balance - amountToWithdraw
        return self.balance

    def getBalance(self, password):
        if password != self.password:
            print('Podane hasło jest nieprawidłowe.')
            return None
        return self.balance

    # Kod dodany na potrzeby debugowania.
    def show(self):
        print('       Imię:', self.name)
        print('       Saldo:', self.balance)
        print('       Hasło:', self.password)
        print()
```

Spójrz na funkcje i zwróć uwagę, jak bardzo są podobne do wcześniejszych przykładów programowania proceduralnego. Te funkcje mają takie same nazwy jak we wcześniejszych fragmentach kodu — `show()`, `getBalance()`, `deposit()` i `withdraw()` — i zawierają jeszcze porozrzucane w całym kodzie słowo kluczowe `self` (lub `self.`), którego znaczenie poznasz w następnych rozdziałach.

Podsumowanie

Na początku rozdziału przedstawiłem proceduralną implementację kodu gry karcianej o nazwie *Większa czy mniejsza*. W rozdziale 12. pokażę, jak można utworzyć zorientowaną obiektowo wersję tej gry, wyposażoną w graficzny interfejs użytkownika.

Następnie poruszyłem problem symulacji banku zapewniającego obsługę najpierw jednego, a później wielu kont. Omówiłem różne sposoby użycia programowania proceduralnego do zaimplementowania symulacji banku, a także wymieniłem kilka problemów pojawiających się podczas stosowania takiego podejścia. Na końcu pokazałem, jak wygląda opisujący konto bankowe kod utworzony z użyciem klasy.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

PYTHON:
ZORIENTUJ SWÓJ
KOD NA OBIEKTY!

Python jest jednym z najczęściej używanych języków programowania. Może służyć do wielu celów, a jego możliwości są nieustannie poszerzane przez wciąż powstające biblioteki i narzędzia. Równocześnie niejeden programista poprzestaje na tworzeniu kodu w sposób proceduralny. Tymczasem w Pythonie jak najbardziej możliwe jest programowanie zorientowane obiektowo, pozwalające organizować kod w poręczne jednostki, których później można wielokrotnie używać. Taki kod jest czytelny, łatwy w rozbudowie i dużo efektywniejszy w działaniu.

Oto intuicyjny przewodnik dla średnio zaawansowanych programistów Pythona, pomyślany tak, by przyswajać zasady programowania zorientowanego obiektowo podczas praktycznych ćwiczeń. Dowiesz się, jakie problemy wiążą się z zastosowaniem podejścia proceduralnego i jak dzięki podejściu obiektowemu pisać kod łatwy w utrzymaniu i rozbudowie. Nauczysz się tworzyć klasy i obiekty w Pythonie i skorzystasz z tych umiejętności, by budować atrakcyjne elementy GUI. Niejako przy okazji poznasz framework pygame i płynnie przejdziesz do pisania interaktywnych gier i aplikacji zawierających widżety GUI, animacje i wiele różnych scen. Opanujesz ponadto takie koncepcje jak maszyna stanów, modalne okna dialogowe czy wzorce projektowe — a w praktyce zastosujesz wzorzec model-widok-kontroler.

W książce między innymi:

- gruntowne podstawy programowania zorientowanego obiektowo
- tworzenie obiektów i zarządzanie nimi
- praktyczne stosowanie hermetyzacji w kodzie
- zastosowanie polimorfizmu podczas tworzenia interfejsów
- mechanizm dziedziczenia w praktyce

Irv Kalb jest nauczycielem akademickim. Od ponad trzech dekad stosuje programowanie zorientowane obiektowo w różnych językach programowania. Od lat tworzy oprogramowanie służące celom edukacyjnym, jest również autorem książki *Learn to Program with Python 3: A Step-by-Step Guide to Programming*, wydanej przez Apress.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej! ▶

ISBN 978-83-283-9406-3

Cena: 89,00 zł

no starch press