**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:**



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:**



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...**

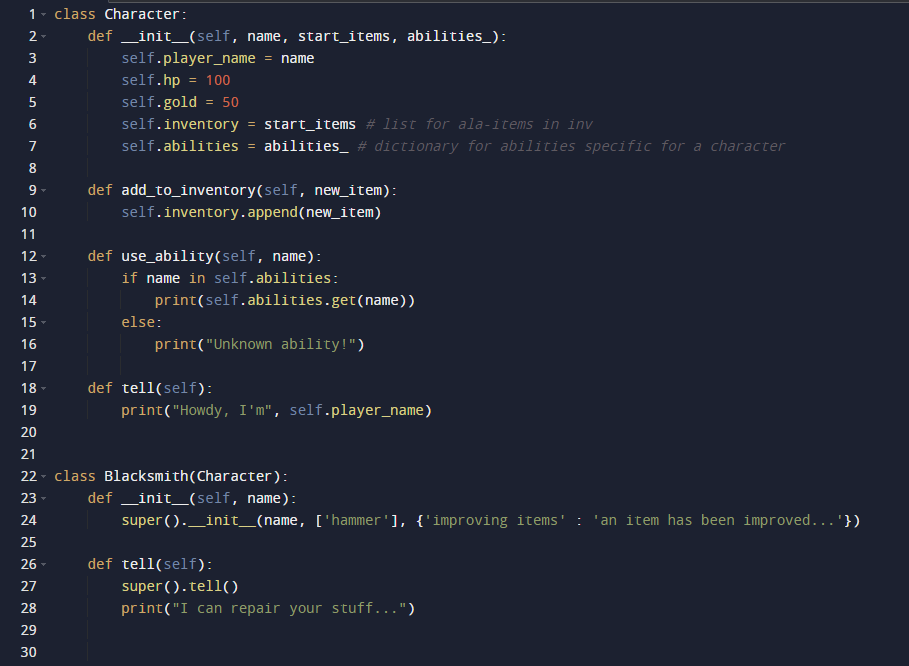
*Przed erą Pythona 3, standardowym zapisem wywołującym konstruktor klasy bazowej z przykładu byłby:* ***Wielokat.\_\_init\_\_().*** *Chciałbym więc, abyś nie był zaskoczony obecnością takiego wywołania w programach, z którymi możesz mieć do czynienia. Na tym etapie nie będę się rozwodził, w czym jest lepszy* **super().\_\_init\_\_()** *od starszego wywołania konstruktora rodzica, ale póki co powinieneś być świadomy, że jest na pewno bardziej uniwersalny, schludniejszy i praktyczny.*

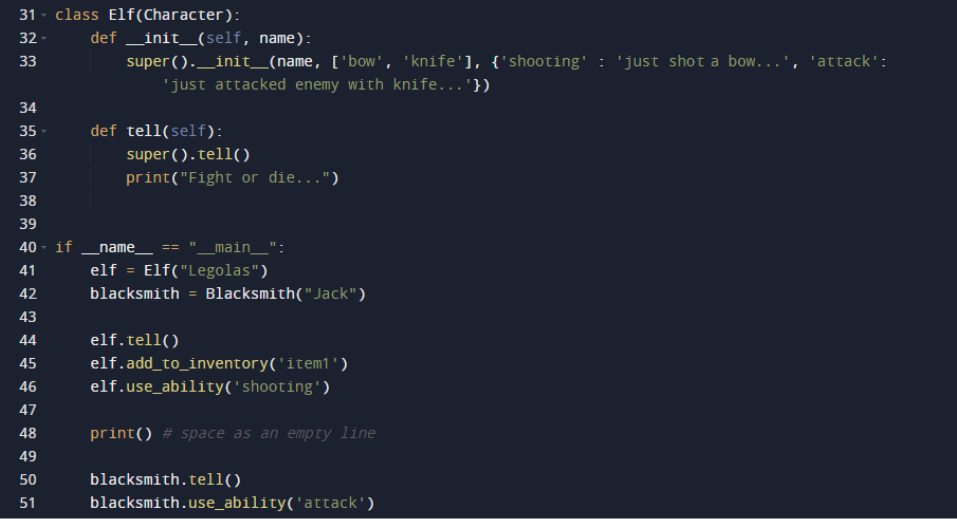
Dla tych, którzy nie są przekonani lub nie do końca rozumieją proces dziedziczenia, jeszcze jeden przykład implementujący klasy bazowe i pochodne.

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:**





**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).

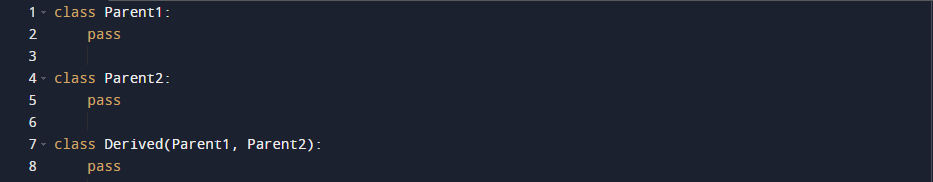
A teraz, analizując klasy pochodne, widzimy, że implementują one własną metodę **tell**, która rozszerza działanie tej z klasy bazowej o własny komunikat. W celu dostania się do ciała klasy bazowej, ponownie zastosowaliśmy słowo **super** *(linia 27 i 36)* - tak jak w konstruktorze klasy dziedziczącej,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

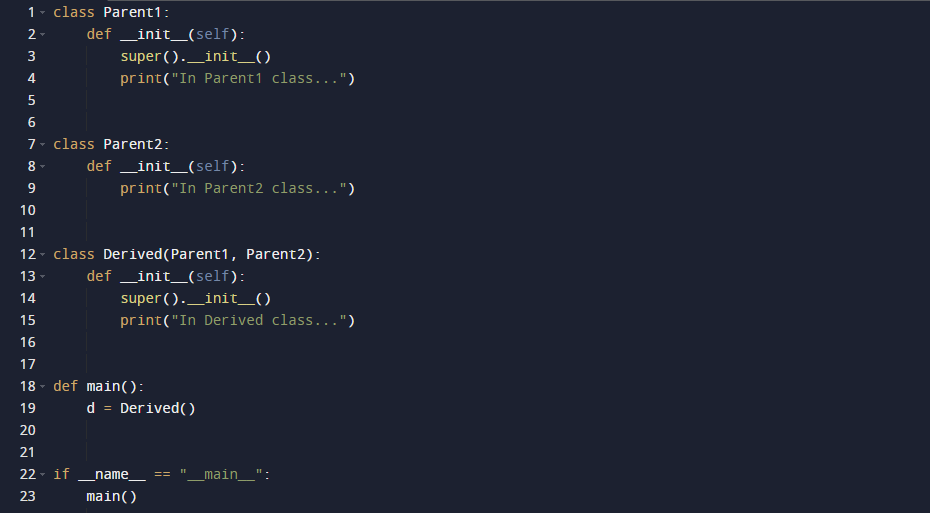


Na razie przeanalizujmy sam schemat dziedziczenia, nie przechodźmy jeszcze do właściwej implementacji programu.

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**



**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():



**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).**

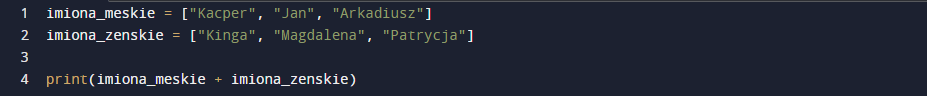
**Przeładowanie operatorów**

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?

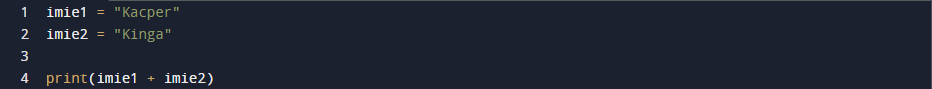
**Przykład 1 (konkatenacja dwóch list)**



**Wynik**

[“Kacper”, “Jan”, “Arkadiusz”, “Kinga”, “Magdalena”, “Patrycja”]

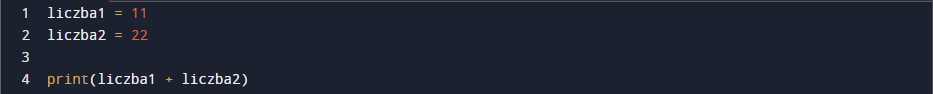
**Przykład 2 (konkatenacja dwóch stringów)**:



**Wynik:**

KacperKinga

**Przykład 3 (Sumowanie dwóch liczb typu int)**



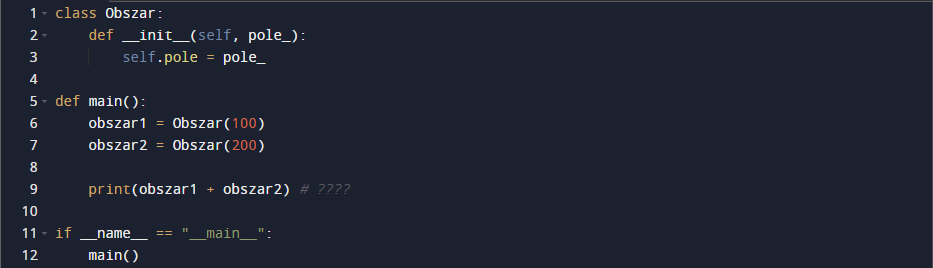
**Wynik**

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:



**Wynik błędu (wyjątek w linii 9):**

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

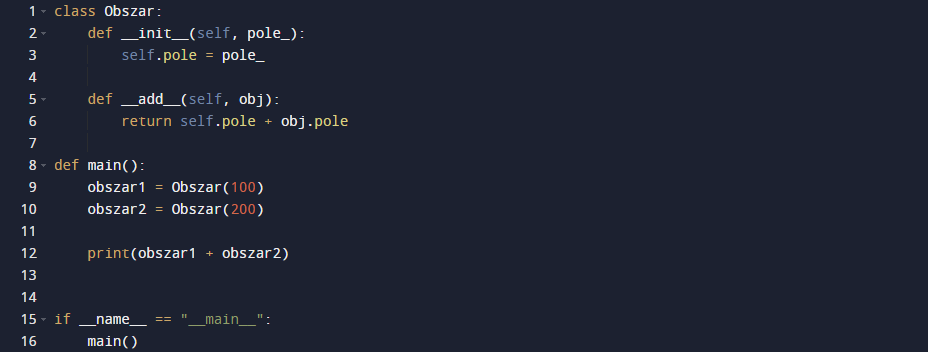
Wyjątek pojawił się dlatego, ponieważ, jak możemy się domyślić, operacja dodawania na obiektach klasy **Obszar (obszar1 oraz obszar2)** jest niezdefiniowana. Program po prostu nie wie, w jaki sposób ma dodać do siebie te dwa obiekty.

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:



**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**

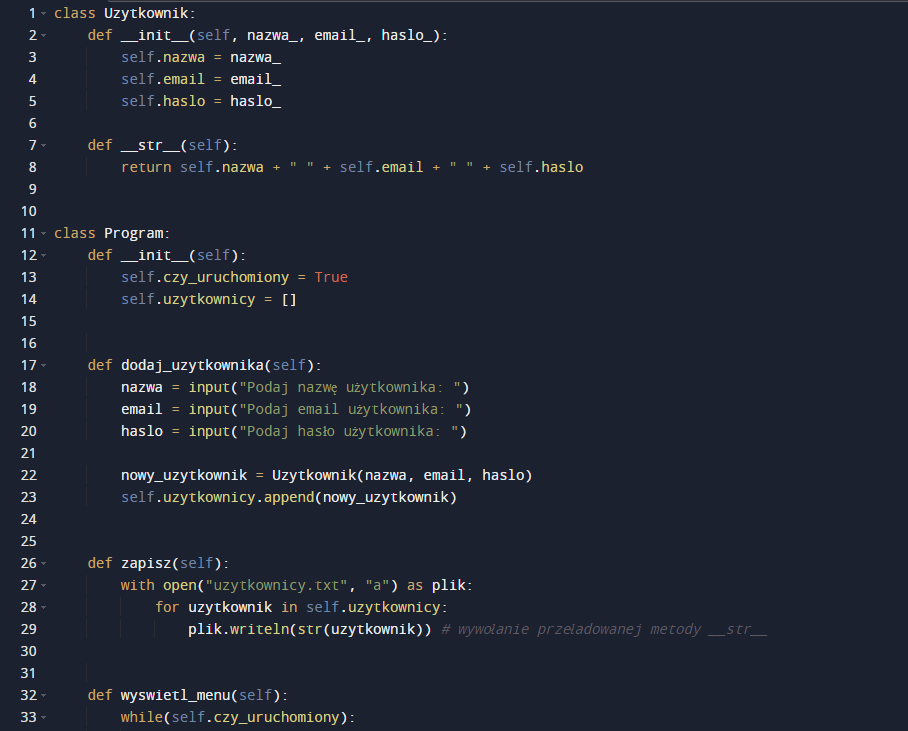
Tak jak zostało wspomniane, możemy przeładowywać naprawdę wiele operatorów (zarówno arytmetyczne, porównania etc). Niektórymi, równie praktycznymi są:

* **\_\_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**





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)**.

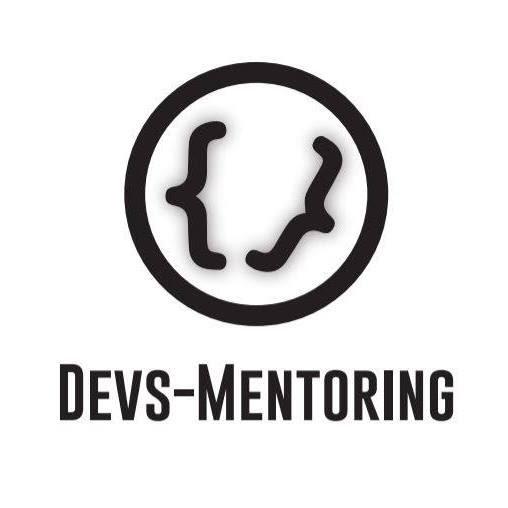
Chciałbym również zwrócić Ci uwagę na to, że jak ma się zachowywać dana przeładowana metoda, zależy tylko od Ciebie. Tak naprawdę, implementując metodę **\_\_str\_\_,** mógłbyś zawsze zwracać napis “Ala ma kota”, ale wtedy jaki by to miało sens?

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.pl 2021**