Język skryptowy lab5

Programowanie obiektowe

Wcześniej zostały przeanalizowane dwa paradygmaty programowania - imperatywne (używając instrukcji, pętli i funkcji jako podprogramów) i funkcyjne (używając czystych funkcji, funkcji wyższego rzędu i rekursji).

Innym bardzo popularnym paradygmatem jest programowanie obiektowe (OOP). Obiekty są tworzone za pomocą klas, które są w rzeczywistości centralnym punktem OOP. Klasa opisuje, czym będzie obiekt, ale jest oddzielona od samego obiektu. Innymi słowy, klasa może być opisana jako projekt, opis lub definicja obiektu. Możesz użyć tej samej klasy jako plan tworzenia wielu różnych obiektów.

Klasy są tworzone przy użyciu słowa kluczowego `class` i wciętych bloków, które zawierają metody klas (które są funkcjami). Poniżej znajduje się przykład prostej klasy i jej obiektów.

```
class Cat:
    def __init__(self, color, legs):
        self.color = color
        self.legs = legs


felix = Cat("ginger", 4)
rover = Cat("dog-colored", 4)
stumpy = Cat("brown", 3)

print(felix.color)
```

> Ten kod definiuje klasę o nazwie `Cat`, która ma dwa atrybuty: *color* i *legs*. Następnie klasa służy do tworzenia trzech oddzielnych obiektów (reprezentantów) tej klasy.

`__init__`

Metoda `__init__` jest najważniejszą metodą w klasie. Jest ona wywoływana, gdy tworzona jest instancja (obiekt) klasy, używając nazwy klasy jako funkcji.

Wszystkie metody muszą mieć `self` jako pierwszy parametr, chociaż nie jest on jawnie przekazywany, Python dodaje za ciebie argument `self` - nie trzeba go uwzględniać, gdy wywołuje się metode. W definicji metody self odnosi się do instancji wywołującej metodę.

Instancje klasy mają atrybuty, które są związanymi z nimi fragmentami danych. W tym przykładzie instancje klasy `Cat` mają atrybuty *color* i *legs*. Dostęp do nich można uzyskać, umieszczając kropkę i

nazwę atrybutu po instancji. W metodzie `__init__` można użyć `self.atrybut` do ustawienia początkowej wartości atrybutów instancji.

> W powyższym przykładzie metoda `__init__` pobiera dwa argumenty i przypisuje je do atrybutów obiektu. Metoda `__init__` jest konstruktorem klasy.

Metody

Klasy mogą mieć zdefiniowane inne metody w celu dodania do nich funkcjonalności. Należy pamiętać, że wszystkie metody muszą mieć `self` jako pierwszy parametr. Dostęp do tych metod można uzyskać za pomocą taki sam sposób jak do atrybutów, czyli przy pomocy składni z kropką.

```
class Dog:
  def __init__(self, name, color):
    self.name = name
    self.color = color

  def bark(self):
    print("Woof!")

fido = Dog("Fido", "brown")
print(fido.name)
fido.bark()
```

Klasy mogą również posiadać atrybuty klas, utworzone przez przypisanie zmiennych w treści klasy. Dostęp do nich można uzyskać z instancji klasy lub samej klasy.

```
class Dog:
    legs = 4
    def __init__(self, name, color):
        self.name = name
        self.color = color

fido = Dog("Fido", "brown")
print(fido.legs)
print(Dog.legs)
```

> Atrybuty klas są wspólne dla wszystkich instancji klasy.

Próba dostępu do atrybutu instancji, który nie jest zdefiniowany, powoduje wystąpienie błędu `AttributeError`. Dotyczy to również wywołania niezdefiniowanej metody.

```
class Rectangle:
  def __init__(self, width, height):
    self.width = width
    self.height = height
```

```
rect = Rectangle(7, 8)
print(rect.color)
```

Dziedziczenie

Dziedziczenie zapewnia sposób udostępniania funkcji między klasami. Wyobraź sobie kilka klas, `Cat`, `Dog`, `Rabbit` i tak dalej. Chociaż mogą się różnić pod pewnymi względami (tylko pies może mieć taką metodę), prawdopodobnie będą podobne w innych (wszystkie mają atrybuty kolor i imię). Podobieństwo to można wyrazić, czyniąc je wszystkie dziedziczącymi z superklasy `Animal`, która zawiera wspólną funkcjonalność. Aby dziedziczyć klasę z innej klasy, umieść nazwę nadklasy w nawiasach po nazwie klasy.

```
class Animal:
    def __init__(self, name, color):
        self.name = name
        self.color = color

class Cat(Animal):
    def purr(self):
        print("Purr...")

class Dog(Animal):
    def bark(self):
        print("Woof!")

fido = Dog("Fido", "brown")
print(fido.color)
fido.bark()
```

Klasa, dziedzicząca z innej klasy nazywana jest podklasą. Klasa po której dziedziczą inne klasy nazywana jest superklasą. Jeśli klasa dziedziczy po innej z tymi samymi atrybutami lub metodami, nadpisuje je.

```
class Wolf:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def bark(self):
        print("Grr...")

class Dog(Wolf):
    def bark(self):
        print("Woof")
```

```
husky = Dog("Max", "grey")
husky.bark()
```

> W powyższym przykładzie `Wolf` jest superklasą, `Dog` jest podklasą.

Dziedziczenie może być również pośrednie. Jedna klasa może dziedziczyć od innej i ta klasa może dziedziczyć z trzeciej klasy.

```
class A:
    def method(self):
        print("A method")

class B(A):
    def another_method(self):
        print("B method")

class C(B):
    def third_method(self):
        print("C method")

c = C()
c.method()
c.another_method()
c.third_method()
```

> Jednak nie jest możliwe dziedziczenie w koło.

Funkcja *super* jest użyteczną funkcją związaną z dziedziczeniem, która odnosi się do klasy nadrzędnej. Można jej użyć do znalezienia metody o określonej nazwie w nadklasie obiektu.

```
class A:
    def spam(self):
        print(1)

class B(A):
    def spam(self):
        print(2)
        super().spam()

B().spam()
```

> `super().spam()` wywołuje metodę `spam` z nadklasy.

Magiczne metody

Magiczne metody są specjalnymi metodami, które mają podwójne podkreślenia na początku i na końcu ich nazw.

Jak dotąd jedynym, z którym się spotkaliśmy, jest `__init__`, ale jest jeszcze kilka innych. Są używane do tworzenia funkcjonalności, które nie mogą być reprezentowane jako normalna metoda.

Jednym z ich powszechnych zastosowań jest przeciążanie operatorów. Oznacza to zdefiniowanie operatorów dla niestandardowych klas, które pozwalają na użycie operatorów takich jak `+` i `*`. Przykładową magiczną metodą jest `__add__` dla `+`.

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)

    def toStr(self):
        return self.x, self.y

first = Vector2D(5, 7)
second = Vector2D(3, 9)
result = first + second
print(result.toStr())
```

> Metoda `__add__` pozwala na zdefiniowanie niestandardowego zachowania operatora `+` w klasie. Jak widać, dodaje odpowiednie atrybuty obiektów i zwraca nowy obiekt, zawierający wynik. Po zdefiniowaniu można razem dodać dwa obiekty danej klasy.

Metoda	Operator
`__sub__`	-
`__mul__`	*
`__truediv__`	/
`__floordiv__`	//
`__mod__`	%
`__pow__`	**
`__and__`	&
`__xor__`	^
`__or__`	\

Wyrażenie `x + y` jest tłumaczone na `x.__add__(y)`.

```
class SpecialString:
    def __init__(self, cont):
        self.cont = cont

    def __truediv__(self, other):
        line = "=" * len(other.cont)
        return "\n".join([self.cont, line, other.cont])


spam = SpecialString("spam")
hello = SpecialString("Hello world!")
print(spam / hello)
```

Python dostarcza również magiczne metody do porównań.

Metoda	Operator
`__lt__`	<
`__le__`	<=
`__eq__`	==
`__ne__`	!=
`__gt__`	>
`__ge__`	>=

Jeśli `__ne__` nie jest zaimplementowane, zwraca przeciwieństwo `__eq__`. Pomiędzy innymi operatoramin nie ma żadnych innych relacji.

```
class SpecialString:
    def __init__(self, cont):
        self.cont = cont

    def __gt__(self, other):
        for index in range(len(other.cont) + 1):
            result = other.cont[:index] + ">" + self.cont
            result += ">" + other.cont[index:]
            print(result)

spam = SpecialString("spam")
eggs = SpecialString("eggs")
spam > eggs
```

> Jak widać, można zdefiniować dowolne niestandardowe zachowanie dla przeciążonych operatorów.

Istnieje kilka magicznych metod tworzenia klas, które działają jak kontenery. `__len__` dla `len()` `__getitem__` dla indeksowania `__setitem__` dla przypisania do wartości indeksowanych `__delitem__` do usuwania indeksowanych wartości `__iter__` dla iteracji obiektów (np. dla pętli) `__contains__` dla sprawdzenia zawierania

Istnieje wiele innych metod magicznych, takich jak `__call__` dla wywoływania obiektów jako funkcji i `__int__` , `__str__` , i tym podobne, do konwersji obiektów na typy wbudowane.

```
import random

class VagueList:
    def __init__(self, cont):
        self.cont = cont

    def __getitem__(self, index):
        return self.cont[index + random.randint(-1, 1)]

    def __len__(self):
        return random.randint(0, len(self.cont) * 2)

vague_list = VagueList(["A", "B", "C", "D", "E"])
print(len(vague_list))
print(len(vague_list))
print(vague_list[2])
print(vague_list[2])
```

> Funkcja `len()` klasy VagueListz ostała przesłonięta, aby zwrócić losową liczbę. Funkcja indeksowania zwraca również losowy element z zakresul listy na podstawie wyrażenia.

Cykl życia obiektów

Cykl życia obiektu składa się z jego tworzenia, manipulacji i destrukcji.

Pierwszym etapem cyklu życia obiektu jest definicja klasy, do której należy. Następnym etapem jest tworzenie instancja danego typu, gdy wywoływana jest funkcja `__init__` . Pamięć jest przydzielona aby przechowywać utworzoną instancję. Tuż przed tym zdarzeniem wywoływana jest metoda `__new__` klasy. Jest ona przesłaniana tylko w specjalnych przypadkach. Po tym zdarzeniu obiekt jest gotowy do użycia.

> Inny kod może następnie wchodzić w interakcję z obiektem, wywołując na nim funkcje i uzyskując dostęp do jego atrybutów. W końcu zostanie wykorzystany i może zostać zniszczony.

Gdy obiekt zostanie zniszczony, pamięć przydzielona do niego zostanie zwolniona i może być wykorzystana do innych celów. Zniszczenie obiektu następuje, gdy jego licznik odniesienia(referencji) osiągnie zero. Liczba referencyjna to liczba zmiennych i innych elementów odnoszących się do obiektu. Jeśli nic nie odnosi się do obiektu (ma zerową wartość odniesienia), nic nie może z nim współdziałać, więc można go bezpiecznie usunąć.

W niektórych sytuacjach dwa (lub więcej) obiekty mogą być przywoływane tylko przez siebie, dlatego też można je usunąć. Instrukcja **del** zmniejsza liczbę odwołań obiektu o jeden, co często prowadzi do jego usunięcia. Magiczną metodą jest `__del__`. Liczba odwołań do obiektu wzrasta, gdy jest przypisana nowa nazwa lub umieszczona w kontenerze (liście, krotce lub słowniku). Liczba odwołań obiektu zmniejsza się, gdy zostanie usunięta za pomocą metody `__del__`, odniesienie zostanie ponownie przypisane lub jego odwołanie wykracza poza zakres. Kiedy licznik odwołań do obiektu osiąga zero, Python automatycznie go usuwa.

Ukrywanie danych - enkapsulacja

Kluczową częścią programowania obiektowego jest enkapsulacja, która polega na pakowaniu powiązanych zmiennych i funkcji w pojedynczy, łatwy w użyciu obiekt - instancję klasy. Powiązaną koncepcją jest ukrywanie danych, które stwierdza, że szczegóły implementacji klasy powinny być ukryte, a dla tych, którzy chcą korzystać z klasy, prezentowany jest czysty standardowy interfejs. W innych językach programowania zwykle odbywa się to za pomocą prywatnych metod i atrybutów, które blokują zewnętrzny dostęp do określonych metod i atrybutów w klasie.

Filozofia Pythona jest nieco inna. Często jest to określone jako "wszyscy tutaj jesteśmy dorośli", co oznacza, że nie powinno się nakładać arbitralnie ograniczeń dostępu do części klasy. Dlatego nie ma sposobów na wymuszenie aby metoda lub atrybut, był ściśle prywatny.

> Istnieją jednak sposoby zniechęcenia ludzi do uzyskiwania dostępu do części klasy, na przykład poprzez oznaczenie, że jest to szczegół implementacji i będą one używane na własne ryzyko.

Słabo prywatne metody i atrybuty mają na początku jedno podkreślenie. Oznacza to, że są one prywatne i nie powinny być używane przez zewnętrzny kod. Jednak jest to głównie konwencja i nie blokuje dostępu do kodu zewnętrznego. Jego jedynym faktycznym skutkiem jest to, że `from module_name import *` nie importuje zmiennych rozpoczynających się od pojedynczego podkreślenia.

```
class Queue:
    def __init__(self, contents):
        self._hiddenlist = list(contents)

    def push(self, value):
        self._hiddenlist.insert(0, value)

    def pop(self):
        return self._hiddenlist.pop(-1)

    def __repr__(self):
        return "Queue({})".format(self._hiddenlist)

queue = Queue([1, 2, 3])
print(queue)
queue.push(0)
print(queue)
queue.pop()
```

```
print(queue)
print(queue._hiddenlist)
```

> W powyższym kodzie atrybut _hiddenlist jest oznaczony jako prywatny, ale wciąż można uzyskać do niego dostęp w kodzie zewnętrznym. Metoda magiczna **repr** jest używana do reprezentacji ciągów instancji.

Silnie prywatne metody i atrybuty mają **podwójne podkreślenie** na początku ich nazw. To powoduje, że ich nazwy są zniekształcone, co oznacza, że nie można uzyskać do nich dostępu spoza klasy. Celem tego nie jest zapewnienie, że są one prywatne, ale unikanie błędów, gdy istnieją podklasy, które mają metody lub atrybuty o tych samych nazwach.

Zniekształcone nazwy metod mogą być nadal dostępne zewnętrznie, ale pod inną nazwą. Metoda `__privatemethod` klasy `Spam` może być dostępna zewnętrznie poprzez `_Spam__privatemethod`.

```
class Spam:
    __egg = 7

    def print_egg(self):
        print(self.__egg)

s = Spam()
s.print_egg()
print(s._Spam__egg)
print(s.__egg)
```

> Zasadniczo Python chroni metody i atrybuty, przez wewnętrzne zmiany nazw, aby zawierały nazwę klasy.

Metody klas

Metody obiektów, które przyjrzeliśmy się do tej pory, są wywoływane przez instancję klasy, która jest następnie przekazywana do parametru ***self*** metody. Metody klas są różne - są wywoływane przez klasę, która jest przekazywana do parametru ***cls*** metody. Powszechnym ich zastosowaniem są metody wytwórcze, które tworzą instancję klasy, używając innych parametrów niż te zwykle przekazywane do konstruktora klasy. Metody klas są oznaczone adnotacją ***classmethod***.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

    @classmethod
    def new_square(cls, side_length):
```

```
        return cls(side_length, side_length)

square = Rectangle.new_square(5)
print(square.calculate_area())
```

new_square jest metodą klasy i jest wywoływany w klasie, a nie w instancji klasy. Zwraca nowy obiekt klasy ***cls***.

> Technicznie, parametry `self` i `cls` są po prostu konwencjami; można je zmienić na cokolwiek innego. Są one jednak powszechnie stosowane, więc rozsądnie jest trzymać się ich używania.

Metody statyczne

Metody statyczne są podobne do metod klasowych, z tym wyjątkiem, że nie otrzymują żadnych dodatkowych argumentów; są one identyczne z normalnymi funkcjami należącymi do klasy. Są one oznaczone adnotacją ***staticmethod***.

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    @staticmethod
    def validate_topping(topping):
        if topping == "pineapple":
            raise ValueError("No pineapples!")
        else:
            return True

ingredients = ["cheese", "onions", "spam"]
if all(Pizza.validate_topping(i) for i in ingredients):
    pizza = Pizza(ingredients)
```

Właściwości

Właściwości zapewniają sposób dostosowywania dostępu do atrybutów instancji. Są one tworzone przez umieszczenie adnotacji ***property*** nad metodą, co oznacza, że gdy zostanie wywołany atrybut instancji o tej samej nazwie co metoda, metoda zostanie wywołana. Jednym z typowych zastosowań właściwości jest ustawienie atrybutu jako atrybutu "tylko do odczytu".

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    @property
    def pineapple_allowed(self):
        return False
```

```
pizza = Pizza(["cheese", "tomato"])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
```

Zadania do wykonania

:one: Wypróbuj kod z *listingów* znajdujących się w instrukcji i sprawdź ich działanie.

:two: Napisz prostą klasę **Student** która będzie posiadać atrybuty takie jak imię i numer albumu. Utwórz 3 różne obiekty tej klasy.

:three: Napisz klasę **Osoba** która będzie posiadać atrybuty takie jak imię i nazwisko. Edytuj klasę **Student** by dziedziczyła po klasie osoba i dodatkowo posiadała atrybut numer_albumu. Utwórz 3 różne obiekty tej klasy a następnie wypisz informacje o nich.

:four: Do klasy **Student** dopisz metodę po której wywołaniu student się przedstawi. ***Cześć nazywam się `imię` `nazwisko` i mój numer albumu to `nr_albumu`***

:five: Napisz klasę **Liczba** mającą jeden atrybut przechowujący liczbę i nadpisz wybraną z magicznych metod tak aby realizowała działanie

$x^2+2xy+y$

:six: Do poniższej klasy dopisz metodę statyczną która wypisze liczbę obiektów (piesków) znajdujących się w liście i wypisze imiona wszystkich obiektów. Przetestuj kod na co najmniej dwóch obiektach.

```
class Dog:
    count = 0  # this is a class variable
    dogs = []  # this is a class variable

    def __init__(self, name):
        self.name = name  # self.name is an instance variable
        Dog.count += 1
        Dog.dogs.append(name)

    def bark(self, n):  # this is an instance method
        print("{} says: {}".format(self.name, "woof! " * n))

...
```

:seven: Napisz metodę która ustawi właściwość *pineapple_allowed* na wartość odpowiednią wartośćktóra będzie podana jako parametr do tej metody. Zadanie dotyczy fragmentu kodu z opisu właściwości. (Dla chętnych)

:exclamation: zadania 2-7 mają zostać dodane na GitHuba :exclamation: