Szkolenie 10

Dziedziczenie. Przeładowanie operatorów.

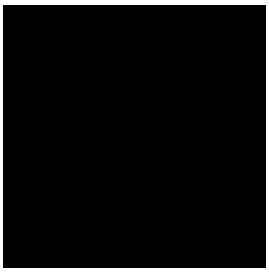
0. Słownik terminologii:

- **Klasa bazowa** – klasa, z której się dziedziczy.
- **Klasa pochodna** – klasa, która dziedziczy z klasy bazowej
- **Metody specjalne (np. __init__, __add__, __sub__, __str__)** - to wszystkie metody, które w nazwie posiadają prefix i sufix nazwy jako __. Specjalne metody klasy umożliwiają zachowywanie się jej instancjom w określony sposób, wykonując na nich różne operacje (np. dodawanie/odejmowanie dwóch obiektów itd.).
- **MRO – Methods Resolution Order –** to kolejność uruchomienia metoda ustalona na podstawie hierarchii klas.

Dziedziczenie jako potężne narzędzie w pisaniu programów.

Dziedziczenie to kolejna abstrakcja wprowadzona do paradygmatu programowania obiektowego umożliwiająca jeszcze lepsze odwzorowanie rzeczywistości. Dziedziczenie *(inheritance)* umożliwia bowiem przekazywanie pewnych cech **klasy bazowej (rodzica)** do **klas potomnych (dzieci)**. Dzięki takiemu podejściu, możemy tworzyć “rodziny klas” o wspólnych cechach zawierających, np. takie same atrybuty, metody.

Ponadto nie musimy wielokrotnie pisać tego samego kodu! Wyobraź sobie bowiem, że możesz utworzyć klasę bazową ***Zwierzę***, która będzie posiadała takie pola jak: **ilosc_nog, waga, gatunek** oraz metody: **daj_glos(), idz()**. Następnie utworzyłbyś klasy dziedziczące po klasie Zwierzę. Mogłyby to być takie klasy jak: **Gad, Ssak, Ptak itd**. W wyniku dziedziczenia, automatycznie każda z klas przejęłaby **(odziedziczyłaby)** wyżej wymienione **pola i metody**, przez co nie byłoby konieczne ich ponowne definiowanie.



Syntax klasy bazowej i dziedziczącej:

```
class KlasaBazowa:
    # ciało klasy bazowej

class KlasaPotomna(KlasaBazowa):
    # ciało klasy dziedziczącej
```

Zwróć uwagę na to, jak określamy, po czym ma dziedziczyć klasa. W momencie tworzenia klasy dziedziczącej, dodajemy do jej nazwy nawiasy i między nimi podajemy nazwę klasy bazowej - rodzica.

Pierwszy program wykorzystujący dziedziczenie.

Pierwszym, często przytaczanym przykładem na dziedziczenie jest program, którego klasy reprezentują wielokąty. Będziemy więc implementowali klasę bazową - **Wielokąt** oraz klasy pochodne: **Trójkąt, Prostokąt** dziedziczące po **Wielokącie**.

Przykład:

```python
class Wielokat:
    def __init__(self, boki_, suma_katow_):
        self.boki = boki_
        self.suma_katow = suma_katow_

    def oblicz_obwod(self):
        return sum(self.boki)

    def wyswietl_sume_katow(self):
        print(self.suma_katow)

class Trojkat(Wielokat):
    def __init__(self, a, b, c):
        super().__init__([a, b, c], 180)

    # Pole liczone z wzoru Herona
    def oblicz_pole(self):
        a, b, c = self.boki[0], self.boki[1], self.boki[2]
        p = (self.boki[0] + self.boki[1] + self.boki[2]) / 2
        return (p*(p-a)*(p-b)*(p-c)) ** 0.5

def main():
    trojkat_rownoramienny = Trojkat(5, 5, 5)
    print(trojkat_rownoramienny.oblicz_obwod())
    trojkat_rownoramienny.wyswietl_sume_katow()
    print("{:.4f}".format(trojkat_rownoramienny.oblicz_pole()))

if __name__ == "__main__":
    main()
```

W powyższym programie stworzyliśmy obiekt klasy Trojkat reprezentujący trójkąt równoramienny **(linia 23)**. Zwróć uwagę na to, że **konstruktor klasy bazowej** przyjmuje 2 argumenty – **boki_** oraz **suma_katow_.** Pierwszy argument jest odbierany jako **lista** boków, drugi jako wartość typu **int**.

W programie wyświetlamy obwód i sumę kątów trójkąta, a następnie jego pole. Co jest najważniejsze w powyższym przykładzie to fakt, iż w celu stworzenia klasy Trojkat, zastosowaliśmy **dziedziczenie po klasie Wielokat**. Klasa Trójkąt odziedziczyła metody **oblicz_obwod**, **wyswietl_sume_katow** oraz pola: **boki** oraz **suma_katow**. Co to znaczy, że odziedziczyła? To znaczy tyle, że uchroniliśmy się przed **powielaniem kodu**. Pola oraz funkcjonalność zaimplementowanych metod w Wielokat została **powielona** w klasie Trojkat.

Jak to wygląda od strony implementacyjnej?

Na szczególną uwagę zasługują linijki: **12** oraz **14**. Zacznijmy od linii, w której rozpoczynamy tworzenie klasy **(linia 12)**. Jak zapewne się domyśliłeś, w celu określenia po czym ma dziedziczyć klasa, podajemy nazwy konkretnych klas w nawiasach klasy dziedziczącej.

```
class nowa_klasa(klasa_rodzic):
```

Co ciekawe, możemy wymienić więcej niż jednego rodzica danej klasy **(wielodziedziczenie)**, ale o tym w następnej sekcji lekcji.

Linia 14 kodu: **super().__init__([a, b, c], 180)** *jest niczym innym jak konstruowaniem obiektu klasy rodzica. Musisz bowiem wiedzieć, że w momencie tworzenia klasy dziedziczącej,* ***MUSIMY NAJPIERW*** *z jej poziomu wywołać konstruktor klasy bazowej. To jest dość logiczne, ponieważ dane dziecko nie mogłoby się narodzić przed swoim rodzicem.*

A więc najpierw musimy w pełni utworzyć obiekt klasy, po której dziedziczymy, a następnie jego dziecko mogące już odziedziczyć konkretne cechy (pola i metody) po rodzicu.

Super *więc znaczy tyle, że masz się odwołać do rodzica i jego dowolnej metody lub pola (w przykładzie był to konstruktor* ***__init__()****).*

W starszych wersjach Pythona...

Przed erą Pythona 3, standardowym zapisem wywołującym konstruktor klasy bazowej z przykładu byłby: ***Wielokat.__init__().*** *Chciałbym więc, abyś nie był zaskoczony obecnością takiego wywołania w programach, z którymi możesz mieć do czynienia. Na tym etapie nie będę się rozwodził, w czym jest lepszy*

super().__init__() *od starszego wywołania konstruktora rodzica, ale póki co powinieneś być świadomy, że jest na pewno bardziej uniwersalny, schludniejszy i praktyczny.*

Dla tych, którzy nie są przekonani lub nie do końca rozumieją proces dziedziczenia, jeszcze jeden przykład implementujący klasy bazowe i pochodne.

Wyobraź sobie, że tworzysz grę RPG. Gra ma zawierać różne postaci (rasy) posiadające pewne wspólne cechy i zachowania.

Do głowy w tym momencie przychodzi mi gra Skyrim (polecam!). Mieliśmy tam do czynienia choćby z takimi postaciami jak: Elf, Blacksmith. Pozwól, abym na rzecz zgrabnego nazewnictwa zmiennych, program stworzył w języku angielskim.

Przykład:

```python
class Character:
    def __init__(self, name, start_items, abilities_):
        self.player_name = name
        self.hp = 100
        self.gold = 50
        self.inventory = start_items # list for ala-items in inv
        self.abilities = abilities_ # dictionary for abilities specific for a character

    def add_to_inventory(self, new_item):
        self.inventory.append(new_item)

    def use_ability(self, name):
        if name in self.abilities:
            print(self.abilities.get(name))
        else:
            print("Unknown ability!")

    def tell(self):
        print("Howdy, I'm", self.player_name)


class Blacksmith(Character):
    def __init__(self, name):
        super().__init__(name, ['hammer'], {'improving items' : 'an item has been improved...'})

    def tell(self):
        super().tell()
        print("I can repair your stuff...")


```

```python
class Elf(Character):
    def __init__(self, name):
        super().__init__(name, ['bow', 'knife'], {'shooting' : 'just shot a bow...', 'attack':
            'just attacked enemy with knife...'})

    def tell(self):
        super().tell()
        print("Fight or die...")


if __name__ == "__main__":
    elf = Elf("Legolas")
    blacksmith = Blacksmith("Jack")

    elf.tell()
    elf.add_to_inventory('item1')
    elf.use_ability('shooting')

    print() # space as an empty line

    blacksmith.tell()
    blacksmith.use_ability('attack')
```

Wynik:

Howdy, I'm Legolas

Fight or die...

just shot a bow...

Howdy, I'm Jack

I can repair your stuff...

Unknown ability!

Wyjaśnienie:

Oczywiście powyższy program wprowadza duże uproszczenie do całej mechaniki gry, ale myślę, że idea dziedziczenia jest tutaj dobrze odzwierciedlona.

Przyjrzyjmy się **klasie bazowej** – **Character**. Dziedziczą po niej klasy takie jak: **Blacksmith** oraz **Elf**. Klasa Character ma takie **pola jak**:

— **Nazwa gracza, hp (poziom życia), ilość złota, symulacja ekwipunku** oraz **słownik na umiejętności** (klucz słownika identyfikuje umiejętność, a wartość do niego przypisana odpowiada za komunikat, który ma się wyświetlić)

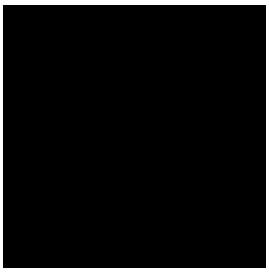
oraz metody:

— **add_to_inventory** (dodawanie nowej nazwy przedmiotu do ekwipunku), **use_ability** (umożliwiająca "użycie" danej umiejętności, a przy próbie skorzystania z nieistniejącej, metoda wyświetla komunikat o nieudanej operacji), **tell** (która służy do przedstawienia się przez gracza).

A teraz, analizując klasy pochodne, widzimy, że implementują one własną metodę **tell**, która rozszerza działanie tej z klasy bazowej o własny komunikat. W celu dostania się do ciała klasy bazowej, ponownie zastosowaliśmy słowo **super** *(linia 27 i 36)* - tak jak w konstruktorze klasy dziedziczącej, wywołując __init__ rodzica.

To teraz wyobraźmy sobie, że musimy napisać powyższy program, nie wykorzystując dziedziczenia. Zauważ, jak wiele kodu musielibyśmy powielić z klasy bazowej w klasach po niej dziedziczących. Klasy takie jak Blacksmith, Elf oraz wiele innych, które możemy stworzyć, aby rozwinąć naszą grę RPG, mają niektóre wspólne cechy i funkcjonalności. Dlatego wystarczy zaimplementować je jednorazowo (w klasie bazowej), a później tylko opierać się na potężnym narzędziu, jakim jest dziedziczenie, które tak mocno upraszcza tworzony kod. W ten sposób łatwo możemy rozszerzać zaimplementowane w rodzicu metody albo po prostu powielać je 1:1 w klasach dziedziczących.

Dziedziczenie wielokrotne



Super, wiemy już, że klasa może dziedziczyć po jednej klasie bazowej. Co ciekawe, możemy dziedziczyć po wielu innych klasach, czyli posiadać **więcej** niż jednego rodzica. Logika jest podobna jak przy dziedziczeniu pojedynczym. Dziecko dziedziczące po kilku klasach nie przejmuje już cech pojedynczego, ale kilku rodziców.

Schemat dziedziczenia wielokrotnego

```
class Parent1:
    pass

class Parent2:
    pass

class Derived(Parent1, Parent2):
    pass
```

Na razie przeanalizujmy sam schemat dziedziczenia, nie przechodźmy jeszcze do właściwej implementacji programu.

Jak widzisz, aby stworzyć klasę, która dziedziczy po wielu rodzicach, należy po prostu to sprecyzować między nawiasami dziecka – tak jak przy jednokrotnym dziedziczeniu, z tą różnicą, że teraz podajemy wielu rodziców.

Zasadnicza różnica jednak pojawia się przy wywoływaniu **konstruktorów klas bazowych**. Poniżej prezentujemy sposób na wywoływanie wielu konstruktorów wielu klas bazowych od wersji Python'a 3...

Przykład

```
1  class Parent1:
2      def __init__(self):
3          super().__init__()
4          print("In Parent1 class...")
5
6
7  class Parent2:
8      def __init__(self):
9          print("In Parent2 class...")
10
11
12 class Derived(Parent1, Parent2):
13     def __init__(self):
14         super().__init__()
15         print("In Derived class...")
16
17
18 def main():
19     d = Derived()
20
21
22 if __name__ == "__main__":
23     main()
```

Wynik

In Parent2 class...

In Parent1 class...

In Derived class...

Na pierwszy rzut oka kod wygląda na prosty i zrozumiały. W **linijce 14** mamy wywołanie funkcji super() i domyślamy się, że odpowiada ona za wywołanie konstruktorów klas bazowych. Jednak przyjrzyj się **linijce 3**... Tam też pojawił się super()! Super(), jak wiemy przecież, służy do odwoływania się do klas bazowych. A przecież jawnie nie sprecyzowaliśmy, iż Parent2 po czymś dziedziczy. Jak więc efekt przynosi super().__init__() w **3. linii** kodu?

Aby lepiej zrozumieć mechanizm, który jest tutaj uruchamiany, przeanalizujmy, co się dzieje w kodzie. W **19. linijce** tworzymy instancję (obiekt) klasy **Derived**. A więc uruchamiany jest jej konstruktor, po czym od razu napotykamy **linię 14. - super().__init__().**

Jak wiemy powoduje ona wywołanie konstruktora klasy bazowej – pytanie tylko, którego? Jako rodziców podaliśmy bowiem 2 klasy – **Parent1** oraz **Parent2** – **czy klasa Derived wybiera dowolny z nich**? Otóż nie. Reguła dotycząca tego, który konstruktor zostanie wywołany jako pierwszy **(Parent1 czy Parent2)**, jest związana z kolejnością nazw rodziców podanych w **linii 12.** My na liście klas bazowych jako pierwszy podaliśmy Parent1, tak więc to właśnie jego konstruktor zostanie uruchomiony z poziomu klasy dziedziczącej. Ekstra, ale co z konstruktorem **drugiej klasy** bazowej? W taki sposób jesteśmy przecież w stanie uruchomić jeden z dwóch istotnych dla nas konstruktorów klas rodzicielskich. Gdzie uruchomić konstruktor klasy Parent2 i zapewnić klasie Derived odziedziczenie po nim cech?

No właśnie realizowane jest to w wcześniej wspomnianej **3. linii kodu**. W tym kontekście **super().__init__()** umieszczony w tejże linijce nie znaczy tyle, co **wywołaj klasę bazową dla klasy Parent1**, ale wywołaj **drugą klasę bazową dla klasy Derived, czyli Parent2**. Przypadek dość specyficzny, temat ten bardziej poruszymy w przypadku, tzw. **Problemu diamentowego**, który omówimy w części zaawansowanej kursu. Na ten moment chciałbym, abyś był świadomy tego, iż pojawia się tutaj pojęcie **MRO (Method Resolution Order)**, które właśnie wpływa na kolejność wywołań konstruktorów.

Po dodaniu poniższej linii kodu w funkcji main():

```
20      print(Derived.mro())
```

Ujrzysz komunikat:

```
[<class '__main__.Derived'>, <class '__main__.Parent1'>,
<class '__main__.Parent2'>, <class 'object'>]
```

Metoda **mro()** wywołana na rzecz klasy Derived ukazuje właśnie w jakiej kolejności będą wywoływane konstruktory - począwszy od klasy dziedziczącej,

a skończywszy na klasach bazowych i klasie Object (w Pythonie każda klasa niejawnie dziedziczy po Object).

Tak więc, zgodnie z kolejnością klas ukazaną w wyniku wywołania mro() na rzecz klasy Derived, **linia 14.** kodu powoduje wywołanie konstruktora **Parent1,** w **linii 3.** natomiast następuje wywołanie konstruktora **Parent2** (patrz na MRO), dochodzimy więc do konstruktora **Parent2** i następuje wyświetlanie komunikatów ukazanych w wyniku powyżej napisanego programu **(In Parent2 class, In Parent1 class, In Derived classs).**

Przeładowanie operatorów

Temat dziedziczenia został już dość mocno przez Nas wyczerpany, a teraz skupimy się na, tzw. **Operators Overloading (przeładowanie operatorów)**. W prostym tłumaczeniu jest to nic innego jak nadawanie pewnych nowych zachowań znanym nam operatorom (pamiętasz z poprzednich zajęć operatory arytmetyczne, porównania itd.?) w **kontekście użycia z obiektami klas**.

Mam do Ciebie pytanie, nad którym zastanowienie się, pomoże lepiej zrozumieć sens przeładowywania operatorów.

Rozważ poniższe przykłady kodu, jakie będą one miały efekt?

Przykład 1 (konkatenacja dwóch list)

```
1  imiona_meskie = ["Kacper", "Jan", "Arkadiusz"]
2  imiona_zenskie = ["Kinga", "Magdalena", "Patrycja"]
3
4  print(imiona_meskie + imiona_zenskie)
```

Wynik

```
["Kacper", "Jan", "Arkadiusz", "Kinga", "Magdalena", "Patrycja"]
```

Przykład 2 (konkatenacja dwóch stringów):

```
1  imie1 = "Kacper"
2  imie2 = "Kinga"
3
4  print(imie1 + imie2)
```

Wynik:

```
KacperKinga
```

Przykład 3 (Sumowanie dwóch liczb typu int)

```
1  liczba1 = 11
2  liczba2 = 22
3
4  print(liczba1 + liczba2)
```

Wynik

```
33
```

Jak prezentują powyższe przykłady, **operator dodawania zachowuje się w różny sposób w każdym z przypadków**. W przypadku list i typu tekstowego, operator dodawania powoduje **konkatenacje (połączenie)** dwóch obiektów (listy i napisy też są obiektami), a w przypadku **liczb** standardową operację **sumowania**.

W zasadzie nic nowego, ale chciałbym, żebyś był świadomy, iż efekt taki a nie inny przy sumowaniu dla różnych typów, zależy od tego, jak jest przeładowany operator. Twórcy Pythona zadbali o to, aby operator dodawania wykorzystywany dla różnych struktur czy typów określonych w standardzie, dawał zawsze ten sam charakterystyczny efekt.

A co w momencie, gdybyśmy chcieli dodać do siebie obiekty dwóch klas stworzonych przez nas? Na przykład tak jak poniżej:

```python
class Obszar:
    def __init__(self, pole_):
        self.pole = pole_

def main():
    obszar1 = Obszar(100)
    obszar2 = Obszar(200)

    print(obszar1 + obszar2) # ????

if __name__ == "__main__":
    main()
```

Wynik błędu (wyjątek w linii 9):

TypeError: unsupported operand type(s) for +: 'Obszar' and 'Obszar'

Wyjątek pojawił się dlatego, ponieważ, jak możemy się domyślić, operacja dodawania na obiektach klasy **Obszar (obszar1 oraz obszar2)** jest niezdefiniowana. Program po prostu nie wie, w jaki sposób ma dodać do siebie te dwa obiekty.

I właśnie tutaj z pomocą przychodzi nam przeładowanie operatorów w klasach, a dokładnie przeładowanie operatora dodawania. Tak jak wspomnieliśmy na początku tej sekcji, przez przeładowanie operatora, chcemy zapewnić programowi, aby wiedział, co ma zrobić, gdy napotka dany rodzaj operatora.

Przeładowanie w praktyce

Przeładowanie operatorów będziemy realizowali za pomocą metod specjalnych (jedną z metod specjalnych już poznaliśmy i jest to __init__). W przypadku przeładowania dodawania - tą metodą specjalną jest __add__.

W kodzie prezentować się to będzie następująco:

```
1  class Obszar:
2      def __init__(self, pole_):
3          self.pole = pole_
4
5      def __add__(self, obj):
6          return self.pole + obj.pole
7
8  def main():
9      obszar1 = Obszar(100)
10     obszar2 = Obszar(200)
11
12     print(obszar1 + obszar2)
13
14
15 if __name__ == "__main__":
16     main()
```

Wynik:

300

Jak widzisz, przeładowanie w programie zrealizowaliśmy przez dodanie do definicji klasy Obszar **metodę __add__**. Tak jak zostało wspomniane, jest to **metoda specjalna**. Metody specjalne łatwo rozpoznać po zapisie **__ przed** i **po** danym wyrazie. Innymi metodami specjalnymi są: **__init__**, **__add__**, **__len__**, **__str__** i wiele, wiele innych. Wszystkie z nich poznasz z czasem, kiedy będziesz pisał coraz to więcej programów. Nie ma sensu przytaczać je wszystkie, ponieważ sposób ich implementacji jest podobny, a dokładniejsze informacje łatwo wyszukać w dokumentacja Pythona.

Jeszcze jedna sprawa. Metody te nazywane są specjalnymi również dlatego, ponieważ nie zawsze są wywoływane w sposób jawny **(czyli przez bezpośrednie podane nazwy**). Zauważ, bowiem, że metoda specjalna **__add__** zdefiniowana w **5. linijce kodu** programu powyżej została **niejawnie** wywołana po napotkaniu operatora **+ (linia 12).**

Przykłady innych metod przeładowujących

Tak jak zostało wspomniane, możemy przeładowywać naprawdę wiele operatorów (zarówno arytmetyczne, porównania etc). Niektórymi, równie praktycznymi są:

- **__sub__** - przeładowany operator odejmowania (realizuje zaimplementowane przez nas odejmowanie dla obiektów danej klasy)
- **__lt__ (lower than)** - przeładowany operator porównania, zwraca True lub False w zależności od tego, czy sprawdzone przez nas wartości są mniejsze niż dane wartości obiektu porównywanego
- **__eq__ (equal to)** - sprawdza, czy obiekty mają tę samą zawartość
- **__str__ (to str)** - powoduje zwrócenie wartości typu str, która ma reprezentować obiekt, na rzecz którego ta metoda została zwrócona

Zadanie

Wyobraź sobie program, w którym utworzoną listę jakichś obiektów (załóżmy - utworzonych użytkowników systemu), chcesz zapisywać do pliku. Z poziomu aplikacji realizowane by było tworzenie user'a (nadawanie mu nazwy, hasła oraz email'a), a po wybraniu odpowiedniej opcji w menu, następowałby zapis wszystkich dodanych użytkowników do pliku.

Kod

```python
class Uzytkownik:
    def __init__(self, nazwa_, email_, haslo_):
        self.nazwa = nazwa_
        self.email = email_
        self.haslo = haslo_

    def __str__(self):
        return self.nazwa + " " + self.email + " " + self.haslo


class Program:
    def __init__(self):
        self.czy_uruchomiony = True
        self.uzytkownicy = []


    def dodaj_uzytkownika(self):
        nazwa = input("Podaj nazwę użytkownika: ")
        email = input("Podaj email użytkownika: ")
        haslo = input("Podaj hasło użytkownika: ")

        nowy_uzytkownik = Uzytkownik(nazwa, email, haslo)
        self.uzytkownicy.append(nowy_uzytkownik)


    def zapisz(self):
        with open("uzytkownicy.txt", "a") as plik:
            for uzytkownik in self.uzytkownicy:
                plik.writeln(str(uzytkownik)) # wywołanie przeładowanej metody __str__


    def wyswietl_menu(self):
        while(self.czy_uruchomiony):
```

```
34              print("Wybierz opcje: ")
35              print("1 - Dodaj użytkownika")
36              print("2 - Zapisz do pliku")
37              print("3 - Zakończ program")
38
39              wybor = input()
40              self.wykonaj_z_menu(wybor)
41
42
43      def wykonaj_z_menu(self, wybor):
44          if wybor == "1":
45              self.dodaj_uzytkownika()
46          elif wybor == "2":
47              self.zapisz()
48          elif wybor == "3":
49              self.czy_uruchomiony = False
50          else:
51              print("Nieznana komenda!")
52
53
54  def main():
55      menu = Program()
56
57      menu.wyswietl_menu()
58
59
60  if __name__ == "__main__":
61      main()
```

Powyższy program został napisany zgodnie z **podejściem OOP (Object-Oriented Programming)**, ponadto wykorzystuje wcześniej poznane operacje na plikach **(zapis haseł do pliku)**. Rozmieszczenie klas i funkcjonalności w tym programie pozostawia jednak sporo do życzenia **(kod łamie tzw. Zasadę Open-Closed),** ale na rzecz przykładu musimy to zignorować. Problem się pojawia dopiero wtedy, gdybyśmy program tworzyli go na zamówienie jakiejś firmy, która chciałaby go z czasem rozwijać. Program bowiem w przypadku próby dodania nowej funkcjonalności, narzuca programiście modyfikację istniejącego kodu, co jest niepożądaną rzeczą w profesjonalnym programowaniu (łamiemy zasadę **Closed** z **Open-Closed Principle**). Na rzecz tego szkolenia i zrozumienia materiału, taka struktura programu jest jednak jak najbardziej odpowiednia.

Ale to na marginesie. Przechodząc konkretnie już do tego, co się dzieje w programie, możemy dostrzec **2 klasy**, które odpowiednio reprezentują całą funkcjonalność naszego programu (funkcjonalność **menu** i operacji **zapisu/dodawania** nowego użytkownika). Odwołując się jeszcze do pól klasy Program, widzimy, że przechowuje ona zmienną typu bool, która określa, czy użytkownik chce zamknąć aplikację. Korzystamy z niej w **linijce 33.** programu oraz w **linii 49**, w której zmieniamy wartość tej zmiennej na **False** (w ten sposób określamy, by program przy sprawdzaniu warunku pętli while w **linii 33**, przestał wykonać operacje w niej zawarte, co skutkuje zamknięciem programu).

Klasa **Uzytkownik** natomiast reprezentuje **dowolnego użytkownika**. Posiada ona atrybuty: **nazwa, email, haslo** oraz co najważniejsze w tym punkcie, **metodę specjalną __str__ (linia 7).** Zgodnie z tym, co wyjaśniliśmy w sekcji powyżej, powoduje ona zwrócenie wartości **typu str**, która tekstowo przedstawia obiekt, który chcemy zapisać do pliku **(linia 29)**.

Chciałbym również zwrócić Ci uwagę na to, że jak ma się zachowywać dana przeładowana metoda, zależy tylko od Ciebie. Tak naprawdę, implementując metodę **__str__,** mógłbyś zawsze zwracać napis “Ala ma kota”, ale wtedy jaki by to miało sens?

Metoda **__str__** jest wywoływana w **linii 29**, czyli tam, gdzie zapisujemy obiekt do pliku. Aby zapisać obiekt do pliku, musimy bowiem uzyskać jego reprezentację tekstową, którą dostarcza nam właśnie wcześniej wspomniana przeciążona metoda. Metoda __str__ nie zostaje również wywołana w sposób charakterystyczny **(uzytkownik.__str__)**, ale w wyniku konwersji obiektu klasy Uzytkownik na typ str (**str(uzytkownik)**).

Koniec części dla początkujących

Z tego miejsca bardzo chcielibyśmy Ci podziękować za dotrwanie do tej części kursu. Z czystym sumieniem, po opanowaniu wiedzy od Szkolenia nr 1 do teraz, możesz zacząć się nazywać coś pomiędzy początkującym a średnio zaawansowanym programistą Pythona. Pamiętaj, aby nie pomijać żadnych z dostarczanych przez mentorów zadań do każdego tematu. Tylko sumienna praca i dogłębne poznawanie tematów wraz szlifowaniem umiejętności praktycznych pomoże Ci wznieść się na wyżyny!

Następny cel, który powinieneś sobie postawić, to nauka z kolejnych materiałów szkoleniowych Devs-Mentoring, które poruszają tematy bardziej już zaawansowane oraz sposoby na wykorzystanie poznanej wiedzy w praktyce.

DEVS-MENTORING

– **Devs-Mentoring.pl 2021**