Programowanie obiektowe

Paradygmat Obiektowy

W programowaniu istnieje kilka popularnych paradygmatów (idei programowania), są to między innymi:

- imperatywny,
- deklaratywny,
- funkcjonalny,
- proceduralny,
- obiektowy,

Paradygmat imperatywny oznacza, że używane są instrukcje, które zmieniają stan programu. Lista poleceń do wypełnienia przez komputer. Przykładami języków imperatywnych są C++, Python, Java i wiele innych. Paradygmat deklaratwny pozwala budować programy opisując pożądany efekt zamiast kolejnych procedur, przykładem takiego języka jest HTML, w którym opisujemy jak ma strona wyglądać, nie wgłębiając się w to jak ten efekt zostanie osiągnięty. Paradygmat funcjonalny oznacza, że wykorzystywane są jedynie funkcje, które zawsze zwracają tą samą wartość dla tych samych argumentów. Nacisk jest wtedy kładziony na matematyczny opis funkcji. Jedną z istotnych zalet tego paradygmatu jest możliwość matematycznego udowodnienia skuteczności programu. Paradygmat proceduralny polega na tym, że program wykonuje listę procedur, które to procedury są zgrupowane w byty, które można nazwać funkcjami. W tym przypadku nie jest to jendak funkcja matematyczna (jak w przypadku paradygmatu funkcjonalnego), a lista poleceń i komend.

Paradygmat obiektowy polega na tym, że program manipuluje złożonymi obiektami, z których każdy ma swój własny stan i ten stan można modyfikować metodami przypisanymi do tego obiektu. Paradygmat obiektówy pozwala pisać bardzo przejrzysty kod i zrozumiały kod.

Dziedziczenie

```
1  class Pojazd:
2      marka = None
3      kierowca = None
4      kola = 4
5
6  class Samochod(Pojazd):
7      marka = None
8      kierowca = {'imie': 'Hori', 'nazwisko': 'Kamatani'}
9
10 class Motor(Pojazd):
11     marka = 'honda'
12     kola = 2
```

Wielodziedziczenie

```python
class Pojazd:
    marka = None

class Samochod(Pojazd):
    marka = None
    kierowca = {'imie': 'Hori', 'nazwisko': 'Kamatani'}

class Jeep(Samochod):
    marka = 'jeep'

class Star(Samochod):
    marka = 'star'
```

Kompozycja

```python
class OtwieralneSzyby:
    def otworz_szyby(self):
        raise NotImplementedError


class OtwieralnyDach:
    def otorz_dach(self):
        raise NotImplementedError


class UmieTrabic:
    def zatrab(self):
        print('\bbiip')


class Pojazd:
    kola = None


class Samochod(Pojazd, UmieTrabic, OtwieralneSzyby):
    kola = 4

    def wlacz_swiatla(self, *args, **kwargs):
        print('włączam światła')


class Cabrio(Samochod, OtwieralnyDach):
    def wlacz_swiatla(self, *args, **kwargs):
        print('Podnieś obudowę lamp')
        print('Puść muzyzkę')
        super(Cabrio, self).wlacz_swiatla(*args, **kwargs)
        print('Zatrąb')


class Motor(Pojazd, UmieTrabic):
    kola = 2

```

```
38
39 c = Cabrio()
40 c.wlacz_swiatla()
```

```
1  class OtwieralnyDach:
2      def otworz_dach(self):
3          pass
4
5      def zamknij_dach(self):
6          pass
7
8
9  class Trabi:
10     def zatrab(self):
11         raise NotImplementedError
12
13
14
15 class Pojazd:
16     kola = None
17
18
19 class Samochod(Pojazd):
20     kola = 4
21
22
23 class Motor(Pojazd, Trabi):
24     kola = 2
25
26     def zatrab(self):
27         print('biip')
28
29
30 class Cabriolet(Samochod, OtwieralnyDach, Trabi):
31     def zatrab(self):
32         print('tru tu tu tu')
33
34
35 class Mercedes(Samochod, OtwieralnyDach, Trabi):
36     pass
37
38
39 class Maluch(Samochod, Trabi):
40     pass
```

Dziedziczenie czy kompozycja?

- Kompozycja ponad dziedziczenie!

Polimorfizm

```
1  >>> class Pojazd:
```

```
...     def zatrab(self):
...         raise NotImplementedError
...
>>> class Motor(Pojazd):
...     def zatrab(self):
...         print('bip')
...
>>> class Samochod(Pojazd):
...     def zatrab(self):
...         print('biiiip')
...
>>> obj = Motor()
>>> obj.zatrab()
>>>
>>> obj = Samochod()
>>> obj.zatrab()
```

Klasy abstrakcyjne

Klasa abstrakcyjna to taka klasa, która nie ma żadnych instancji (w programie nie ma ani jednego obiektu, który jest obiektem tej klasy). Klasy abstrakcyjne są uogólnieniem innych klas, wykorzystuje się to często przy dziedziczeniu. Na przykład tworzy się najpierw abstrakcyjną klasę `figura`, która definiuje, że figura ma pole oraz, że jest metoda, która to pole policzy na podstawie jedynie prywatnych zmiennych. Po klasie `figura` możemy następnie dziedziczyć tworząc klasy `kwadrat` oraz `trójkąt`, które będą miały swoje instancje i na których będziemy wykonywali operacje.

Składnia

Klasy

```
class Pojazd:
    marka = None
    kierowca = None
    kola = 4
```

```
class Samochod:
    def __init__(self, marka, kola=4):
        self.marka = marka
        self.kola = kola

auto = Samochod(marka='mercedes', kola=3)
print(auto.kola)
```

Metody

`self`

Pola klasy

```python
import logging


class Samochod:
    kola = 4
    marka = None

    def set_marka(self, marka):
        logging.warning('Ustawiamy marke')
        self.marka = marka

    def get_marka(self):
        return self.marka


mercedes = Samochod()
mercedes.set_marka('Mercedes')
print(mercedes.get_marka())


maluch = Samochod()
maluch.marka = 'Maluch'
print(maluch.marka)


maluch = Samochod(marka='Maluch')
print(maluch.marka)
```

Funkcja inicjalizująca

`__init__` jest metodą klasy, która wykonuje się podczas tworzenia nowego obiektu. Nie jest to do końca konstruktor tego obiektu, ale dla większości zastosowań można przyjąć, że metoda `__init__` jest konstruktorem klasy.

```python
import logging

class Samochod:
    kierowca = None

    def __init__(self, marka, kola=4):
        logging.warning('inicjalizujemy obiekt %s', marka)
        self.marka = marka
        self.kola = kola


sam1 = Samochod(marka='Maluch')
print(sam1.marka)
print(sam1.kola)

print(dir(sam1))
print(sam1.__dict__)

```

```
19
20 sam2 = Samochod(marka='Merc')
21 print(sam2.marka)
22 print(sam2.kola)
```

`super()`

Funkcja `super` pozwala uzyskać dostęp do obiektu po którym dziedziczymy, do jego parametrów statycznych i metod, które przeciążamy (m.in. funkcji `__init__`).

```
1  class Human:
2      def __init__(self):
3          self.short_species = 'human'
4      species = 'Homo Sapiens'
5
6  class Man(Human):
7      def __init__(self, name='man'):
8          super().__init__()
9          self.name = name
10     def my_parent(self):
11         print(super().species)
12     def get_my_species(self):
13         print(self.species)
14
15 print(John.short_species)
16 John.my_parent()
17
18 John.species
19 John.get_my_species()
```

`@property` i `@x.setter`

Dekoratory `@propery`, `@x.setter` i `@x.deleter` służą do zdefiniowania dostępu do 'prywatnych' pól klasy. W Pythonie z definicji nie ma czegoś takiego jak pole prywatne. Jest natomiast konwencja nazywania zmiennych zaczynając od symbolu podkreślnika (np. `_x`), jeżeli chcemy zaznaczyć, że to jest zmienna prywatna. Nic nie blokuje jednak użytkownika przed dostępem do tej zmiennej. Dekoratory `@x.setter` i `@property` tworzą metody do obsługi zmiennej `_x` (w przykładzie poniżej).

```
1  class Cls:
2      def __init__(self):
3          self._x = None
4
5      @property
6      def x(self):
7          """I'm the 'x' property."""
8          print("I'm the x property!")
9          return self._x
10
11     @x.setter
12     def x(self, value):
13         print("The x setter has been called!")
```

```
        self._x = value

    @x.deleter
    def x(self):
        del self._x

new_object = Cls()
print(new_object.x)
new_object.x = 1
print(new_object.x)
```

`@staticmethod`

Dekorator `@staticmethod` służy do tworzenia metod statycznych, takich które odnoszą się do klasy jako całości, nie do konkretnego obiektu.

```
class Person:
    population = 0

    def __init__(self, name = 'NN'):
        self.name = name
        Person.increment_population()

    @staticmethod
    def increment_population():
        Person.population += 1

    @staticmethod
    def get_population():
        return Person.population


Anna = Person("Anna")
John = Person("John")

Person.get_population()
```

`__str__()` i `__repr__()`

Dwiema dość często używanymi metodami systemowymi są `__repr__` i `__str__`. Obie te funkcje konwertują obiekt klasy do stringa, mają jednak inne przeznaczenie: * cel `__repr__` to być jednoznacznym, * cel `__str__` to być czytelnym.

Albo jeszcze inaczej - `__repr__` jest dla developerów, `__str__` dla użytkowników.

```
class Samochod:
    def __init__(self, marka, kola=4):
        self.marka = marka
        self.kola = kola

    def __str__(self):
```

```
        return f'Marka: {self.marka} i ma {self.kola} koła'

    def __repr__(self):
        return f'Samochód(marka: {self.marka}, kola: {self.kola})'


Samochod(marka='mercedes', kola=3)

auto = Samochod(marka='mercedes', kola=3)
print(auto)

auta = [
    Samochod(marka='mercedes', kola=3),
    Samochod(marka='maluch', kola=4),
    Samochod(marka='fiat', kola=4),
]

print(auta)
```

```
import datetime

datetime.datetime.now() # wyświetli w konsoli napis zdefiniowany przez ``__repr__``
print(datetime.datetime.now()) # wyświetli w konsoli napis zdefiniowany przez
``__str__``
```

Metaclass

Każdy obiekt klasy jest instankcją tej klasy. Każda napisana klasa jest instancją obiektu, który nazywa się metaklasą. Domyślnie klasy są obiektem typu `type`

```
class FooClass:
    pass

f = FooClass()
isinstance(f, FooClass)
isinstance(f, type)
```

Przeciążanie operatorów

Python implementuje kilka funkcji systemowych (magic methods), zaczynających się od podwójnego podkreślnika. Są to funkcje wywoływane m.in podczas inicjalizacji obiektu (`__init__`). Innym przykładem może być funkcja `obiekt1.__add__(obiekt2)` , która jest wywoływana gdy wykonamy operację `obiekt1 + obiekt2` .

Poniżej przedstawiono kilka przykładów metod magicznych w Pythonie.

`__add__()`

```
class Vector:
    def __init__(self, x=0.0, y=0.0):
```

```
        self.x = x
        self.y = y

    def __abs__(self):
        return (self.x**2 + self.y**2)**0.5

    def __str__(self):
        return f"<{self.x}, {self.y}>"

    def __repr__(self):
        return f"Vector: [x: {self.x}, y: {self.y}]"

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
```

`__eq__()`

`__ne__()`

`__lt__()`

`__le__()`

`__gt__()`

`__ge__()`

Dobre praktyki

Ask don't tell

"Tell-Don't-Ask is a principle that helps people remember that object-orientation is about bundling data with the functions that operate on that data. It reminds us that rather than asking an object for data and acting on that data, we should instead tell an object what to do. This encourages to move behavior into an object to go with the data."

Inicjalizacja parametrów

Wszystkie parametry lokalne dla danej instancji klasy powinny być zainicjalizowane w funkcji `__init__`.

Private, public? konwencja `_` i `___`

W Pythonie nie ma czegoś takiego jak prywatne pole klasy. Czy prywatna metoda klasy. Wszystkie obiekty zdefiniowane wewnątrz klasy są publiczne. Istnieje jednak ogólnie przyjęta konwencja, że obiekty poprzedzone `_` są prywatne dla tej klasy i nie powinny być bezpośrednio wywoływane przez użytkownika. Podobnie z funkcjami rozpoczynającymi się od `___` (m.in. metody magiczne wspomniane powyżej). Są tu funkcje systemowe, które są używane przez interpreter Pythona i raczej nie powinny być używane bezpośrednio.

Co powinno być w klasie a co nie?

```
class Osoba:
    wiek = 10

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

    @staticmethod
    def powiedz_hello():
        print('hello')


Osoba.powiedz_hello()
print(Osoba.wiek)


o = Osoba(imie='Ivan')
o.powiedz_hello()
print(Osoba.wiek)
```

Klasa per plik?

Przykłady praktyczne

```
>>> class Osoba:
...     nazwisko = 'Kamatani'
...
...     def __init__(self, imie):
...         self.imie = imie

>>> o1 = Osoba('Hori')
>>> o2 = Osoba('Kazu')


>>> print(o1.nazwisko)
Kamatani

>>> print(o2.nazwisko)
Kamatani



>>> o1.nazwisko = 'Hakiai'

>>> print(o1.nazwisko)
Hakiai

>>> print(o2.nazwisko)
Kamatani

```

```
>>> Osoba.nazwisko = 'Asai'

>>> print(o1.nazwisko)
Hakiai

>>> print(o2.nazwisko)
Asai
```

Zadania kontrolne

Punkty i wektory

Przekształć swój kod z przykładu z modułu "Matematyka" tak żeby wykorzytywał klasy.

Zadanie 0

: Napisz klasę `ObiektGraficzny`, która implemtuje "wirtualną" funkcję `plot()`. Niech domyślnie ta funkcja podnosi `NotImplementedError` (podpowiedź: `raise NotImplementedError`).

Zadanie 1

:

> Napisz klasę `Punkt`, która dziedziczy po `ObiektGraficzny`, która będzie miała "ukryte" pola `_x`, `_y`. Konstruktor tej klasy ma przyjmować współrzędne `x` oraz `y` jako argumenty. Napisz obsługę pól ukrytych `_x` oraz `_y` jako `@property` tej klasy (obsługiwane jako `x` oraz `y`). Dopisz implementacje metod `__str__` oraz `__repr__`. Zaimplementuj metodę `plot(kolor)`, która wyrysuje ten punkt na aktualnie aktywnym wykresie. Kolor domyślnie powinien przyjmować wartość `'black'`.
>
> Dopisz do tej klasy metodę statyczną, która zwróci losowy punkt w podobny sposób jak funkcja `random_point(center, std)` zwracała obiekt dwuelementowy.

Zadanie 2

:

> Dopisz do tej klasy dwie metody, które pozwolą obliczyć odległość między dwoma punktami. Jedna z tych metod niech będzie metodą statyczną, która przyjmuje dwa punkty jako argumenty, a zwraca odległość między nimi (przykładowe wywołanie tej metody: `Punkt.oblicz_odleglosc_miedzy_punktami(punkt_A, punkt_B)`). Druga z tych metod niech będzie zwykłą metodą klasy, która przyjmie jeden punkt jako argument oraz obliczy odległość od tego punktud opunktu na którym jest wykonywana (`punkt_A.oblicz_odleglosc_do(punkt_B)`).

Zadanie 3

:

> Napisz kod, który wykorzystując klasę zaimplementowaną w przykładzie powyżej, wygeneruje listę losowych punktów wokół punktów A i B. Wyrysuj te punkty na wykresie, podobnie jak w przykładzie z modułu "Matematyka".

Zadanie 4

:

> Napisz kod, który zaklasyfikuje te losowo wygenerowane punkty do punktów A oraz B na podstawie odległości. W tym celu wykorzystaj napisane metody do obliczania odległości między punktami. Po klasyfikacji wyrysuj te punkty na wykresie, podobnie jak w przykładzie z modułu "Matematyka".

Książka adresowa

Zadanie 1

: Zmień swój kod zadania z książką adresową, aby każdy z kontaktów był reprezentowany przez:

```
> -   imię
> -   nazwisko
> -   telefon
> -   adresy:
>
>     > -   ulica
>     > -   miasto
>     > -   kod\_pocztowy
>     > -   wojewodztwo
>     > -   panstwo
>
-   Wszystkie dane w książce muszą być reprezentowane przez klasy.
-   Klasa osoba powinna wykorzystywać domyślne argumenty w `__init__`.
-   Użytkownik może mieć wiele adresów.
-   Klasa adres powinna mieć zmienną liczbę argumentów za pomocą `**kwargs` z domyślnymi wartościami.
-   Zrób tak, aby się ładnie wyświetlało. Zarówno dla jednego wyniku (`print(adres)`, `print(osoba)` jak i dla wszystkich w książce `print(ksiazka_adresowa)`.
-   API programu powinno być tak jak na listingu poniżej

``` python
ksiazka_adresowa = [
    Kontakt(imie='Max', nazwisko='Peck', adresy=[
        Adres(ulica='...', miasto='...'),
        Adres(ulica='...', miasto='...'),
        Adres(ulica='...', miasto='...'),
    ]),
    Kontakt(imie='José', nazwisko='Jiménez'),
    Kontakt(imie='Иван', nazwisko='Иванович', adresy=[]),
]
```
```

Zadanie 2

: Napisz książkę adresową, która będzie zapisywała a później odczyta i sparsuje dane do pliku w formacie Pickle.

Zadanie 3

: Napisz książkę adresową, która będzie zapisywała a później odczyta i sparsuje dane do pliku w formacie JSON.

Podpowiedź

: - Dane w formacie Pickle muszą być zapisane do pliku binarnie - `pickle.loads()` przyjmuje uchwyt do pliku, a nie jego zawartość