Szkolenie 9

Własne klasy. Nowe podejście do pisania programów.

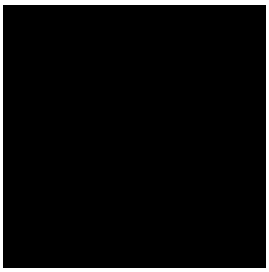
0. Słownik terminologii:

- **Programowanie obiektowe** – to nowe podejście do pisania programów, które ma ułatwić zrozumienie kodu. W kodzie bowiem korzystamy z klas i ich obiektów, co odzwierciedla rzeczywistość - mózg ludzki jest w naturalny sposób najlepiej przystosowany do takiego podejścia przy przetwarzaniu informacji.
- **Programowanie proceduralne (rodzaj programowania imperatywnego)** – to podejście, z którego korzystaliśmy do tej pory. Kod programu wydzielaliśmy bowiem na procedury (funkcje) realizujące określone działania.
- **Programowanie imperatywne** - to paradygmat programowania, w którym skupiamy się na tym **co chcemy zrobić** (wykonujemy konkretne czynności step-by-step). Tworzymy wynik.
- **Programowanie funkcyjne** (o którym więcej w zaawansowanej części kursu) - to obok programowanie imperatywnego, podejście, w którym skupiamy się na tym, **co chcemy osiągnąć**.

1. Podejście OOP (Object-Oriented Programming). Klasy i Obiekty.

Dobre wieści! Doszliśmy do etapu, który jest ostatnią totalnie nową rzeczą w nauce programowania. Dalsze kroki to szlifowanie poznanych umiejętności i poznawanie bardziej zaawansowanych technik, które rozszerzają to, co już wiesz.

Klasy i ich obiekty, to nieodłączne elementy obiektowego podejścia do pisania programów.



Klasa to reprezentacja jakiejś abstrakcji, jakiegoś tworu. Klasą może być **Człowiek, Zwierzę, Mebel, Pojazd**. Każdą z tych klas charakteryzują jakieś **pola** (lub inaczej **atrybuty**) oraz **metody** (to samo, co funkcje tylko, że z zmienioną nazwą). Przykładami pól klasy Człowiek mogą być: *kolor_oczu*, *kolor_skóry*, *wzrost*, *wiek* itd., a metodami: *śpij(), jedz(), przedstaw_się()* itd.

Jak widzisz, nie ma tu większej filozofii i faktycznie takie przedstawienie pewnych struktur w programie mocno działa na wyobraźnie i ułatwia zrozumienie kodu.

Obiekt (nazywany też instancją) to **"powołana do życia klasa",** a ujmując to bardziej technicznie - twór, który został powołany do życia gdzieś w pamięci RAM (pamięci ulotnej) komputera i możemy się do niego odwoływać i przeprowadzać na nim operacje (bądź co bądź obiekt jest również zmienną w programie). Jednak aby efektywnie pisać programy, musimy zrozumieć zasadniczą **różnicę** między obiektem a klasą.

Obiekt bowiem tak naprawdę jest **reprezentantem** klasy. Czyli, odwołując się jeszcze do naszego przykładu z klasą Człowiek, obiekt mógłby być tworem o nazwie **osoba1**, **osoba2**, **osoba3** itd. Z każdym z tych obiektów identyfikowalibyśmy różne **wartości pól** (jakie to pola? określilibyśmy to w momencie tworzenia klasy; jakie konkretne wartości? wartości zostałyby nadane w momencie tworzenia obiektów tejże klasy, czyli np. atrybut ***kolor_oczu*** ustawilibyśmy na wartość ***niebieski***, ***płeć*** na ***mężczyzna*** dla obiektu osoba1 itd.).

Jeżeli nie widzisz jeszcze zasadniczej różnicy między obiektem a klasą, przeanalizuj prosty przykład:

```python
class Czlowiek:
    def __init__(self, imie_, plec_, wiek_):
        self.imie = imie_ # tworzę i inicjalizuję pole imie
        self.plec = plec_ # tworzę i inicjalizuję pole plec
        self.wiek = wiek_ # tworzę i inicjalizuję pole wiek

def main():
    osoba1 = Czlowiek("Kacper", "M", 45) # tworzę pierwszy obiekt
    osoba2 = Czlowiek("Kinga", "K", 20) # tworzę drugi obiekt

    print("Imię pierwszej osoby to:", osoba1.imie) # wyświetlam imię pierwszego obiektu

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

Spokojnie, wiemy, że pojawiają się nieznane słowa takie jak **__init__, class** (choć znaczenie tego słowa można się domyślić) oraz **self**. W następnych akapitach dojdziemy do ich wyjaśnienia. Na ten moment, najważniejsze jest, abyś zauważył różnicę między klasą a obiektem. Komentarze zawarte w programie pomogą Ci ją w pełni zrozumieć.

Reasumując, klasa więc jest niejako uogólnionym schematem tworzenia obiektów. Wiemy, że każdy człowiek będzie miał takie właściwości (choć brzmi to dziwnie) jak ***imie, plec, wiek***. Jakie konkretnie będą wartości tych właściwości, określamy w momencie tworzenia obiektów danych klas (tak jak w trakcie narodzin – wtedy wówczas poznajemy, jaki kolor włosów ma dziecko, kolor skóry itd.).

2. Budowa klasy. Pola i metody.

Zacznijmy od budowy własnej klasy. Tak jak w przypadku tworzenia funkcji (gdzie przed nazwą określonej funkcji musiało się znaleźć słowo kluczowe ***def),*** konieczne jest przed umieszczenie słowa kluczowego **class** przed nazwą tworzonej klasy.

Po zapisaniu linijki składającej się z ***class*** oraz wybranej przez programistę **nazwy klasy** zakończonej **dwukropkiem**, budujemy **ciało** klasy. W ciele

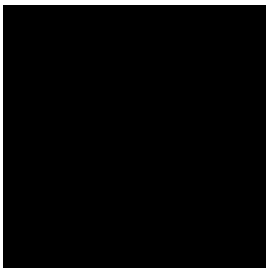
umieszcza się metody jak i pola (atrybuty klasy). Czym one się różnią i jakie jest ich przeznaczenie, wytłumaczyliśmy w sekcji powyżej.

Na szczególną uwagę zasługuje natomiast metoda specjalna zwana **konstruktorem -** __init__().

Konstruktor klasy __init__()

Metoda __init__() to pewnego rodzaju magiczna metoda, która jest zawsze automatycznie uruchamiana przy tworzeniu nowych obiektów klasy. Jak sama nazwa wskazuje – konstruuje ona obiekt danej klasy. Czyli umożliwia Nam sprecyzowanie, jakie wartość pól będzie on **przechowywał**. Metodę inicjalizującą dany obiekt tworzymy tak samo jak zwykłą funkcję, z tą różnicą, że posiada ona z góry określoną nazwę i jest to właśnie **__init__.** Ważne są jeszcze argumenty, jakie ona przyjmuje (to, co podajemy między nawiasami). Jakie to będą argumenty i jaka będzie ich ilość, zależy od ilości pól, jakie przechowuje klasa. To znaczy, jeżeli chcemy nadawać obiektowi danej klasy takie właściwości jak **kolor_orczu, kolor_skóry**, to musimy zadbać o to, aby te argumenty przyjmował nasz __init__(). Oczywiście nie jest to warunek konieczny, aby móc utworzyć obiekt danej klasy, ale jak inaczej efektywnie chcielibyśmy określać, jakimi cechami ma się oznaczać dana instancja klasy?

Poniżej przykłady konstruktorów różnych klas:



```python
class Pojazd:
    def __init__(self, ilosc_kol_, kolor_lakieru_):
        self.ilosc_kol = ilosc_kol_
        self.kolor_lakieru = kolor_lakieru_

class Mebel:
    def __init__(self, rozmiar_, cena_):
        self.rozmiar = rozmiar_
        self.cena = cena_

class Dom:
    def __init__(self, powierzchnia_, cena_, lokalizacja_):
        self.powierzchnia = powierzchnia_
        self.cena = cena_
        self.lokalizacja = lokalizacja_
```

Self?

Za co odpowiadają linie **self.nazwa_pola = ...** oraz self umieszczony na **liście argumentów konstruktora**?

Odpowiadając na pytania w odwrotnej kolejności - **self** jako **argument formalny** danej funkcji musi być umieszczany zawsze wtedy, gdy tworzymy metodę w obrębie danej klasy. Mówi on interpreterowi tyle, że metoda (na przykład __init__(), czyli konstruktor) **należy do klasy**, którą właśnie definiujemy.

Natomiast zatrzymując się przy self zawartym w ciele konstruktora...

Używamy go, aby:

- Utworzyć i skojarzyć określone pole z tworzoną klasą (***self.nazwa_pola*** = *wartość*)
- Nadać wartość danemu polu (self.nazwa_pola ***= wartość***)

Zapamiętaj!

Słowo kluczowe self, umieszczone na liście argumentów formalnych metody, służy do informowania interpretera, że tworzymy metodę, która ma być jedną z składowych danej klasy.

Po drugie, self umożliwia odwoływanie się do pól danej klasy, gdy znajdujemy się wewnątrz niej. Self będziemy wykorzystywali za każdym razem, gdy programując funkcjonalność danej klasy, będziemy odwoływali się do jej składników **(do pól i metod)**.

Linię: self.rozmiar = rozmiar_ z powyższego przykładu dotyczącego klasy Mebel, możesz więc rozumieć jako:

Odwołaj się do pola o nazwie rozmiar (jeżeli nie istnieje, to go utwórz) i nadaj mu wartość zgodną z argumentem przesłanym do konstruktora pod nazwą rozmiar_.

Przykład:

```
1  class Pojazd:
2      def __init__(self, ilosc_kol_, kolor_lakieru_):
3          self.ilosc_kol = ilosc_kol_
4          self.kolor_lakieru = kolor_lakieru_
5
6      def jedz(self):
7          print("Jadę na {} kołach.".format(self.ilosc_kol))
8
9  def main():
10     audi = Pojazd(4, "czerwony")
11     audi.jedz()
12
13 if __name__ == "__main__":
14     main()
```

Wynik*:*

Jadę na 4 kołach.

Metody

Super, pola mamy już z głowy. Z metodami w zasadzie większej filozofii nie ma Poznałeś już, jak tworzyć metodę specjalną, jaką jest __init__(). Pamiętasz o argumencie **self**, który musiałeś umieścić w momencie definiowania funkcji? Czyli **def __init__(self, ...).** W momencie definiowania dowolnych metod należących do danej klasy też musi się to słowo pojawić na jej liście argumentów.

Spójrz na przykłady:

```python
class Pojazd:
    def __init__(self, ilosc_kol_, kolor_lakieru_):
        self.ilosc_kol = ilosc_kol_
        self.kolor_lakieru = kolor_lakieru_

    def metoda_bezargumentowa(self):
        pass

    def metoda_z_jednym_argumentem(self, argument):
        pass

    # itd.
```

Stworzyliśmy 2 nowe metody - na razie bez ciała (słowo pass oznacza tyle: interpreterze, pozwól mi zostawić definicję funkcji, która nic nie będzie robiła, później zajmę się implementacją jej ciała) przyjmujące różną ilość argumentów. Powtarzamy - self pełni tu rolę taką, że umożliwia wywołanie metody na rzecz obiektu danej klasy - dzięki temu możemy z poziomu metody odwoływać się choćby do pól, którym nadaliśmy dane wartości w momencie tworzenia obiektów.

Zadanie:

Stwórzmy klasę reprezentującą Pojazd z dowolnymi atrybutami. Następnie utwórzmy instancje tej klasy, które będą reprezentowały pojazd osobowy i ciężarowy. W ramach klasy Pojazd utwórz metodę opisującą obiekt, na rzecz którego została metoda wywołana.

Rozwiązanie:

```
class Pojazd:
    def __init__(self, ilosc_kol_, marka_, kolor_lakieru_):
        self.ilosc_kol = ilosc_kol_
        self.marka = marka_
        self.kolor_lakieru = kolor_lakieru_

    def opisz_pojazd(self):
        print("Znajdujesz się w pojeździe, który: ")
        print("Ma {} kół, jest marki - {} i jest koloru - {}".format(self.ilosc_kol, self.marka, self
            .kolor_lakieru))

def main():
    samochod_osobowy = Pojazd(4, "Audi", "niebieski")
    samochod_ciezarowy = Pojazd(10, "Ashok Leyland", "żółty")

    samochod_osobowy.opisz_pojazd()
    samochod_ciezarowy.opisz_pojazd()

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

3. Pola prywatne i chronione

W Pythonie pojęcie pól prywatnych i chronionych jest niezbyt rozpowszechnione. Aczkolwiek, choćby z racji tego, iż w przyszłości możesz uczyć się innych języków programowania, warto poruszyć ten temat. Dotyczy to bowiem **enkapsulacji danych** w klasie. W Pythonie nie cieszy się to dużą popularnością i zastosowaniem, ale w innych językach wysokopoziomowych jak choćby Java czy C++, zgodnie z zasadą enkapsulacji danych, wszystkie pola powinny być prywatne (czyli niedostępne poza klasą). Tak więc korzystanie z nich, choćby w funkcji main() byłoby niemożliwe. Aby móc się do nich odwołać potrzebne by było stworzenie metod, tzw. **getterów** (pobierających wartość pól) i **setterów** (ustawiających wartości pól).

Dzięki enkapsulacji możemy **hermetyzować** dane, chroniąc je przed dostępem i ewentualną modyfikacją z zewnątrz.

Jak rozróżnić, czy pole jest prywatne, chronione lub publiczne? Jest na to proste rozwiązanie. Pola, których nazwa rozpoczyna się od __ są traktowane jako **prywatne,** natomiast te zaczynające się od _ jako **chronione.** Nie stosując żadnego z podkreślników przed nazwą pola, tworzymy go jako publiczne.

Przykład:

```
1  class Pojazd:
2      def __init__(self, ilosc_kol_, marka_, kolor_lakieru_):
3          self.__ilosc_kol = ilosc_kol_ # pole o dostępie prywatnym
4          self._marka = marka_ # pole o dostępie chronionym (protected)
5          self.kolor_lakieru = kolor_lakieru_ # pole o dostępi publicznym
6
7      ''' getter pola __ilosc_kol '''
8      def get_ilosc_kol(self):
9          return self.__ilosc_kol # OK
10
11     ''' setter pola __ilosc_kol '''
12     def set_ilosc_kol(self, ilosc_kol_):
13         self.__ilosc_kol = ilosc_kol_ # OK
14
15 def main():
16     samochod_osobowy = Pojazd(4, "Audi", "niebieski")
17     samochod_ciezarowy = Pojazd(10, "Ashok Leyland", "żółty")
18
19     print(samochod_osobowy.__ilosc_kol) # BŁĄD! Nie można się odwołać do pola prywatnego poza klasą.
20     print(samochod_osobowy.get_ilosc_kol()) # OK
21     print(samochod_osobowy._marka) # OK
22     print(samochod_osobowy.kolor_lakieru) # OK
23
24     samochod_ciezarowy.__ilosc_kol = 12 # BLAD!
25     samochod_ciezarowy.set_ilosc_kol(12) # OK
26
27
28 if __name__ == "__main__":
29     main()
```

Tak jak powiedzieliśmy - odwołanie do pól prywatnych poza klasą jest niemożliwe. Aby móc nimi manipulować konieczne jest utworzenie getter'a i setter'a dla danego pola. Gdybyśmy w klasie mieli kilka pól prywatnych, konieczne by było utworzenie metod ustawiających i zwracających każdy atrybut z osobna.

Na koniec zapytasz jeszcze pewnie, czym się różni pole prywatne (private) od chronionego (protected). Na pewno tym, że do prywatnego nie mamy dostępu poza klasą, ale czy czymś jeszcze? Tak. Prywatne atrybuty nie są jeszcze bowiem dziedziczone. O dziedziczeniu szerzej powiemy niebawem, także bądź, proszę, cierpliwy.

4. Referencja do obiektów. Usuwanie obiektów. Garbage Collector.

Czym jest referencja do obiektu? To informacja, gdzie obiekt jest przechowywany w pamięci RAM (dokładnie na stercie – heap'ie). Każdy z obiektów w momencie tworzenia i alokacji ma bowiem przypisany adres, do którego możemy tworzyć referencje (odnośniki).

Za pomocą tych referencji natomiast mamy możliwość modyfikowania obiektu (np. wywołanie setter'a dla danego atrybutu obiektu itd.). Powodem, dlaczego wspominamy o referencjach jest to, iż nieumiejętne operowanie na nich może prowadzić do nieoczekiwanych rezultatów.

Wyobraźmy sobie program, w którym utworzyliśmy instancję pewnej klasy z publicznymi polami. Następnie, przypisujemy "wartość" zmiennej, pod którą znajduje się obiekt, do innej utworzonej przez nas zmiennej.

Przykład:

```
1  class Pudelko:
2      def __init__(self):
3          self.ilosc_elementow = 4
4
5  def main():
6      p1 = Pudelko()
7      p2 = p1
8      p2.ilosc_elementow = 8
9
10     print(p1.ilosc_elementow) # Na ekranie wyświetli się 8
11
12 if __name__ == "__main__":
13     main()
```

Jak widzisz stworzyliśmy obiekt klasy pudełko o etykiecie p1. Następnie stworzyliśmy kolejną zmienną p2 i ustawiliśmy drugą referencję na wcześniej utworzony obiekt. Czyli otrzymujemy przez to niejako **2 referencje (wyobrażaj je sobie jako "sznurki" do adresu w pamięci)**, które są połączone z **jednym** obiektem. Wykonując jakieś operacje na obiekcie, korzystając z dowolnego "sznurka" **p1** lub **p2**, modyfikujemy obiekt, przez co zmiany są widoczne, odwołując się zarówno do pierwszej jak i drugiej referencji.

Jak widzisz w **linijce 10**., mimo że zmianę wartości pola obiektu wykonaliśmy, posługując się **zmienną (referencją) p2**, to zmiana również jest widoczna, gdy odwołujemy się do pola **przez zmienną p1**.

Zrozumienie i uświadomienie sobie, jakie zasady towarzyszą operowaniu na referencjach jest bardzo ważne w kontekście pisania rozbudowanych programów. Łatwo bowiem o pomyłkę i zamiast **kopiować** zawartość danego obiektu, możemy stworzyć **dwa odnośniki do jednego obiektu** i później modyfikować obiekt, który chcieliśmy, aby **został niezmieniony**.

To teraz zmieńmy Nasz przykład tak, aby zmienne p1 i p2 były od siebie **niezależne**.

Załóżmy, że chcemy stworzyć nowe pudełko, które będzie miało tę samą pojemność, ale będzie nowym obiektem. Jak to rozwiązać? Po prostu przez tworzenie kopii obiektu.

Przykład:

```python
class Pudelko:
    def __init__(self):
        self.ilosc_elementow = 4

def main():
    p1 = Pudelko()

    # Pierwszy sposób
    p2 = Pudelko()
    p2.ilosc_elementow = p1.ilosc_elementow
    p2.ilosc_elementow = 8
    print(p1.ilosc_elementow)

    # Drugi sposób (shallow copy)
    import copy
    p3 = copy.copy(p1)
    p3.ilosc_elementow = 8

    print(p1.ilosc_elementow)

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

Powyżej zaprezentowano **2 sposoby** kopiowania zawartości jednego obiektu do drugiego. Pierwszy jest niezbyt elegancki, ale został zaprezentowany dla lepszego zobrazowania rozwiązania. Jest to dość trywialne rozwiązanie i nie wymaga zbyt wielu wyjaśnień. Tworzymy drugi obiekt, a następnie polu **ilosc_elementow** nadajemy taką samą wartość, jaka była umieszczona w **pierwszym** obiekcie. Sposób dobry, gdy klasa zawiera **niewiele pól**, ale co w momencie, gdybyśmy zwiększyli ich ilość? Musielibyśmy przy każdym tworzeniu nowego obiektu dokonywać tych samych wielu operacji przypisania. Totalnie nieoptymalny sposób, dlatego został on zaprezentowany raczej do celów demonstracyjnych.

Kopie płytkie i głębokie

O wiele lepszym sposobem jest wykonywanie, tzw. **kopii płytkich i głębokich**, korzystając z modułu **copy** (pamiętaj jednak, aby go zaimportować przed użyciem).

Zacznijmy od **kopii płytkiej** (shallow copy), która jest realizowana w **linijce 16.** Odpowiada ona za "płytki" proces kopiowania zawartości danego obiektu do

innego. Co to znaczy płytki? Otóż chodzi o to, że ten rodzaj kopiowania sprawdzi się, gdy **żadne z pól** danej klasy **nie jest referencją** do obiektu jakiejś innej klasy.

Spójrz na przykład *(linia 7 jest właśnie polem klasy Pudelko, które jest równocześnie referencją do obiektu innej klasy - Przedmiot)*:

```
1  class Przedmiot:
2      def __init__(self, nazwa_):
3          self.nazwa = nazwa_
4
5  class Pudelko:
6      def __init__(self, nazwa_przedmiotu_):
7          self.przedmiot = Przedmiot(nazwa_przedmiotu_)
8          self.ilosc_elementow = 1
9
10 def main():
11     pudelko_na_sprzet = Pudelko("laptop")
12
13     import copy
14     pudelko_na_nowy_sprzet = copy.copy(pudelko_na_sprzet) # shallow copy
15
16     pudelko_na_nowy_sprzet.przedmiot.nazwa = "Telewizor"
17     pudelko_na_nowy_sprzet.ilosc_elementow = 2
18
19     print("Wartość z pierwszego pudełka powinna zostać niezmieniona (laptop), a przechowuje wartość:",
           pudelko_na_sprzet.przedmiot.nazwa) # NIEOCZEKIWANY WYNIK
20
21     print("Pierwsze pudełko powinno wciąż przechowywać pole ilosc_elementow = 1 i aktualnie jest ono
           równe:", pudelko_na_sprzet.ilosc_elementow) # OK
22
23 if __name__ == "__main__":
24     main()
```

Wynik*:*

>>> *Wartość z pierwszego pudełka powinna zostać niezmieniona (laptop), a przechowuje wartość: Telewizor*

>>> *Pierwsze pudełko powinno wciąż przechowywać pole ilosc_elementow = 1 i aktualnie jest ono równe: 1*

Jak widzisz, w wyniku kopiowania płytkiego, poprawnie zostały skopiowane tylko pola będące **typami prostymi** i de facto **niereferencjami**.

Problem pojawił się przy polu będącym obiektem pewnej innej klasy. W wyniku kopii płytkiej, takie pole nie jest **kopiowane**, ale jest **dowiązywana** do niego nowa referencja. Czyli ponownie dowiązujemy kolejny "sznurek", tym razem do pola innej klasy będącego obiektem *(przedmiot)*, w wyniku czego przy modyfikacji tego atrybutu, zmiana następuje również w polu obiektu, z którego chcieliśmy skopiować zawartość *(pudelko_na_sprzet)*.

Rozwiązanie:

Aby zapewnić kompleksowy proces kopiowania obiektów w momencie, gdy przechowują one w sobie obiekty innych klas, musimy skorzystać właśnie z kopiowania głębokiego.

Aby osiągnąć oczekiwany efekt kopiowania głębiokiego, należy zmodyfikować 14. linijkę kodu powyższego programu na:

```
14      pudelko_na_nowy_sprzet = copy.deepcopy(pudelko_na_sprzet) # deep copy
```

Zastąpiliśmy bowiem proces kopiowania płytkiego (copy.copy()) na kopiowanie głębokie (copy.deepcopy()).

Jeżeli czujesz, że temat jest dla Ciebie dość skomplikowany, nie przejmuj się. Udostępnione do tego modułu zadania, ugruntują Twoją wiedzę i pomogą lepiej zrozumieć temat!