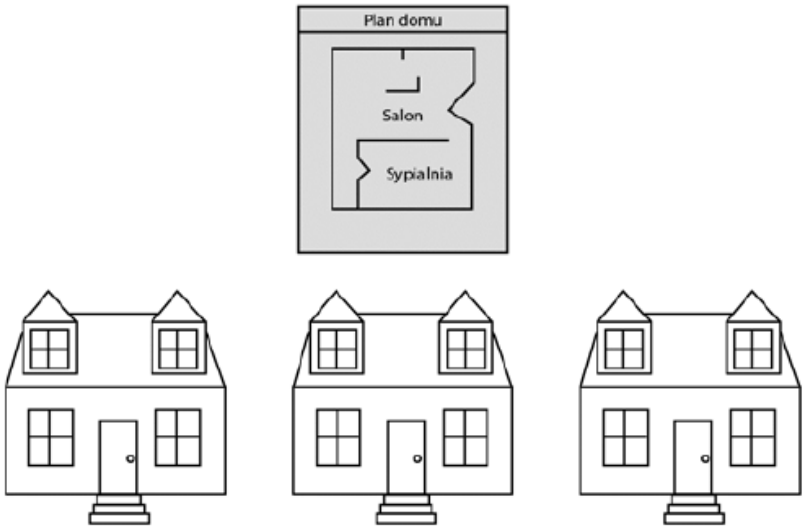
Wstęp do programowania obiektowego w języku Python

Programowanie proceduralne opiera się na procedurach (funkcjach), które realizują określone zadania, natomiast programowanie zorientowane obiektowo (ang. OOP – *object-oriented programming*) to paradygmat programowania, w którym programy zdefiniowane są za pomocą obiektów. **Obiekt** to encja, która zawiera zarówno dane, jak i procedury. Dane zawarte w obiekcie nazywamy **atrybutami danych**. Są to po prostu zmienne przechowywane w obiekcie. Procedury, które może wykonywać obiekt, nazywamy **metodami**. Metoda to nic innego jak funkcja przeprowadzająca operacje na atrybutach danych. Obiekt jest więc samodzielną jednostką zawierającą atrybuty danych i operujące na nich metody.

Programowanie zorientowane obiektowo za pomocą tzw. hermetyzacji i ukrywania danych rozwiązuje problem oddzielenia od siebie danych i kodu. **Hermetyzacja** polega na połączeniu danych i kodu w pojedynczy obiekt. Z kolei **ukrywanie danych** odwołuje się do możliwości obiektu w zakresie ukrycia atrybutów danych przed kodem znajdującym się na zewnątrz obiektu. Do odczytywania i modyfikacji atrybutów danych bezpośredni dostęp mają jedynie metody obiektu. Zanim utworzymy obiekt, należy go zaprojektować. Programista określa w tym celu wszystkie wymagane atrybuty danych i metody, a następnie tworzy **klasę**. Klasa to kod, za pomocą którego określamy atrybuty danych i metody danego typu obiektu.

Paradygmat OOP wziął się z otaczającej rzeczywistości. Przykładowo, gdy na podstawie planu budujemy dom, możemy powiedzieć, że budujemy **instancję** klasy (zwaną też **egzemplarzem**) budynku przedstawionego na planie. Jeżeli trzeba, na podstawie jednego planu można wybudować więcej domów. Każdy z wybudowanych domów jest oddzielnym egzemplarzem (instancją) domu przedstawionego na planie.



Klasa nie jest więc obiektem, lecz stanowi jego opis. Po uruchomieniu programu możemy za pomocą klasy tworzyć w pamięci komputera dowolną liczbę obiektów określonego typu. Każdy z tych obiektów nazywamy egzemplarzem (instancją) klasy. Obiekty odpowiadają więc rzeczownikom występującym w realnym świecie (np. auto, menu, klient, koszyk itp.), natomiast metody zajmują się działaniem na rzecz klasy lub instancji (np. obliczaniem, przetwarzaniem danych itp.). Weźmy pod uwagę obiekt w postaci zegara w smartfonie. Może on następujące atrybuty danych: bieżąca sekunda, minuta, godzina, czas, a także ustawienia budzika, alarmów itd. Atrybuty opisują więc stan, w jakim znajduje się zegar. Aby zmienić ustawienia zegara, trzeba wywołać odpowiednią metodę tego obiektu (ustawienie czasu, daty, budzika itp.). Są to tzw. metody publiczne, do których użytkownicy mają bezpośredni dostęp. Zegar ma także tzw. metody prywatne, które są przed nimi ukryte, przykładowo zwiększanie czasu co sekundę, minutę, godzinę itd.

Aby utworzyć klasę, należy przygotować definicję klasy, która jest zbiorem poleceń opisujących metody i atrybuty danych. Weźmy pod uwagę prosty program symulujący rzut monetą i ustalający, czy wypadła reszka (rewers) bądź orzeł (awers).

```
import random
class Coin:
    def __init__(self):
        self.__sideup = 'orzeł'
    def toss(self):
        if random.randint(0, 1) == 0:
            self.sideup = 'orzeł'
        else:
            self.sideup = 'reszka'
    def get_sideup(self):
        return self.sideup
```

Zdefiniowaliśmy tutaj klasę `Coin`, której pisownia z wielkiej litery nie jest obowiązkowa, ale jest dobrym zwyczajem programistycznym, bo łatwo zauważyć klasę podczas przeglądaniu kodu. Ma ona trzy metody:

- `__init__(self)`, która inicjalizuje atrybut danych `sideup` o wartości `orzeł`. Jest ona powszechnie nazywana metodą inicjalizacyjną, ponieważ w większości klas Pythona przeprowadza inicjalizację atrybutów danych obiektów i na ogół jest pierwszą metodą w definicji klasy, przy czym słowo `init` zawiera po dwa podkreślniki przed i po nim. Parametr `self` musi znajdować się w każdej metodzie klasy (wprawdzie nie musi mieć takiej nazwy, ale zaleca się, by zachować zgodność ze standardową praktyką stosowaną przez programistów). Po utworzeniu obiektu w pamięci operacyjnej następuje wywołanie metody `__init__()`, a parametr `self` ma automatycznie przypisany utworzony przed chwilą obiekt.
- `toss(self)`, która symuluje rzut monetą. W trakcie jej wywoływania instrukcja `if` wywołuje z kolei funkcję `random.randint()`, aby otrzymać liczbę losową `0` lub `1`. Liczba ta przypisywana jest atrybutowi danych o nazwie `self.sideup`.
- `get_sideup(self)`, która wywoływana jest za każdym razem, gdy chcemy poznać wynik rzutu monetą.

Pełny program, ilustrujący zastosowanie klasy `Coin`, może mieć następujący listing:

```
import random

class Coin:
    def __init__(self):
        self.sideup = 'orzeł'
    def toss(self):
        if random.randint(0, 1) == 0:
            self.sideup = 'orzeł'
        else:
            self.sideup = 'reszka'
    def get_sideup(self):
        return self.sideup

def main():
    my_coin = Coin()
    print('Wynik rzutu monetą:', my_coin.get_sideup())
    print('Symulacja rzutu monetą...')
```

```
    my_coin.toss()
    print('Wynik rzutu monetą:', my_coin.get_sideup())
main()
```

Polecenie `mycoin = Coin()` w programie powoduje wykonanie dwóch zadań, tj. utworzenie w pamięci obiektu o nazwie `Coin` oraz wywołanie metody `__init__` tej klasy, co powoduje automatyczne przypisanie parametrowi `self` utworzonego przed chwilą obiektu. W wyniku tej operacji atrybut danych `sideup` obiektu będzie miał przypisany ciąg tekstowy `orzeł`.

Powyższy przykład zawiera poważny błąd logiczny polegający na tym, że atrybut `sideup` w klasie `Coin` nie jest prywatny, a przez to nie jest chroniony przed przypadkową modyfikacją bądź uszkodzeniem. W języku Python ukrycie atrybutu polega na poprzedzeniu jego nazwy dwoma podkreślnikami. Dzięki temu jakikolwiek kod spoza klasy `Coin` nie będzie mógł uzyskać do niego dostępu. Poniższy listing zawiera nową definicję klasy `Coin` uwzględniającą przedstawione uwagi, a przy okazji losująca monetę dziesięciokrotnie.

```
import random

class Coin:
    def __init__(self):
        self.__sideup = 'orzeł'
    def toss(self):
        if random.randint(0, 1) == 0:
            self.__sideup = 'orzeł'
        else:
            self.__sideup = 'reszka'
    def get_sideup(self):
        return self.__sideup

def main():
    my_coin = Coin()
    print('Symulacja dziesięciu rzutów monetą:')
    for count in range(10):
        my_coin.toss()
        print(my_coin.get_sideup())
main()
```

Ze względu na lepszą organizację dużych projektów programistycznych zachodzi konieczność umieszczania definicji klas w modułach, które można zaimportować w dowolnym programie wymagającym danych klas.

Ćwiczenie 1.
Podziel kod przedstawionego programu na moduł `coin` (umieszczony w pliku `coin.py`), a następnie utwórz program `demo_coin.py`, w którym zaimportujesz ten moduł.

Utwórzmy teraz program, którego zadaniem będzie obsługa konta bankowego za pomocą klasy `BankAccount` umieszczonej w module `bankaccount`. Obiekty utworzone na podstawie tej klasy będą symulowały sytuację na koncie bankowym, pozwalającym użytkownikowi na zdefiniowanie salda początkowego, tworzenie depozytów, wypłatę środków i sprawdzenie bieżącego salda. Zawartość pliku `bankaccount.py`, w którym znajduje się moduł `bankaccount`, może być następująca:

```
class BankAccount:
    def __init__(self, bal):
        self.__balance = bal
    def deposit(self, amount):
        self.__balance += amount
    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print('Błąd: niewystarczająca ilość środków')
    def get_balance(self):
        return self.__balance
```

Metoda `def __init__(self, bal)` akceptuje argument w postaci salda konta i właśnie ta wartość zostanie przypisana argumentowi `__balance`. Metoda `deposit(self, amount)` symuluje utworzenie depozytu, metoda `withdraw(self, amount)` symuluje wypłatę środków z konta, natomiast metoda `get_balance(self)` zwraca bieżącą wysokość salda. Należy zwrócić uwagę, że wszystkie metody mają dwie zmienne parametrów. Parametr `bal` akceptuje początkową wysokość salda i w następnym wierszu kodu została mu przypisana wartość atrybutu `__balance` obiektu `BankAccount`. Po wywołaniu metody `deposit()` zdeponowana kwota zostanie przekazana metodzie za pomocą argumentu `amount` i w następnym wierszu kodu zostaje dodana do atrybutu `__balance`. Podobnie metoda `withdraw(self, amount)` ma dwa parametry, tj. `self` i `amount`. Po jej wywołaniu zostanie wypłacona kwota przekazana metodzie za pomocą argumentu `amount`. Metoda ta zawiera również zabezpieczenie przed wypłatą kwoty przekraczającej aktualną wysokość depozytu. Ostatnia metoda, `get_balance(self)`, ma za zadanie zwrócić wartość atrybutu `__balance`.

Przykładowy listing programu, korzystającego z modułu `bankaccount`, może wyglądać następująco:

```
import bankaccount
def main():
    # Określenie salda początkowego
    start_bal = float(input('Podaj początkową wysokość salda: '))

    # Utworzenie obiektu BankAccount
    savings = bankaccount.BankAccount(start_bal)

    # Zdeponowanie pewnej kwoty na koncie
    pay = float(input('Jaką kwotę zarobiłeś w tym miesiącu? '))
    print('Ta kwota została zdeponowana na koncie.')
    savings.deposit(pay)

    # Wyświetlenie bieżącego salda
    print('Wysokość salda wynosi ',
          format(savings.get_balance(), '.2f'), sep='')

    # Pobranie kwoty, która ma zostać wypłacona
    cash = float(input('Jaką kwotę chcesz wypłacić? '))
    print('Ta kwota zostanie odjęta od bieżącej wysokości salda.')
    savings.withdraw(cash)

    print('Wysokość salda wynosi ',
          format(savings.get_balance(), '.2f'), sep='')
main()
```

W programach bardzo często trzeba wyświetlić komunikat o stanie obiektu, czyli wartość atrybutów w danej chwili. W języku Python istnieje specjalna metoda `__str()__`, którą można wykorzystać, aby otrzymać ciąg tekstowy przedstawiający aktualny stan obiektu. Warto więc zmodyfikować odpowiednio kod modułu oraz kod programu wywołującego ten moduł.

Listing modułu `bankaccount.py`:

```
class BankAccount:
    def __init__(self, bal):
        self.__balance = bal
    def deposit(self, amount):
        self.__balance += amount
    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print('Błąd: niewystarczająca ilość środków')
    def get_balance(self):
        return self.__balance
    def __str__(self):
        return 'Wysokość salda wynosi ' + format(self.__balance,
'.2f')
```

Natomiast listing zmodyfikowanego programu wywołującego ten moduł może mieć postać:

```
import bankaccount
def main():
    start_bal = float(input('Podaj początkową wysokość salda: '))
    savings = bankaccount.BankAccount(start_bal)

    pay = float(input('Jaką kwotę zarobiłeś w tym miesiącu? '))
    print('Ta kwota została zdeponowana na koncie.')
    savings.deposit(pay)

    print(savings)

    cash = float(input('Jaką kwotę chcesz wypłacić? '))
    print('Ta kwota zostanie odjęta od bieżącej wysokości salda.')
    savings.withdraw(cash)

    print(savings)
main()
```

Ćwiczenie 2.

Jako pracownik działu IT firmy sprzedającej smartfony oraz inne urządzenia bezprzewodowe otrzymałeś zadanie opracowania klasy zakodowanej w module `cellphone` i przeznaczonej do zarządzania wszystkimi telefonami komórkowymi w firmie, przy czym zarząd narzucił działowi IT następujące zadania i warunki:

1. Dane, które muszą być przechowywane jako atrybuty danych klasy:
   - Nazwa producenta telefonu komórkowego ma być przypisana atrybutowi `__manufact`.
   - Numer modelu telefonu komórkowego ma być przypisana atrybutowi `__model`. Cena detaliczna telefonu komórkowego będzie przypisana atrybutowi `__retail_price`.

2. W klasie muszą być również zastosowane następujące metody:
   - `__init__()`. Metoda akceptuje argumenty przedstawiające producenta, numer modelu i cenę detaliczną.
   - `set_manufact()`. Metoda akceptuje argument przedstawiający producenta. Pozwala ona na zmianę wartości atrybutu `__manufact` po utworzeniu obiektu.
   - `set_model()`. Metoda akceptuje argument przedstawiający model. Pozwala ona na zmianę wartości atrybutu `__model` po utworzeniu obiektu.
   - `set_retail_price()`. Metoda akceptuje argument przedstawiający cenę detaliczną. Pozwala ona na zmianę wartości atrybutu `__retail_price` po utworzeniu obiektu.
   - `get_manufact()`. Metoda zwraca nazwę producenta telefonu komórkowego.
   - `get_model()`. Metoda zwraca numer modelu telefonu komórkowego.
   - `get_retail_price()`. Metoda zwraca cenę detaliczną telefonu komórkowego.
3. Po utworzeniu modułu z klasą musisz napisać program o nazwie `test_cellphone`, który ją wywołuje poprzez pobranie od użytkownika danych o kilku telefonach (o ich liczbie decyduje użytkownik) oraz wyświetleniu ich na ekranie.

Zadanie domowe (nieobowiązkowe)
Utwórz program przechowujący informacje o osobach, tj. imię i nazwisko, numer telefonu oraz adres e-mail. Program powinien zawierać menu i spełniać następujące funkcje: dodawać nową osobę, modyfikować dane istniejącej osoby, usuwać wybraną osobę, wyszukiwać dane wskazanej osoby oraz kończyć swoje działanie.