Python

Zaawansowane programowanie w języku Python

Radosław Roszczyk

27 luty 2022

radek@people.pl

O czym będzie

1. Paradygmat obiektowy
2. Klasy i obiekty
3. Dziedziczenie
4. Polimorfizm
5. Wirtualizacja
6. Hermetyzacja
7. Klasy abstrakcyjne

Paradygmat obiektowy

Dlaczego ?

W Pythonie wszystko jest obiektem !!!

Wprowadzenie

Programowanie obiektowe reprezentuje podejście do programowania pozwalającego na proste modelowanie elementów świata rzeczywistego przy użyciu kodu.

Początek

- Simula 67 – lata 60 XX w.
- Język C++ – główny rozwój OOP, lata 80 XX.

Języki wspierające OOP

- C++
- C#
- Java
- Python
- Ruby

Zalety programowania obiektowego

- **Hermetyzacja danych** – ukrycie danych wewnątrz obiektu przed innymi częściami systemu
- **Prostota** – obiektowość pozwala na łatwy podział dużych problemów na mniejsze
- **Łatwość modyfikacji** – znacznie łatwiej wyodrębnić części systemu które muszą zostać zmodyfikowane
- **Łatwość konserwacji** – obiekty ułatwiają znalezienie miejsc wymagających modyfikacji
- **Możliwość ponownego użycia** – definicje obiektów mogą być wykorzystane wielokrotne w systemie

Programowanie proceduralne a obiektowe

- W programowaniu proceduralnym rozwiązujemy problem poprzez kolejne wywoływanie różnych funkcji przy użyciu odpowiednich komend sterujących. W poszczególnych fazach wykonania programu używamy funkcji oraz zmiennych.
- W aspekcie obiektowym zasadniczo nie ma różnic. Zmianie podlega jedynie sposób widzenia świata. Zamiast przekazywać dane pomiędzy poszczególnymi etapami przetwarzania programu, próbujemy zmieniać stan tych danych posługując się metodami z nimi związanymi zamkniętymi w obiektach.
- W programowaniu obiektowym należy wyodrębnić artefakty oraz operacje z nimi związane.

Podstawowe cechy programowania obiektowego

- **Polimorfizm** – użycie klas potomnych w miejscu klas bazowych
- **Hermetyzacja** – ochrona dostępu do niektórych elementów obiektu przed innymi obiektami
- **Dziedziczenie** – możliwość wykorzystania metod bądź atrybutów obiektów bazowych przez obiekty nadrzędne
- **Wirtualizacja** – wywołanie metody z klasy danego typu z poziomu typu bazowego

Klasy i obiekty

Klasy i obiekty

Klasa jest typem danych, który jest szablonem dla definicji określonego rodzaju obiektu. Obiekt jest instancją danej klasy.

Klasa definiująca samochód

```
class Samochod:
  "Klasa definiujaca samochod"
  def __init__(self, marka, kolor, drzwi):
    self.marka = marka
    self.kolor = kolor
    self.drzwi = drzwi
```

Klasa definiująca punkt

```
class Punkt:
  "Klasa definiujaca punkt"

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

Obiekty na bazie klasy Samochod

```
ZoltaHonda = Samochod("Honda", "zolty", 4)
CzerwonaSkoda = Samochod("Skoda", "czerwony", 5)
```

Punkty na płaszczyźnie

```
PO = Punkt(0, 0)
A = Punkt(10, 5)
```

Obiekt

Konstruktor jest specjalną funkcją wewnątrz klasy, która jest wywoływana tylko raz podczas tworzenia obiektu. W tej metodzie jest możliwość tworzenia atrybutów obiektu.

Klasa definiująca samochód

```
1  class Samochod:
2    "Klasa definiujaca samochod"
3    def __init__(self):
4      self.marka = "Honda"
5      self.kolor = "czerwony"
6      drzwi = 5
7
8  samochod = Samochod()
9
10 print(samochod.marka)
11 print(samochod.kolor)
12 print(samochod.drzwi)
```

Właściwości obiektów

- konstruktor uruchamiany tylko raz
- atrybuty ustawiane przez konstruktor
- mogą posiadać metody
- właściwości obiektu – wywołują funkcje modyfikujące
- destruktor obiektu

Atrybuty oraz stałe

Każda klasa może posiadać dowolne atrybuty oraz stałe. Dostęp do atrybutów jest tylko z poziomu obiektu, stałe są dostępne zawsze.

Klasa definiująca samochód

```
1  class Samochod:
2    "Klasa definiujaca samochod"
3    Predkosc = 120
4    def __init__(self):
5      self.marka = "Honda"
6
7  samochod = Samochod()
8
9  print(samochod.Predkosc)
10 print(samochod.marka)
11
12 print(Samochod.Predkosc)
13 print(Samochod.marka)
```

Właściwości klas

- nie wymagają inicjalizacji
- ich wartość jest kopiowana z konstruktora
- są dostępne z poziomu klasy
- mogą być nadpisane w ramach obiektu

Metody

Każda klasa może posiadać dowolną liczbę metod. Metody są dostępne tylko z poziomu obiektu.

Klasa posiadająca metodę

```
1  class Samochod:
2    "Klasa definiujaca samochod"
3    Predkosc = 120
4    def __init__(self):
5      self.marka = "Honda"
6    def DrukujPredkosc(self):
7      print(f"{self.marka} - {self.Predkosc}")
8
9  honda = Samochod()
10 honda.DrukujPredkosc()
```

Właściwości metod

- należą do obiektów
- operują na danych obiektu
- można je wywoływać na zewnątrz i wewnątrz obiektu
- nie można ich wywoływać dla klas

Metody

Klasa posiadająca metodę

```
1  class Samochod:
2    "Klasa definiujaca samochod"
3    Predkosc = 120
4    def __init__(self):
5      self.marka = "Honda"
6    def DrukujPredkosc(self):
7      print(f"Samochod {self.marka} jezdzi z ąpredkosci {self.Predkosc} km/h")
8    def ZmienPredkosc(self, nowaPredkosc):
9      self.Predkosc = nowaPredkosc
10     #Predkosc = nowaPredkosc
11
12 honda = Samochod()
13 honda.DrukujPredkosc()
14 honda.ZmienPredkosc(400)
15 honda.DrukujPredkosc()
```

Destruktor

Każda klasa może posiadać jeden destruktor, który jest wywoływany przy usuwaniu klasy.

Klasa posiadająca destruktor

```
1  class Samochod:
2    "Klasa definiujaca samochod"
3    Predkosc = 120
4    def __init__(self):
5      self.marka = "Honda"
6    def DrukujPredkosc(self):
7      print(f"{self.marka} - {self.Predkosc}")
8    def __del__(self):
9      print("łZomuje samochod")
10
11 honda = Samochod()
12 honda.DrukujPredkosc()
13 del honda
```

Właściwości metod

- należą do obiektów
- operują na danych obiektu
- można je wywoływać na zewnątrz i wewnątrz obiektu
- nie można ich wywoływać dla klas

Tworzenie kontekstu

Dowolna klasa może utworzyć kontekst.

Klasa tworząca kontekst

```
1  class Samochod:
2    def __init__(self):
3      self.kluczyki = "Kluczyki do auta"
4      print("Produkuje samochod")
5    def __enter__(self):
6      print("Rejestruje samochod")
7      return self.kluczyki
8    def __exit__(self, type, value, tb):
9      print("Wyrejestrowywuje samochod")
10   def __del__(self):
11     print("Zlomuje samochod")
12
13 with Samochod() as samochod:
14   print(samochod)
```

Metody specjalne

- **__init__** – konstruktor obiektu
- **__enter__** – wywoływane przy tworzeniu kontekstu, zwraca obiekt kontekstowy
- **__exit__** – usuwanie kontekstu, parametry wykorzystywane są do zwracania błędu
- **__del__** – destruktor obiektu
- **__str__** – postać napisowa obiektu

Dziedziczenie

Dziedziczenie

Klasy dziedziczące po klasie bazowej

```
1  class Samochod:
2    def __init__(self, marka):
3      self.marka = marka
4    def CoToZaAuto(self):
5      print(f"to jest {self.marka}")
6
7  class Honda(Samochod):
8    def __init__(self):
9      Samochod.__init__(self, "Honda")
10
11 class Skoda(Samochod):
12   def __init__(self):
13     Samochod.__init__(self, "Skoda")
```

Mechanizmy

- klasa dziedzicząca po klasie bazowej zachowuje dostęp do „wszystkich” jej metod oraz atrybutów
- klasa dziedzicząca powinna wykonać konstruktor klasy bazowej
- klasa dziedzicząca powinna wykonać destruktor klasy bazowej
- Python pozwala na wielodziedziczenie – można dziedziczyć po więcej niż jednej klasie bazowej

Polimorfizm

Polimorfizm – czy słoń jest zwierzęciem?

Wirtualizacja metody naprzod

```
class Pojazd:
  def naprzod(self):
    pass

class Samochod(Pojazd):
  def naprzod(self):
    pass

class Okret(Pojazd):
  def naprzod(self):
    pass

class Amfibia(Samochod, Okret):
  def naprzod(self):
    pass
```

Mechanizm

Polimorfizm pozwala nam na wykorzystanie klasy dziedziczącej wszędzie tam gdzie może być użyta klasa bazowa. Oznacza to, że instancja klasy dziedziczącej jest uznawana za instancję klasy bazowej.

Czy amfibia jest pojazdem?

```
pojazd = Pojazd()
okret = Okret()
samochod = Samochod()
amfibia = Amfibia()

isinstance(amfibia, Pojazd)
```

Wirtualizacja

Wirtualizacja

Wirtualizacja metody naprzod

```
1  class Pojazd:
2    def naprzod(self):
3      pass
4    def jedziemy(self):
5      self.naprzod()
6
7  class Samochod(Pojazd):
8    def naprzod(self):
9      print("Samochod jedzie do przodu")
10
11 class Okret(Pojazd):
12   def naprzod(self):
13     print("Okret plynie do przodu")
14
15 pojazd = Okret()
16 pojazd.jedziemy()
```

Mechanizmy

- W obiekcie bazowym wywoływana jest metoda naprzód, która to jest przykryta wersją zależną od typu w obiekcie potomnym
- Wywoływana jest zawsze ostatnia wersja metody w łańcuchu dziedziczenia

Hermetyzacja

Hermetyzacja

Podstawowym założeniem hermetyzacj jest ukrycie danych wewnętrznych obiektu. Dane te nie powinny być dostępne z zewnątrz obiektu.

Do czego potrzebna jest hermetyzacja?

```
1  class Kwadrat:
2    def __init__(self):
3      self.wysokosc = 2
4      self.szerokosc = 2
5    def zmienBok(nowaDlugoscBoku):
6      self.wysokosc = nowaDlugoscBoku
7      self.szerokosc = nowaDlugoscBoku
8
9  kwadrat = Kwadrat()
10 kwadrat.wysokosc = 10
```

Mechanizmy

- Python nie posiada mechanizmów ochrony danych
- Poziomy dostępu są czysto umowne
- Atrybuty chronione poprzedzamy pojedynczym symbolem podłogi „_”
- Atrybuty prywatne poprzedzamy podwójnym symbolem podłogi „__”

Hermetyzacja

Umownymi modyfikatorami dostępu są znaki podłogi „_”

Pola chronione – protected

```
1  class Kwadrat:
2    def __init__(self):
3      self._wysokosc = 2
4      self._szerokosc = 2
5    def zmienBok(nowaDlugoscBoku):
6      self._wysokosc = nowaDlugoscBoku
7      self._szerokosc = nowaDlugoscBoku
8
9  kwadrat = Kwadrat()
10 kwadrat._wysokosc = 10
```

Pola prywatne – private

```
1  class Kwadrat:
2    def __init__(self):
3      self.__wysokosc = 2
4      self.__szerokosc = 2
5    def zmienBok(nowaDlugoscBoku):
6      self.__wysokosc = nowaDlugoscBoku
7      self.__szerokosc = nowaDlugoscBoku
8
9  kwadrat = Kwadrat()
10 kwadrat.__wysokosc = 10
```

Dobra praktyka – metody pobierające i ustawiające

Jak to działa?

```
1  class Kwadrat:
2    def __init__(self):
3      self.__wysokosc = 2
4      self.__szerokosc = 2
5    def zmienBok(nowaDlugoscBoku):
6      self.__wysokosc = nowaDlugoscBoku
7      self.__szerokosc = nowaDlugoscBoku
8    @property
9    def wysokosc(self):
10     return self.__wysokosc;
11   @wysokosc.setter
12   def wysokosc(self, w):
13       self.__wysokosc = w
14
15 kwadrat = Kwadrat()
16 kwadrat.wysokosc
```

Jak to działa?

- Dekorator **property** definiuje metodę odczytującą właściwość obiektu
- Dekorator **wysokosc.setter** definiuje metodę ustawiającą właściwość obiektu
- Metody mogą wykonywać dowolne operacje podczas pobierania bądź ustawiania wartości

Klasy abstrakcyjne

Abstrakcja

Abstrakcja pozwala nam tworzyć klasy które nie mogą posiadać instancji jednocześnie wymusza implementacje wszystkich metod abstrakcyjnych w klasach potomnych

Przykład

```
1  class Figura(ABC):
2    @abstractmethod
3    def PolePowierzchni(self):
4      pass
5
6  class Kwadrat(Figura):
7    def __init__(self):
8      self.__wysokosc = 2
9      self.__szerokosc = 2
10   def zmienBok(nowaDlugoscBoku):
11     self.__wysokosc = nowaDlugoscBoku
12     self.__szerokosc = nowaDlugoscBoku
13   def PolePowierzchni(self):
```

Mechanizmy

- Importujemy Abstract Base Classes (ABC)
- Klasy powinny dziedziczyć po ABC
- Dekorator **abstractmethod**

Pytania?