Programowanie obiektowe - dziedziczenie. Iteratory i generatory.

1. Dziedziczenie

Mając klasę bazową możemy utworzyć klasę pochodną, która będzie dziedziczyć po klasie bazowej czyli będzie miała dostęp do atrybutów i metod z klasy bazowej. W klasie pochodnej można dodać nowe metody lub atrybuty.

Przykład

```
class Kształty():
    #definicja konstruktora
    def __init__(self, x, y):
        #deklarujemy atrybuty
        #self wskazuje że chodzi o zmienne własnie
definiowanej klasy
        self.x = x
        self.y = y
        self.opis = "To jest klasa dla ogólnych kształtów"

    def pole(self):
        return self.x * self.y

    def obwod(self):
        return 2 * self.x + 2 * self.y

    def dodaj_opis(self, text):
        self.opis = text

    def skalowanie(self, czynnik):
        self.x = self.x * czynnik
        self.y = self.y * czynnik

#klasa która dziedziczy po klasie Kształty
class Kwadrat(Kształty):
    def __init__(self,x):
        self.x = x
        self.y = x

# i jeszcze klasa, która dziedziczy po klasie Kwadrat
# bedzie definiwoać figurę złożoną z 3 kwadratów w kształcie
litery L
#  _
# | |_
# |_ _|
class KwadratLiteraL(Kwadrat):
    def obwod(self):
        return 8 * self.x

    def pole(self):
        return 3 * self.x * self.y
```

```
#inicjujemy klasę Kwadrat
kwadrat = Kwadrat(5)

#sprawdzenie metod z klasy bazowej
print(kwadrat.obwod())
print(kwadrat.pole())
kwadrat.dodaj_opis("Nasza figura to kwadrat")
print(kwadrat.opis)
kwadrat.skalowanie(0.3)
print(kwadrat.obwod())
print("")

#inicjujemy klasę KwadratLiteraL
litera_l = KwadratLiteraL(5)
print(litera_l.obwod())
print(litera_l.pole())
litera_l.dodaj_opis("Litera L")
print(litera_l.opis)
litera_l.skalowanie(0.5)
print(litera_l.obwod())
```

2. Przesłanianie metod.

Przykład przesłaniania metody został przedstawiony w przykładzie 1, ale warto dodać, że możemy również przesłaniać metody i zmienne dziedziczone po superklasie bazowej object, czyli tej, po której dziedziczy każdy obiekt w Pythonie. Możemy np. przeciążyć metodę __str__(), która zwraca tekstową reprezentację obiektu i domyślnie wyświetla informację o typie obiektu oraz adresie zajmowanym w pamięci komputera.

Przykład

```
class Kwadrat(Kształty):
    def __init__(self,x):
        self.x = x
        self.y = x

kwadrat = Kwadrat(5)
print(kwadrat)
```

```
class Kwadrat(Kształty):
    def __init__(self,x):
        self.x = x
        self.y = x

    def __str__(self):
        return 'Kwadrat o boku {}'.format(self.x)

kwadrat = Kwadrat(5)
print(kwadrat)
```

W pierwszym przypadku zostanie wywołana metoda __str__() klasy object, bo w żadnej wcześniejszej klasie (Kwadrat, Ksztalty) taka metoda nie została znaleziona (funkcja print() wypisuje string więc najpierw mui nastąpić konwersja dowolnego typu na string).

3. Atrybuty globalne, 'pusta' klasa oraz ponownie zmienna 'prywatna'

Nawiązując do wprowadzenia do programowania obiektowego w języku Python należy wspomnieć o możliwości stworzenia atrybutów współdzielonych przez wszystkie instancje danej klasy.

Przykład

```
class Point:
    counter = []

    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def update(self, n):
        self.counter.append(n)

p1 = Point(0,0)
p2 = Point(1,1)

print(p1.counter)
print(p2.counter)
p1.update(1)
print(p1.counter)
print(p2.counter)
```

Na wyjściu otrzymamy:

[]

[]

[1]

[1]

Ciekawostką jest to, że możemy stworzyć „pustą” klasę tylko po to, żeby przechować wartość wielu zmiennych w pojedynczej referencji, coś jak struktura w języku C.

```
class Pracownik():
    pass

lista = []
janek = Pracownik()
janek.imie = "Janek"
janek.nazwisko = "Kowalski"
janek.wiek = 30
```

Wracając do zmiennych prywatnych z zestawu 4 warto jeszcze wiedzieć o sposobie „dostania” się do tej zmiennej.

```
class Pracownik():
    __prywatna = "tajne haslo"

    def __init__(self, imie):
        self.imie = imie

janek = Pracownik("Janek")
print(janek.__prywatna) #to nie zadziała
print(janek._Pracownik__prywatna) #ale to już tak
```

4. Konstruktor klasy bazowej i dziedziczenie wielokrotne.

Poniższy przypadek pokazuje ponownie dziedziczenie jednokrotne po klasie bazowej, gdzie mamy 3 klasy:

```
class Osoba:

    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko

    def przedstaw_sie(self):
        return "{} {}".format(self.imie, self.nazwisko)


class Pracownik(Osoba):

    def __init__(self, imie, nazwisko, pensja):
        Osoba.__init__(self, imie, nazwisko)
        # lub
        # super().__init__(imie, nazwisko)
        self.pensja = pensja

    def przedstaw_sie(self):
        return "{} {} i zarabiam {}".format(self.imie,
self.nazwisko, self.pensja)


class Menadzer(Pracownik):

    def przedstaw_sie(self):
        return "{} {}, jestem menadżerem i zarabiam
{}".format(self.imie, self.nazwisko, self.pensja)


jozek = Pracownik("Józek", "Bajka", 2000)
adrian = Menadzer("Adrian", "Mikulski", 12000)
```

```
print(jozek.przedstaw_sie())
print(adrian.przedstaw_sie())
```

Zwróć uwagę na konstruktor klasy Pracownik, który wywołuje konstruktor bazowej klasy Osoba. Natomiast w definicji klasy Manadzer konstruktora nie ma a mimo to jestem w stanie zainicjalizować obiekt tak jak obiekt Pracownik.

Zwróć uwagę na poniższy przykład dziedziczenia wielokrotnego i konstruktor.

```
class Osoba:

    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko

    def przedstaw_sie(self):
        return "{} {}".format(self.imie, self.nazwisko)


class Pracownik:

    def __init__(self, pensja):
        self.pensja = pensja

    def przedstaw_sie(self):
        return "{} {} i zarabiam {}".format(self.imie,
self.nazwisko, self.pensja)


class Menadzer(Osoba, Pracownik):

    def __init__(self, imie, nazwisko, pensja):
        Osoba.__init__(self, imie, nazwisko)
        Pracownik.__init__(self, pensja)

    def przedstaw_sie(self):
        return "{} {}, jestem menadżerem i zarabiam
{}".format(self.imie, self.nazwisko, self.pensja)


adrian = Menadzer("Adrian", "Mikulski", 12000)
print(adrian.przedstaw_sie())
```

5. Iteratory i generatory.

Rozpatrując poniższy fragment kodu:

```
for element in range(1,11):
    print(element)
```

Wszystko raczej jest jasne. Ale skąd pętla for wie jak ma się uniwersalnie zachowywać dla różnych obiektów iterowalnych ? Cały mechanizm jest obsługiwany przez iteratory. W niewidoczny dla nas sposób pętle for wywołuje funkcję iter() na obiekcie kolekcji. Funkcja zwraca obiekt iteratora, który ma zdefiniowaną metodę __next__(), odpowiedzialną za zwracanie kolejnych elementów kolekcji. Kiedy nie ma już więcej elementów kolekcji zgłaszany jest wyjątek StopIteration, kończący działanie pętli for. Można wywołać funkcję __next__() iteratora za pomocą wbudowanej funkcji next().

Przykład

```
imie = "Reks"
it = iter(imie)
print(it)
# na wyjściu <str_iterator object at 0x000001CEB9A2F6D0>
print(next(it))# na wyjściu R
print(next(it))# na wyjściu e
print(next(it))# na wyjściu k
print(next(it))# na wyjściu s
print(next(it))# Traceback (most recent call last):
```

Przykład implementacji własnego iteratora.

```
class Wspak:
    """Iterator zwracający wartości w odwróconym porządku"""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

Generatory są prostymi narzędziami do tworzenia iteratorów. Generatory piszemy jak standardowe funkcje, ale zamiast instrukcji return używamy yield kiedy chcemy zwrócić wartość. Za każdym razem kiedy funkcja next() jest wywoływana na generatorze wznawia on swoje działanie w momencie, w którym został przerwany . Poniżej przykład generatora, którego działanie jest podobne do iteratora zaprezentowanego w przykładzie.

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]


gen = reverse("Feliks")
print(next(gen))
print("Marek")
print(next(gen))
```

Na wyjściu otrzymamy:

s

Marek

k

Podobny efekt możemy również osiągnąć poprzez wyrażenia generujące.

```
litery = (litera for litera in "Zdzisław")
print(litery)
print(next(litery))
```

Na wyjściu:

<generator object <genexpr> at 0x0000014F5FAF1E40>

Z