Класи і об’єкти

Об'єктно-орієнтоване програмування (ООП) – це важлива, потужна та виразна парадигма програмування. Програміст мислить з точки зору класів та об’єктів. Клас - це план об’єкта. Об'єкт містить певні дані та забезпечує функціональні можливості, зазначені в класі.

Скажімо, ви програмуєте гру для побудови, моделювання та розвитку міст. В об'єктно-орієнтованому програмуванні ви б представляли всі речі (будівлі, людей або автомобілі) як об'єкти. Наприклад, кожен будівельний об’єкт зберігає такі дані, як назва, розмір та ціна. Крім того, кожна будівля забезпечує певну функціональність, таку як calculate_monthly_earnings (розрахунок місячних доходів). Це спрощує читання та розуміння вашого коду для інших програмістів. Що ще важливіше, тепер ви можете легко розподілити обов’язки між програмістами. Наприклад, ви кодуєте будівлі, а ваш колега кодує автомобілі, що рухаються.

Коротше кажучи, об’єктно-орієнтоване програмування допомагає писати читабельний код. Вивчаючи його, ваша здатність співпрацювати з іншими над складними проблемами покращується.

Classes. Клас інкапсулює дані і функціональність: дані як атрибути і функціональність як методи. Це план для створення конкретних екземплярів (instances) в пам’яті.

Objects. Об’єкт є конкретним втіленням (екземпляром) класу. Кожен об’єкт має власні атрибути, незалежні від інших об’єктів.

Як ми вже згадували раніше, все в Python, від чисел до функцій, є об’єктом. Однак Python приховує більшість деталей об'єктів за допомогою спеціального синтаксису. Ви можете ввести num = 7, щоб створити об'єкт типу integer зі значенням 7 і призначити посилання на об'єкт імені num. Єдиний раз, коли вам потрібно зазирнути всередину об’єктів, це коли ви хочете створити власні або змінити поведінку існуючих об’єктів. У цьому розділі ви побачите, як це зробити.

Що таке об’єкти?

Об'єкт - це власна структура даних, що містить як дані (змінні, які називаються атрибутами), так і код (функції, що називаються методами). Він являє собою унікальний приклад якоїсь конкретної речі. Подумайте про предмети як про іменники, а про їхні методи - як про дієслова. Об'єкт являє собою окрему річ, а його методи визначають, як він взаємодіє з іншими речами.

Наприклад, цілочисельний об’єкт зі значенням 7 є об’єктом, який має такі методи, як додавання та множення. Цілочисельний об’єкт зі значенням 8 - це інший об’єкт. Це означає, що в Python є вбудований клас integer, до якого належать і 7, і 8. Рядки "cat" та "duck" також є об'єктами в Python і мають рядкові методи, такі як capitalize() та replace().

Визначення класу за допомогою ключового слова class

Щоб створити новий об'єкт, який ніхто ніколи раніше не створював, спочатку визначте клас, який вказує, що він містить.

Об’єкт можна порівняти із пластиковою коробкою. Клас - це як форма, яка робить цю коробку. Наприклад, у Python є вбудований клас, який створює рядкові об'єкти, такі як "cat" та "duck", та інші стандартні типи даних - списки, словники тощо. Щоб створити власний користувацький об’єкт у Python, спочатку потрібно визначити клас за допомогою ключового слова class. Давайте розглянемо кілька простих прикладів.

Припустимо, що ви хочете визначити об’єкти для представлення інформації про кішок. Кожен об'єкт буде представляти одну тварину. Спочатку вам потрібно визначити клас під назвою Cat як форму. У наведених нижче прикладах ми пробуємо декілька версій цього класу, будуючи від найпростішого класу до тих, які насправді роблять щось корисне.

Наша перша спроба - найпростіший можливий клас, порожній:

```
>>> class Cat:
... pass
```

Тепер створимо об’єкт з класу, викликаючи ім’я класу так, ніби це функція:

```
>>> a_cat = Cat()
>>> another_cat = Cat()
```

У цьому випадку виклик Cat () створює два окремих об’єкта з класу Cat, і ми призначаємо їх іменам a_cat та another_cat. Але наш клас Cat не мав іншого коду, тому об’єкти, які ми створили з нього, просто сидять там і більше нічого робити не можуть.

Атрибути

Атрибут - це змінна всередині класу або об'єкта. Під час та після створення об’єкта чи класу ви можете призначити йому атрибути. Атрибутом може бути будь-який інший об’єкт. Давайте знову зробимо два котячих об’єкта:

```
>>> class Cat:
...     pass
...
>>> a_cat = Cat()
>>> a_cat
<__main__.Cat object at 0x100cd1da0>
>>> another_cat = Cat()
>>> another_cat
<__main__.Cat object at 0x100cd1e48>
```

Коли ми визначали клас Cat, ми не вказували, як друкувати об’єкт із цього класу. Python друкує щось на зразок `<__ main__.Cat object 0x100cd1da0>`. Далі ми розглянемо, як змінити цю поведінку за замовчуванням.

Тепер призначимо нашому першому об’єкту кілька атрибутів:

```
>>> a_cat.age = 3
>>> a_cat.name = "Mr. Fuzzybuttons"
>>> a_cat.nemesis = another_cat
```

Чи можемо ми отримати до них доступ? Ми на це сподіваємось:

```
>>> a_cat.age
3
```

```
>>> a_cat.name
'Mr. Fuzzybuttons'
>>> a_cat.nemesis
<__main__.Cat object at 0x100cd1e48>
```

Оскільки nemesis був атрибутом, що посилається на інший об’єкт Cat, ми можемо використовувати a_cat.nemesis для доступу до нього, але цей інший об’єкт ще не має атрибута name:

```
>>> a_cat.nemesis.name
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Cat' object has no attribute 'name'
>>> a_cat.nemesis.name = "Mr. Bigglesworth"
>>> a_cat.nemesis.name
'Mr. Bigglesworth'
```

Навіть найпростіший об’єкт, такий як цей, можна використовувати для зберігання кількох атрибутів. Отже, ви можете використовувати кілька об’єктів для зберігання різних значень, замість того, щоб використовувати щось на зразок списку чи словника.

Коли ми еоворимо атрибути, це зазвичай означає атрибути об’єкта. Існують також атрибути класів, і ми побачимо відмінності трохи пізніше.

Методи

Метод - це функція в класі або об’єкті. Метод виглядає як будь-яка інша функція, але його можна використовувати особливими способами, які ви побачите пізніше.

Ініціалізація

Якщо ви хочете призначити атрибути об’єктів під час створення, вам потрібен спеціальний метод ініціалізації об’єктів Python __init __ ():

```
>>> class Cat:
...     def __init__(self):
...         pass
```

Це те, що ви побачите у реальних визначеннях класу Python. Можна визнати, що __init __ () і self виглядають дивно. __init __ () - це спеціальне ім'я Python для методу, який ініціалізує окремий об'єкт з його визначення класу. Аргумент self вказує, що він відноситься до окремого об'єкта.

Коли ви визначаєте __init __ () у визначенні класу, його перший параметр має бути self.

Але навіть це визначення другого класу Cat не створило об’єкт, який справді щось зробив. Третя спроба дійсно показує, як створити простий об’єкт у Python та призначити один із його атрибутів. Цього разу ми додаємо назву параметра до методу ініціалізації:

```
>>> class Cat():
... def __init__(self, name):
... self.name = name
...
>>>
```

Тепер ми можемо створити об'єкт з класу Cat, передавши рядок для параметра name:

```
>>> furball = Cat('Grumpy')
```

Ось що робить цей рядок коду:

• Виявляє визначення класу Cat.

• Визначає (створює) новий об’єкт у пам’яті.

• Викликає метод __init __ () об'єкта, передаючи цей новостворений об'єкт як self, а інший аргумент ('Grumpy') як ім'я.

• Зберігає значення імені в об’єкті.

• Повертає новий об’єкт.

• Прикріплює змінну `furball` до об’єкта.

Цей новий об’єкт схожий на будь-який інший об’єкт у Python. Ви можете використовувати його як елемент списку, кортежу, словника або набору. Ви можете передати його функції як аргумент або повернути як результат.

Як щодо значення імені, яке ми передали? Воно було збережене з об'єктом як атрибут. Ви можете читати та писати його безпосередньо:

```
>>> print('Our latest addition: ', furball.name)
Our latest addition: Grumpy
```

Не обов'язково мати метод __init __ () у кожному визначенні класу; використовується для того, щоб зробити все, що потрібно, щоб відрізнити цей об’єкт від інших, створених з того ж класу. Це не те, що деякі інші мови називають "конструктором". Python вже створив об’єкт для вас. Подумайте про __init __ () як про ініціалізатор.

ПРИМІТКА

З одного класу можна створити багато окремих об’єктів. Але пам’ятайте, що Python реалізує дані як об’єкти, тому сам клас є об’єктом. Однак у вашій програмі є лише один об’єкт класу.

Успадкування

Коли ви намагаєтесь вирішити якусь проблему кодування, ви часто знаходите існуючий клас, який створює об’єкти, які роблять майже те, що вам потрібно. Що ми можемо зробити?

Одне з рішень - успадкування: створення нового класу з існуючого класу, але з деякими доповненнями або змінами. Це хороший спосіб повторного використання коду. Коли ви використовуєте успадкування, новий клас може автоматично використовувати весь код зі старого класу, але без необхідності копіювати його.

Успадкувати від батьківського класу

Ви визначаєте лише те, що вам потрібно додати або змінити у новому класі, і це замінює поведінку старого класу. Початковий клас називається батьківським (parent), надкласом (superclass) або базовим (base) класом; новий клас називається дочірнім (child), підкласом (subclass) або похідним (derived) класом. Ці терміни взаємозамінні в об'єктно-орієнтованому програмуванні.

Отже, давайте щось успадкуємо. У наступному прикладі ми визначаємо порожній клас під назвою Car. Далі ми визначаємо підклас автомобіля під назвою Yugo. Ви визначаєте підклас, використовуючи те саме ключове слово класу, але з ім’ям батьківського класу всередині дужок (тут клас Yugo (Car)):

```
>>> class Car():
...     pass
...
>>> class Yugo(Car):
...     pass
...
```

Ви можете перевірити, чи походить клас з іншого класу, використовуючи issubclass ():

```
>>> issubclass(Yugo, Car)
True
```

Далі створіть об’єкт з кожного класу:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Дочірній клас - це спеціалізація батьківського класу; в об'єктно-орієнтованому мовленні, Yugo - це автомобіль (is-a Car). Об'єкт з назвою give_me_a_yugo є екземпляром класу Yugo, але він також успадковує все, що може зробити Car. У цьому випадку Car і Yugo не дуже корисні, тому давайте спробуємо нові визначення класів, які насправді щось роблять:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     pass
...
```

Нарешті, створіть по одному об’єкту з кожного класу та викличте метод exclaim:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Car!
```

Не роблячи нічого особливого, Yugo успадкував метод exclaim () від Car. Насправді, Yugo каже, що це Car, і це може призвести до кризи ідентичності. Давайте подивимось, що ми можемо з цим зробити.

ПРИМІТКА

Спадковість приваблива, але її можна надмірно використовувати. Багаторічний досвід програмування показав, що надмірне використання спадкування може ускладнити управління програмами. Натомість часто рекомендується використовувати інші прийоми, такі як агрегація та композиція. Ми дійдемо до цих альтернатив у цьому розділі.

Перевизначити (Override, Перекрити) метод

Як ви тільки що бачили, новий клас спочатку успадковує все від свого батьківського класу. Продовжуючи, ви побачите, як замінити або перевизначити батьківський метод. Напевно, Yugo в чомусь має відрізнятися від Car; інакше,

який сенс визначати новий клас? Давайте змінимо, як метод exclaim() працює для Yugo:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car")
...
```

Тепер зробіть два об’єкти з цих класів:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Що вони кажуть?

```
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Yugo! Much like a Car
```

У цих прикладах ми замінили метод exclaim (). Ми можемо змінити будь-які методи, включаючи __init __ (). Ось ще один приклад, який використовує клас Person. Давайте зробимо підкласи, які представляють лікарів (MDPerson) та юристів (JDPerson):

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
>>> class MDPerson(Person):
...     def __init__(self, name):
...         self.name = "Doctor " + name
...
>>> class JDPerson(Person):
...     def __init__(self, name):
...         self.name = name + ", Esquire"
...
```

У цих випадках метод ініціалізації __init __ () бере ті ж аргументи, що і батьківський клас Person, але зберігає значення імені по-різному в екземплярі об'єкта:

```
>>> person = Person('Fudd')
>>> doctor = MDPerson('Fudd')
>>> lawyer = JDPerson('Fudd')
>>> print(person.name)
Fudd
>>> print(doctor.name)
Doctor Fudd
>>> print(lawyer.name)
Fudd, Esquire
```

Додавання метода

Дочірній клас також може додати метод, якого не було в його батьківському класі. Повертаючись до класів Car і Yugo, ми визначимо новий метод need_a_push () лише для класу Yugo:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car.")
...     def need_a_push(self):
...         print("A little help here?")
...
```

Отримання допомоги від базового кдасу за допомогою super()

Ми побачили, як дочірній клас може додати або замінити метод від батьків. Що якби він хотів викликати цей батьківський метод? "Я радий, що ви запитали", - каже super (). Тут ми визначаємо новий клас під назвою EmailPerson, який представляє особу з адресою електронної пошти. По-перше, знайоме нам визначення Person:

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
```

Зверніть увагу, що виклик __init __ () у наступному підкласі має додатковий параметр електронної пошти:

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         super().__init__(name)
...         self.email = email
```

Коли ви визначаєте метод __init __ () для свого класу, ви замінюєте метод __init __ () його батьківського класу, і останній більше не викликається автоматично. Як наслідок, нам потрібно це явно назвати. Ось що відбувається:

• super () отримує визначення батьківського класу, Person.

• Метод __init __ () викликає метод Person .__ init __ (). Він піклується про передачу аргумента self до суперкласу, тому вам просто потрібно надати йому інші аргументи. У нашому випадку єдиний інший аргумент, який приймає Person (), - це ім'я.

• Рядок self.email = email - це новий код, який відрізняє цю EmailPerson від Person.

Продовжуючи, давайте створимо об’єкт EmailPerson:

```
>>> bob = EmailPerson('Bob Frapples', 'bob@frapples.com')
```

Ми маємо доступ до обох атрибутів name та email:

```
>>> bob.name
'Bob Frapples'
>>> bob.email
'bob@frapples.com'
```

Чому ми просто не визначили наш новий клас наступним чином?

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         self.name = name
...         self.email = email
```

Ми могли б це зробити, але це б відкинуло наше використання успадкування. Ми використовували super(), щоб змусити Person виконувати свою роботу, так само як звичайний об'єкт Person. Існує ще одна перевага: якщо визначення Person зміниться в майбутньому, використання super() забезпечить, щоб атрибути та методи, які EmailPerson успадкував від Person, відображали зміни.

Використовуйте super (), коли похідний клас (child) робить щось по-своєму, але все ще потребує чогось від батьків (як у реальному житті).

Множинне успадкування

Ви щойно бачили деякі приклади класів без батьківського класу, а деякі з ним. Насправді об’єкти можуть успадковуватися від кількох батьківських класів.

Якщо ваш клас посилається на метод або атрибут, якого у нього немає, Python буде шукати всіх батьків. Що робити, якщо у кількох з них є щось з такою назвою? Хто виграє?

На відміну від успадкування у людей, де домінантний ген перемагає незалежно від того, від кого він походить, успадкування в Python залежить від *method resolution order* (порядку розрішення методу). Кожен клас Python має спеціальний метод під назвою mro (), який повертає список класів, які слід відвідати, щоб знайти метод або атрибут для об’єкта цього класу. Подібний атрибут, який називається __mro__, є кортежем цих класів. Перемагає перший с в списку.

Давайте визначимо клас Animal, два дочірні класи (Horse та Donkey), а потім два, отримані з них:

```
>>> class Animal:
...     def says(self):
            return 'I speak!'
...
>>> class Horse(Animal):
...     def says(self):
...         return 'Neigh!'
...
>>> class Donkey(Animal):
...     def says(self):
...         return 'Hee-haw!'
...
>>> class Mule(Donkey, Horse):
...     pass
...
>>> class Hinny(Horse, Donkey):
...     pass
...
```

Якщо ми шукаємо метод або атрибут Mule, Python розгляне такі речі в такому порядку:

1. Сам об'єкт (типу Mule)
2. Клас об’єкта (Mule)
3. Перший батьківський клас (Donkey)
4. Другий батьківський клас (Horse)

5. Клас бабусь і дідусів (Animal)

Це приблизно те саме для Hinny, але з Horse перед Donkey:

```
>>> Mule.mro()
[<class '__main__.Mule'>, <class '__main__.Donkey'>,
<class '__main__.Horse'>, <class '__main__.Animal'>,
<class 'object'>]
>>> Hinny.mro()
[<class '__main__.Hinny'>, <class '__main__.Horse'>,
<class '__main__.Donkey'>, <class '__main__.Animal'>,
class 'object'>]
```

То що ж говорять ці прекрасні звірі?

```
>>> mule = Mule()
>>> hinny = Hinny()
>>> mule.says()
'hee-haw'
>>> hinny.says()
'neigh'
```

Ми перерахували батьківські класи у порядку (батько, мати), тому вони розмовляють, як їхні тата.

Якби Horse та Donkey не мали методу says(), mule або hinny скористалися би методом says() класу бабусь і дідусів Animal і повернули б 'I speak!'.

Доступ до атрибутів

У Python атрибути та методи об’єктів зазвичай є загальнодоступними (public), і від вас очікується поводитися згідно політики “consenting adults” («згода дорослих»). Давайте порівняємо прямий підхід з деякими альтернативами.

Прямий доступ

Як ви бачили, ви можете отримувати та встановлювати значення атрибутів безпосередньо:

```
>>> class Duck:
...     def __init__(self, input_name):
...         self.name = input_name
...
>>> fowl = Duck('Daffy')
>>> fowl.name
'Daffy'
```

Але що, якщо хтось поводиться погано?

```
>>> fowl.name = 'Daphne'
>>> fowl.name
'Daphne'
```

Наступні два розділи показують способи отримати певну конфіденційність для атрибутів, які ви не хочете, щоб хтось випадково змінив.

Геттери та сеттери

Деякі об’єктно-орієнтовані мови підтримують приватні атрибути об’єктів, до яких неможливо отримати доступ безпосередньо ззовні. Тоді програмістам може знадобитися написати методи getter та setter для читання та запису значень таких приватних атрибутів.

Python не має приватних атрибутів, але ви можете писати геттери та сеттери з заплутаними іменами атрибутів, щоб отримати трохи конфіденційності. (Найкраще рішення - використовувати властивості, описані в наступному розділі.)

У наступному прикладі ми визначаємо клас Duck з одним атрибутом екземпляра під назвою hidden_name. Ми не хочемо, щоб люди отримували прямий доступ до нього, тому ми визначаємо два методи: getter (get_name ()) та setter (set_name ()). Доступ до кожного здійснюється за допомогою властивості під назвою name. Я додав оператор print () до кожного методу, щоб показати, коли він викликається:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     def get_name(self):
...         print('inside the getter')
...         return self.hidden_name
...     def set_name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
>>> don = Duck('Donald')
>>> don.get_name()
inside the getter
'Donald'
>>> don.set_name('Donna')
inside the setter
>>> don.get_name()
inside the getter
'Donna'
```

Властивості для доступу до атрибутів

Рішенням Pythonic для конфіденційності атрибутів є використання властивостей.

Є два способи зробити це. Перший спосіб - додати name = property (get_name, set_name) як останній рядок нашого попереднього визначення класу Duck:

```
>>> class Duck():
>>>     def __init__(self, input_name):
>>>         self.hidden_name = input_name
>>>     def get_name(self):
>>>         print('inside the getter')
>>>         return self.hidden_name
>>>     def set_name(self, input_name):
>>>         print('inside the setter')
>>>         self.hidden_name = input_name
>>>     name = property(get_name, set_name)
```

Старий геттер і сеттер все ще працюють:

```
>>> don = Duck('Donald')
>>> don.get_name()
inside the getter
'Donald'
```

```
>>> don.set_name('Donna')
inside the setter
>>> don.get_name()
inside the getter
'Donna'
```

Але тепер ви також можете використовувати ім'я властивості, щоб отримати та встановити `hidden_name`:

```
>>> don = Duck('Donald')
>>> don.name
inside the getter
'Donald'
>>> don.name = 'Donna'
inside the setter
>>> don.name
inside the getter
'Donna'
```

У другому методі ви додаєте кілька декораторів і замінюєте імена методів get_name та set_name на ім'я:

• @property, що передує методу getter

• @name.setter, що передує методу setter

Ось як вони виглядають у коді:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.hidden_name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
```

Ви все ще можете отримати доступ до name так, ніби це атрибут:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

Властивості обчислюваних значень

У попередніх прикладах ми використовували властивість name для посилання на єдиний атрибут (hidden_name), що зберігається в об'єкті.

Властивість також може повертати обчислене значення. Давайте визначимо клас Circle, який має атрибут radius та обчислювану властивість diameter:

```
>>> class Circle():
...     def __init__(self, radius):
...         self.radius = radius
```

```
...     @property
...     def diameter(self):
...         return 2 * self.radius
...
```

Створіть об’єкт Circle з початковим значенням для його радіуса:

```
>>> c = Circle(5)
>>> c.radius
5
```

Ми можемо посилатись на diameter так, ніби це атрибут, такий як radius:

```
>>> c.diameter
10
```

Ось найцікавіша частина: ми можемо будь-коли змінити атрибут radius, а властивість diameter буде обчислюватися з поточного значення радіуса:

```
>>> c.radius = 7
>>> c.diameter
14
```

Якщо ви не вказуєте властивість setter для атрибута, ви не можете встановити його ззовні. Це зручно для атрибутів лише для читання:

```
>>> c.diameter = 20
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Існує ще одна перевага використання властивості перед прямим доступом до атрибутів: якщо ви коли-небудь змінюєте визначення атрибута, вам потрібно виправити лише код у визначенні класу, а не у всіх абонентах.

Зміна імен для конфіденційності

У прикладі класу Duck трохи раніше ми називали наш (не повністю) прихований атрибут hidden_name. Python має умову іменування атрибутів, які не повинні бути видимими поза визначенням їх класу: почніть з двох знаків підкреслення (__).

Давайте перейменуємо hidden_name на __name, як показано тут:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.__name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.__name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.__name = input_name
...
```

Перевіримо, чи все ще працює:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
```

```
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

Виглядає добре. І ви не можете отримати доступ до атрибута __name:

```
>>> fowl.__name
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Duck' object has no attribute '__name'
```

Ця угода про іменування не робить її повністю конфіденційною, але Python змінює назву атрибута, щоб на неї не натрапив зовнішній код. Ось що станеться:

```
>>> fowl._Duck__name
'Donald'
```

Атрибути класів та об'єктів

Ви можете призначати атрибути класам, і вони успадковуватимуться їхніми дочірніми об'єктами:

```
>>> class Fruit:
...     color = 'red'
...
>>> blueberry = Fruit()
>>> Fruit.color
'red'
>>> blueberry.color
'red
```

Але якщо ви зміните значення атрибута в дочірньому об'єкті, це не вплине на атрибут класу:

```
>>> blueberry.color = 'blue'
>>> blueberry.color
'blue'
>>> Fruit.color
'red'
```

Якщо ви пізніше змініте атрибут класу, це не вплине на існуючі дочірні об'єкти:

```
>>> Fruit.color = 'orange'
>>> Fruit.color
'orange'
>>> blueberry.color
'blue'
```

Але це вплине на нові:

```
>>> new_fruit = Fruit()
>>> new_fruit.color
'orange'
```

Типи методів

Деякі методи є частиною самого класу, деякі - частиною об'єктів, створених з цього класу, а деякі - жодним із перерахованих вище:

• Якщо попереднього декоратора немає, це метод екземпляра, і його перший аргумент має бути "self" для посилання на сам окремий об'єкт.

• Якщо існує попередній декоратор @classmethod, це метод класу, і його перший аргумент має бути cls (або що завгодно, тільки не зарезервоване слово class), посилаючись на сам клас.

• Якщо існує попередній декоратор @staticmethod, це статичний метод, і його перший аргумент не є об’єктом або класом.

У наступних розділах є деякі деталі.

Методи екземплярів

Коли ви бачите початковий аргумент self у методах у визначенні класу, це метод екземпляра. Це типи методів, які ви зазвичай пишете при створенні власних класів. Першим параметром методу екземпляра є self, і Python передає об'єкт методу під час його виклику. Це ті методи, які ви бачили досі.

Методи класів

Навпаки, метод класу впливає на клас в цілому. Будь-яка зміна класу впливає на всі його об’єкти. У визначенні класу попередній декоратор @classmethod вказує, що наступна функція є методом класу. Крім того, першим параметром методу є сам клас. Традиція Python - називати параметр cls, оскільки class є зарезервованим словом і не може використовуватися тут. Давайте визначимо метод класу для A, який підраховує, скільки екземплярів об’єктів було зроблено з нього:

```
>>> class A():
...     count = 0
...     def __init__(self):
...         A.count += 1
...     def exclaim(self):
...         print("I'm an A!")
...     @classmethod
...     def kids(cls):
...         print("A has", cls.count, "little objects.")
...
>>>
>>> easy_a = A()
>>> breezy_a = A()
>>> wheezy_a = A()
>>> A.kids()
A has 3 little objects.
```

Зверніть увагу, що ми посилалися на A.count (атрибут класу) у __init __ (), а не на self.count (який був би атрибутом екземпляра об'єкта). У методі kids () ми використовували cls.count, але ми могли б так само використовувати A.count.

Статичні методи

Третій тип методу у визначенні класу не впливає ні на клас, ні на його об’єкти; він просто для зручності. Це статичний метод, якому передує декоратор @staticmethod, без початкових параметрів self або cls. Ось приклад, який служить рекламним роликом для класу CoyoteWeapon:

```
>>> class CoyoteWeapon():
...     @staticmethod
...     def commercial():
```

```
...          print('This CoyoteWeapon has been brought to you
by Acme')
...
>>>
>>> CoyoteWeapon.commercial()
This CoyoteWeapon has been brought to you by Acme
```

Зауважте, що нам не потрібно було створювати об’єкт із класу CoyoteWeapon, щоб отримати доступ до цього методу.

Магічні методи

Магічні методи - це спеціальні методи Python назви яких починаються і закінчуються подвійним підкресленням (__). Ви вже бачили такий метод: __init __ () ініціалізує новостворений об’єкт. Є ще багато інших магічних методів.

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...     def __str__(self):
...         return self.text
...     def __repr__(self):
...         return 'Word("' + self.text + '")'
...
>>> first = Word('ha')
>>> first        # uses __repr__
Word("ha")
>>> print(first) # uses __str__
Ha
```

Агрегація та композиція

Спадкування – це хороший прийом, який потрібно використовувати, коли ви хочете, щоб дочірній клас міг діяти як його батьківський клас (коли дитина є (is-a) батьком). Буде спокуса побудувати складні ієрархії спадкування, але іноді композиція або агрегація мають більший сенс. Яка різниця? За композицією одне є частиною іншого. Качка - це птах (спадкування), але має (has-a) хвіст (композиція). Хвіст (tail) - це не вид качки, а частина качки. У цьому наступному прикладі давайте зробимо bill та tail об’єкти та надамо їх новому об’єкту качки:

```
>>> class Bill():
...     def __init__(self, description):
...         self.description = description
...
>>> class Tail():
...     def __init__(self, length):
...         self.length = length
...
>>> class Duck():
...     def __init__(self, bill, tail):
...         self.bill = bill
...         self.tail = tail
...     def about(self):
```

```
...         print('This duck has a', self.bill.description,
...             'bill and a', self.tail.length, 'tail')
...
>>> a_tail = Tail('long')
>>> a_bill = Bill('wide orange')
>>> duck = Duck(a_bill, a_tail)
>>> duck.about()
This duck has a wide orange bill and a long tail
```

Агрегування виражає відносини, але трохи вільніше: одне використовує (uses) інше, але обидва існують незалежно. Качка використовує озеро, але одне не є частиною іншого.

3.3 Приклади розв'язування задач

Приклад 1. Обчислити …

3.4 Питання для самоконтролю

1. Як створити клас у мові Python?
2. Що таке інкапсуляція?
3. Що таке об'єкт?
4. Що таке атрибут?
5. Для чого призначений консруктор класу?
6. Який обов'язковий аргумент приймає метод класу?
7. Що таке self?
8. Як оголосити метод класу статичним?

ЛІТЕРАТУРА

1. Яковенко А.В. Основи програмування. Python. Частина 1: підручник для студ. спеціальності 122 "Комп'ютерні науки". – Київ : КПІ ім. Ігоря Сікорського, 2018. – 195 с.
2. Програмування на мові Python. Методичні вказівки до виконання лабораторних робіт студентами денної та заочної форми навчання спеціальностей 123 "Комп'ютерна інженерія", 125 "Кібербезпека" / Укл.: Є.В. Мелешко – Кропивницький: ЦНТУ, 2017. – 58 с.
3. Програмування числових методів мовою Python : підруч. / А. В. Анісімов, А. Ю. Дорошенко, С. Д. Погорілий, Я. Ю. Дорогий ; за ред. А. В. Анісі-

мова. – К. : Видавничо-поліграфічний центр "Київський університет", 2014. – 640 с.
4. Костюченко А.О. Основи програмування мовою Python: навчальний посібник. – Чернігів: ФОП Баликіна С.М., 2020. 180 с.
5. Креневич А.П. Python у прикладах і задачах. Частина 1. Структурне програмування. Навчальний посібник із дисципліни "Інформатика та програмування" – К.: ВПЦ "Київський Університет", 2017. – 206 с.
6. Лутц М. Изучаем Python, том 1, 5-е изд. — СПб.: ООО “Диалектика”, 2019. — 832 с.
7. Лутц М. Изучаем Python, том 2, 5-е изд. — СПб.: ООО “Диалектика”, 2020. — 720 с.
8. Joakim Sundnes Introduction to Scientific Programming with Python. – Springer, 2020. 148 p.
9. Bill Lubanovic Introducing Python. – O’Reilly Media, 2020. 597 p.