ПЗ мікропроцесорних систем (Програмування штучного інтелекту на Python)

Тема 7. Лекція 7
Основні принципи ООП в Python. Програмування класів в Python

Зміст

- Об'єктно-орієнтоване програмування.
- Поняття ООП.
- Створення класів і об'єктів.
- Конструктори і деструктори.
- Типи методів класу.
- Магічні методи.
- Життевий цикл об'єкта.
- Представлення класів.
- Оператори порівняння.
- Контейнери.
- Замикання (closures).
- Синтаксис декораторів.

Об'єктно-орієнтоване програмування

- Цикли, розгалуження і функції — все це елементи процедурного програмування. Його можливостей цілком достатньо для написання невеликих, простих програм. Однак великі проекти часто реалізують, використовуючи парадигму об'єктно-орієнтованого програмування (ООП).
- Слід зауважити, не усі сучасні мови підтримують ООП. Але в Python ООП грає ключову роль. Навіть програмуючи в рамках структурної парадигми, ви усе одно користуєтесь об'єктами і класами, нехай навіть вбудованими в мову, а не створенними особисто вами.
- Отже, що ж таке об'єктно-орієнтоване програмування? Судячи з назви, ключову роль тут відіграють якісь об'єкти, на які орієнтується весь процес програмування.

Об'єктно-орієнтоване програмування

- Якщо подивитись на реальний світ під тим кутом, під яким звикли на нього дивитись, то для нас він предстане у вигляді багатьох об'єктів, яким притаманні певні властивості, які взаємодіють між собою и внаслідок цього змінюються. Ця звична для погляду людини картина світу була перенесена у програмування. Вона вимагала більш високого рівня абстракції від того, як обчислювальна машина зберігає і обробляє дані, вимагала від програмістів вміння конструювати свого роду "віртуалы світи", розподілювати між собою задачі. Однак дала можливість більш легкої і продуктивної розробки великих програм.
- Припустимо, команда програмістів займається розробкою гри. Програму-гру можна представити як систему, яка складається з цифрових героїв і середовища їх існування, яке включає багато предметів. Кожен воїн, зброя, дерево, будинок — це цифровий об'єкт, в якому "упаковано" його властивості і дії, за допомогою яких він може змінювати свої властивості і властивості інших об'єктів.

Об'єктно-орієнтоване програмування

- Кожен програміст може розробляти свою групу об'єктів. Розробникам достатньо домовитись між собою тільки про те, як об'єкти будуть взаємодіяти між собою.
- Ключову різницю між програмою, написаною в структурному стилі, і об'єктно-орієнтованою програмою можна висловити так: у першому випадку на перший план виходить логіка, розуміння послідовності виконання дій для досягнення поставленої цілі. У другому — важливіше представити програму як систему об'єктів, які взаємодіють.

Поняття ООП

- Основними поняттями в ООП є клас, об'єкт, успадкування, інкапсуляція і поліморфізм. В Python клас рівносильний поняттю тип даних.
- Що таке клас? Проведемо аналогію з реальним світом. Якщо ми візьмем конкретний стіл, то це об'єкт, але не клас. А ось загальне уявлення про столи, їх призначення — це клас. Йому належать усі реальні об'єкти столів, якими б вони не були. Клас столів дає загальну характеристику усім столам в світі, він їх узагальнює.
- Те ж саме з цілими числами в Python. Тип int — це клас цілих чисел. Числа 5, 100134, -10 і т. п. — це конкретні об'єкти цього класа.
- В Python об'єкти також прийнято називати екземплярами. Це пов'язано з тим, що в ньому усі класи самі є об'єктами класа type.

Поняття ООП

- Наступне по важливості поняття ООП — успадкування. Повернемось до столів. Нехай є клас столів, який описує загальні властивості усіх столів. Однак можна розділити усі столи на письмові, обідні і журнальні і для кожної групи створити свій клас, який буде спадкоємцем загального класа, але також вносити ряд своїх особливостей. Таким чином, загальний клас будет батьківським, а класи груп — дочірніми.
- Дочірні класи успадковують особливості батьківських, однак доповнюють або у деякій мірі модифікують їх характеристики. Коли ми створюємо конкретний екземпляр стола, то повинні обрати, до якого класу столів він буде належати. Якщо він належить класу журнальних столів, то отримає усі характеристики загального класа столів і класа журнальних столів. Але не особливості письмових і обідніх.

Поняття ООП

- Інкапсуляція в ООП розуміють двояко. У багатьох мовах цей термін означає приховування даних, тобто неможливість напряму отримати доступ до внутрішньої структури об'єкта, так як це небезпечно. В Python немає такої інкапсуляції, хоча вона є одним з стандартів ООП. В Python можна отримати доступ до будь-якої властивості об'єкта і змінити його. Однак є механізм, який дозволяє імітувати приховування даних, якщо це так необхідно.
- Інший смисл інкапсуляції — об'єднання властивостей і поведінки в одне ціле, тобто в клас.

Поняття ООП

- Поліморфізм — це множина форм. Однак в поняттях ООП мається на увазі скоріш зворотнє. Об'єкти різних класів, з різною внутрішньою реалізацією можуть мати однакові інтерфейси. Наприклад, для чисел є операція додавання, яка позначається знаком +. Однак ми можемо визначити клас, об'єкти котрого також будуть підтримувати операцію, яка позначається цим знаком. Але це зовсім не означає, що об'єкти повинні бути числами, і буде отримуватись якась сума. Операція + для об'єктів нашого класа може означати щось інше. Але інтерфейс, в даному випадку це знак +, у чисел і нашего класа буде однаковим. Поліморфність же проявляється у внутрішній реалізації і результаті операції.
- Ми вже стикались з поліморфізмом операції +. Для чисел вона означає додавання, а для рядків — конкатенацію. Внутрішня реалізація кода для цієї операції у чисел відрізняється від реалізації такої для рядків.

Створення класів і об'єктів

- В мові програмування Python класи створюють за допомогою команди class, після якої вказують им'я класа, потім ставиться двокрапка, далі з нового рядка і з відступом реалізується тіло класа:

```
class NameOfClass:
    pass
```

- Якщо клас є дочірнім, то батьківські класи перераховуються у круглих дужках після імені класа:

```
class MyClass(ParentClass):
    pass
```

Створення класів і об'єктів

- Об'єкт створюється шляхом виклику класа за його іменем. При цьому після імені класа обов'язково ставляться дужки. Оскільки у програмному коді важливо не згубити посилання на щойно створений об'єкт, то зазвичай його пов'язують зі змінною. Отже створення об'єкта найчастіше виглядає так:

```
class MyClass(ParentClass):
    pass

obj = MyClass()
obj2 = MyClass()
```

Клас як модуль

- В Python клас можна представити подібно модулю. Так само як у модулі у ньому можуть бути свої змінні зі значеннями і функції. Так само як у модулі у класа є власний простір імен, доступ до якого можливий через ім'я клас:
- >>> class Adder:
- ... n = 5
- ... def add(v):
- ... return v + Adder.n
- ...
- >>> Adder.n
- 5
- >>> Adder.add(4)
- 9
- >>>

Однак у випадку класів використовується дещо інша термінологія. Імена, визначені в класі, називаються атрибутами цього класа. В вищенаведеному прикладі імена n і add — це атрибути клас Adder. Атрибути-змінні часто називають полями і деколи властивостями. Атрибути-функції називаються методами. Кількість полів і методів у класі може бути довільною.

Клас як створювач об'єктів

```
>>> a = Adder()
>>> a.n
5
>>> a.add(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() takes 1 positional argument but 2 were given
>>>
```

Інтерпретатор повідомляє нам, що add() приймає тільки один аргумент, а було передано два. Звідки ж взявся другий аргумент, і хто він такий, якщо у дужках було вказано тільки одне число 4?

Клас як створювач об'єктів

Клас створює об'єкти, які у певному сенсі є його спадкоємцями. Це означає, що якщо у об'єкта немає власного поля n, то інтерпретатор шукає його на один рівень вище, тобто у класі. Таким чином, якщо ми присвоюємо об'єкту поле с таким самим ім'ям як у класі, то воно перекриває, або перевизначає, поле клас:

```
>>> a.n = 10
>>> a.n
10
>>> Adder.n
5
>>>
```

Тут змінні a.n і Adder.n — це різні змінні. Перша знаходиться у просторі імен об'єкта, друга — у просторі класа Adder. Якщо б ми не додали поле n до об'єкта a, то інтерпретатор піднявся б вище по дереву наслідування і прийшов би у клас, де би і знайшов це поле.

Клас як створювач об'єктів

Щодо методів, то вони також наслідуються об'єктами клас. У даному випадку у об'єкта a немає свого власного метода add, отже, він шукається в клас Adder. Однак від клас Adder може бути породжено багато об'єктів. Методи ж найчастіше призначаються для обробки об'єктів. Таким чином, коли викликається метод, у нього треба передати конкретний об'єкт, який він буде обробляти.

Зрозуміло, що екземпляр, що передається — це об'єкт, до якого застосовується метод. Вираз a.add() виконується інтерпретатором наступним чином:

1. Шукаю атрибут add() у об'єкта a. Не знайшов.
2. Тоді йду шукати у клас Adder, так як він створив об'єкт a.
3. Тут знайшов метод. Передаю йому об'єкт, до якого цей метод треба застосувати, і аргумент, що вказано у дужках.

Клас як створювач об'єктів

- Іншими словами, вираз

a.add(4)

- перетвоюється у вираз

Adder.add(a, 4)

- Таким чином, інтерпретатор спробував передати у метод add() класа Adder два фргумента — об'єкт a і число 4. Але ми запрограмували метод add() так, що він приймає тільки один параметр. В Python визначення методів не передбачають прийняття об'єкта як зрозуміле за замовчуванням. Об’єкт, що приймається треба вказувати явно.
- За згодою у Python для посилання на об'єкт використовується ім'я self. Ось так повинен виглядати метод add(), якщо ми плануємо викликати його через об'єкти:

```
>>> class Adder:
...     n = 5
...     def add(self, v):
...             return v + self.n
...
>>>
```

Змінна self зв'язується з об'ктом, до якого було застосовано даний метод, і через цю змінну ми отримуємо доступ до атрибутів об'єкта. Коли цей же метод застосовується до іншого об'єкта, то self зв'яжеться вже з саме цим іншим об'єктом, і через цю змінну будуть вилучатись тільки його поля.

Давайте протестємо оновлений метод:

```
>>> class Adder:
...     n = 5
...     def add(self, v):
...             return v + self.n
...
>>> a = Adder()
>>> b = Adder()
>>> a.n = 10
>>> a.add(3)
13
>>> b.add(4)
9
>>>
```

Тут від класа Adder створюється два об'єкта – a та b. Для об'єкта a заводиться власне поле n. Об'єкт b, не має такого поля, отже успадковує n від клас Adder. Переконаємось у цьому:

```
>>> a.n is Adder.n
False
>>> b.n is Adder.n
True
>>>
```

У методі add() вираз self.n – це звернення до поля n, переданого об'єкта, і не важливо, на якому рівні наслідування його буде знайдено.

Зміна полів об'єкта

- В Python об'єкту можна не тільки перевизначати поля і методи, успадковані від класа, але можна додавати нові, яких немає у класі:

```
>>> a.test = 'Hi'
>>> a.test
'Hi'
>>> Adder.test
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Adder' has no attribute 'test'
>>>
```

- Однак у програмуванні так не прийнято, тому що тоді об'єкти одного класа будуть відрізнятись між собою по набору атрибутів. Це затруднить автоматизацію їх обробки, внесе у програму хаос.

Конструктор класа

- В ООП конструктором класа називають метод, який автоматично викликається при створенні об'єктів. Його також можна назвати конструктором об'єктів класа. Ім'я такого метода зазвичай регламентується синтаксисом конкретної мови програмування. В Python роль конструктора виконує метод __init__(). Об'єкт створюється в момент виклику класа по імені, і в цей момент викликається метод init(), якщо його визначено в класі.
- Необхідність конструкторів пов'язана з тим, що часто об'єкти повинні мати власні властивості одразу. Припустимо маємо клас Person, об'єкти котрого обов'язково повинні мати ім'я. Якщо клас буде описано наступним способом:

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

то створення об'єкта можливе без полів. Для встановлення імені метод set_name() неообхідно викликати окремо:

```
>>> p1 = Person()
>>> p1.name = 'Jane Doe'
>>> p1.name
'Jane Doe'
>>>
```

Наявність конструктора не дозволить створити об'єкт без полів:

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


p1 = Person('Jane Doe')
print(p1.name)
```

Тут при виклику класа в круглих дужках передаються значення, котрі будуть присвоєні параметрам метода init(). Перший параметр – self – посилання на сам щойно створений об'єкт.

Тепер, якщо ми спробуємо створити об'єкт, не передавши нічого в конструктор, то буде "викинуто" вийняткову ситуацію, і об'єкт не буде створено:

```
>>> p2 = Person()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 1 required positional argument:
'name'
>>>
```

Однак буває, що необхідно допустити створення об'єкта, якщо деякі дані в конструктор не передаються. У такому випадку параметрам конструктора класа задаємо значення за замовчуванням:

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

Якщо викликати конструктор без параметра phone, то буде використано значення за замовчуванням. Однак поля name і phone будуть у всіх об'єктів:

```
>>> p1 = Person('Jane Doe', '+380971234567')
>>> p2 = Person('John Doe')
>>> p1.name, p1.phone
('Jane Doe', '+380971234567')
>>> p2.name, p2.phone
('John Doe', '')
>>>
```

Крім того, конструктору зовсім не обов'язково приймати будь-які параметри, крім self. Значення полям можуть назначатись як завгодно. Також не обов'язково, щоб в конструкторі виконувалось встановлення атрибутів об'єкта. Там може бути, наприклад, код, який створює об'єкти інших класів.

Деструктор класа

- Окрім конструктора об'єктів в ООП є зворотній йому метод – деструктор. Він викликається, коли об'єкт знищується.
- В Python об'єкт знищується, коли зникають усі пов'язані з ним змінні або їм присвоюється інше значення, в результаті чого зв'язок з старим об'єктом втрачається. Видалити змінну також можна за допомогою del.
- В Python функцію деструктора виконує метод __del__(). Але в Python деструктор використовується рідко, інтерпретатор і без нього добре впорається зі "сміттям".

Python пропонує три види методів: статичні, класові і екземпляра класа.

- **Методи екземпляра класа**

Це вже знайомий нам вид методів. Методи екземпляра класа приймають об'єкт класа як перший аргумент, який прийнято називати self і який вказує на сам екземпляр.

Використовуючи параметр self ми можемо міняти стан об'єкта і звертатись до інших його методів і параметрів. Також через атрибут self.__class__ можна отримати доступ до атрибутів класа і можливість міняти стан самого класа. Тобто методи екземплярів класа дозволяють міняти як стан певного об'єкта, так і класа.

Приклад — str.upper():

```
>>> 'abc'.upper()
'ABC'
>>>
```

Класові методи

- Методи класа приймають клас в якості параметра, його прийнято позначати як cls. Він вказує на клас, а не на об'єкт цього класа. При декларації методів цього вида використовують декоратор classmethod.
- Методи класа прив'язані до самого класа, а не його екземпляра. Вони можуть міняти стан класа, що відобразиться на усіх об'єктах цього класа, але не можуть міняти конкретний об`єкт.
- Приклад метода класа — dict.fromkeys() — повертає новий словник з переданими елементами в якості ключів:

```
>>> dict.fromkeys('abc')
{'a': None, 'b': None, 'c': None}
>>>
```

Статичні методи¶

- Статичні методи декларуються за допомогою декоратора staticmethod. Їм не потрібно певного першого аргумента (ні self, ні cls).
- Їх можна сприймати як методи, які "не знають, до якого класа відносяться".
- Таким чином, статичні методи прикріплені до класа лише для зручності і не можуть міняти стан ні класа, ні його екземпляра.

Напишемо клас, де використовуються усі три види методів.

```
class ToyClass:
    def instance_method(self):
        return print('instance method called'), self

    @classmethod
    def class_method(cls):
        return print('class method called'), cls

    @staticmethod
    def static_method():
        return print('static method called')

obj = ToyClass()
```

Розберемось з роботою методів. Спочатку метод екземпляра:

```
>>> obj.instance_method()
instance method called
```

- Приклад вище підтверджує те, що метод instance_method має доступ до обєкта класа ToyClass через аргумент self. Зауважте, що метод obj.instance_method() можна викликати і так:

```
>>> ToyClass.instance_method(obj)
instance method called
```

Тепер скористаємось методом класа:

```
>>> obj.class_method()
class method called
```

Як видно, метод класа class_method() має доступ до самого класа ToyClass, але не до його конкретного екземпляра. Пам'ятаєте що в Python усе є об'єктом? Клас — теж об'єкт, який ми можемо передати функції в якості аргумента.

Викликаємо статичний метод:

```
>>> obj.static_method()
static method called
```

- Це може здатися дивним, але статичні методи можна викликати через об'єкт класа. Насправді статичному методу ніякі спеціальні аргументи (self чи cls) не передаються. Тобто статичні методи не можуть отримати доступ до параметрів класа чи об'єкта. Вони працюють тільки з тими даними, які їм передаються як аргументи.

Тепер давайте викличемо ті ж самі методи, але на самому класі.

```
>>> ToyClass.class_method()
class method called
>>> ToyClass.instance_method()
Traceback (most recent call last):
  File "C:/Users/admin/PycharmProjects/Lectures/7.py", line 26, in <module>
    ToyClass.instance_method()
TypeError: instance_method() missing 1 required positional argument: 'self’
```

- Виклик метода екземпляра класа видає TypeError. Сталося це через те, що метод очікував екземпляр класа, а було передано клас.

Зі статичним методом нічого неочікуваного:

```
>>> ToyClass.static_method()
static method called
```

Розглянемо більш практичний приклад для розуміння того, коли і який метод варто використовувати.

```
from datetime import date
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('Initialized Person: %s' % self.name)
    @classmethod
    def from_birth_year(cls, name, year):
        return cls(name, date.today().year - year)
    @staticmethod
    def is_adult(age):
        return age > 18
```

Спробуємо створити об'єкт:

```
p = Person('Jane', 25)
print(p.age)
```

Отримаємо:

```
Initialized Person: Jane
25
```

Тепер викликаємо метод класа, який у свою чергу створить об'єкт цього ж класа і поверне його:

```
p = Person.from_birth_year('John', 1989)
print(p.age)
```

Отримаємо:

```
Initialized Person: John
32
```

І, нарешті, статичний метод класа Person:

```
print(Person.is_adult(25))
```

Отримаємо:

```
True
```

Коли використовувати кожен з методів?

- Вибір того, який з методів використовувати, може здатись доволі складним. З набутим досвідом цей вибір зробити буде простіше.
- Найчастіше метод класа використовується тоді, коли потрібен генеруючий метод, який повертає об'єкт класа. Статичні методи в основному використовуються як допоміжні функції і працюють з даними, які їм передаються.

Отже,

- Методи екземпляра класа отримують доступ до об’єкта класа через параметр self і до класа через self.__class__
- Методи класа не можуть отримати доступ до певного екземпляра класа, але мають доступ до самого класа через cls
- Статичні методи працюють як звичайні функції, але належать до простору імен класа. Вони не мають доступа ні до самого класа, ні до його екземплярів

"Магічні" методи. Перевизначення операторів

- Спеціальний, або магічний метод — це метод, який викликається неявно, часто щоб виконати операцію для певного типа.
- Наприклад, можна визначити bool метод, щоб вказати чи є об'єкт True/False в булевому контексті. Таким чином дуже багато аспектів поведінки об'єктів в Python можна змінити.
- Імена магічних методів починаються і закінчуються двома символами підкреслення, наприклад:

__init__

__getattr__

__eq__

Життєвий цикл об'єкта

- Нам вже відомий самий базовий магічний метод, __init__. За його допомоги ми можемо ініціалізувати об'єкт. Однак, коли ми напишемо:

```
x = SomeClass()
```

- 'init' це не саме перше, що викликається. Насправді екземпляр об'єкта створює метод __new__, а вже потім ініціалізатор. На іншому кінці життєвого цикла об'єкта знаходиться метод __del__. Давайте детальніше розглянемо ці три магічних метода.

__new__(cls, [...])¶

Життєвий цикл об'єкта

- Конструктор класа. Це перший метод, який буде викликано при створенні об'єкта. Він приймає в якості параметрів клас і потім будь-які інші аргументи, які буде передано в __init__. __new__ використовується дуже рідко, але деколи буває корисним, зокрема, коли клас успадковується від немутабельного (immutable) типа, такого як кортеж (tuple) чи рядок.

__init__(self, [...)¶

- Ініціалізатор класа. Йому передається усе, з чим було викликано конструктор. Наприклад, якщо ми напишемо

```
x = SomeClass(10, 'foo')
```

Життєвий цикл об'єкта

- __init__ отримає 10 і 'foo' в якості аргументів. __init__ майже повсемістно використовується при визначенні класів. Майже завжди цей метод помилково називають конструктором.

__del__(self)¶

- Якщо __new__ і __init__ — конструктори об'єкта, __del__ — його деструктор. Він не визначає поведінку для оператора del, тому цей код не є еквівалентним x.__del__(). Він визначає поведінку об'єкта у той час, коли за об'єкт береться збиральник сміття. Це може бути доволі зручно для об'єктів, які можуть потребувати додаткових "чисток" під час видалення, наприклад сокети чи файлові об'єкти. Однак пам'ятайте, що del не може слугувати заміною для хороших програмістських практик. Завжди завершуйте з'єднання, якщо закінчили з ними працювати і так далі! Фактично, через відсутність гарантії виклику, del не повинен використовуватись майже ніколи. Використовуйте його з обережністю!

Представлення класів

- Часто буває корисним представлення класа у вигляді символьного рядка. В Python існує декілька методів, які ви можете визначити для налаштування поведінки вбудованих функцій при представленні вашого класа.

__str__(self)¶

- Визначає поведінку функції str(), яка була викликана для екземпляра вашого класа.

__repr__(self)¶

- Визначає поведінку функції 'repr()', викликаної для екземпляра вашого класа. Головна відмінність від str() — в цільовій аудиторії. repr() більше призначено для машино-орієнтованого вивода, більш того, це по можливості має бути валідний код на Python) для створення екземпляра класа. str() призначено для читання людьми.

Магічні методи порівняння

- В Python є багато магічних методів, створених для визначення інтуітивного порівняння між об'єктами використовуючи оператори. Крім того, вони надають спосіб перевизначити поведінку Python за замовчуванням для порівняння об'єктів (по посиланню). Ось список цих методів і що вони роблять:
- __eq__(self, other)
- Визначає поведінку оператора рівності, ==.
- __ne__(self, other)
- Визначає поведінку оператора нерівності, !=.
- __lt__(self, other)
- Визначає поведінку оператора "меньше ніж", <
- __gt__(self, other)
- Визначає поведінку оператора "більше ніж", >
- __le__(self, other)
- Визначає поведінку оператора "меньше або дорівнює", <=
- __ge__(self, other)
- Визначає поведінку оператора "більше або дорівнює", >=

Контейнери

- В Python існує багато способів заставити ваші класи поводитись як вбудовані контейнери (словники, кортежі, списки, рядки інше). Для цього клас має реалізувати відповідний протокол.
- Протокол (в контексті) — набір методів, який має реалізовувати клас щоб відповідати певним сутностям.
- Реалізація довільних контейнерних типів в Python призводить до використання деяких з них. Наприклад, протокол для визначення незмінних контейнерів: щоб створити незмінний контейнер, має бути визначено __len__ і __getitem__. Протокол змінного контейнера потребує того ж, плюс __setitem__ і __delitem__. І, якщо ви бажаєте, щоб ваші об'єкти можна було перебирати ітераціями, ви маєте визначити __iter__, який повертає ітератор.

Контейнери

- Ось магічні методи, які використовують контейнери:

__getitem__(self, key)¶

- Визначає поведінку при доступі до елемента використовуючи наступний синтаксис:

self[key]

- Стосується протоколу змінних і протоколу незмінних контейнерів.

- Для послідовностей key має бути цілим або зрізанням. При цьому в методі може бути реалізована інтерпретація від'ємних індексів.

Контейнери

Метод має піднімати відповідний виняток:

- TypeError — коли тип ключа не підтримується.
- KeyError — коли ключу не відповідає жодне значення у відображеннях.
- IndexError — якщо ключ виходить за межі ключей послідовності (у тому числі після інтерпретації від'ємних значень).

- Цикл for in для правильного визначення кінця послідовності очікує, що для недопустимих індексів піднімається виняток IndexError.

__len__(self)¶

- Повертає кількість елементів у контейнері. Частина протоколів для змінного та незмінного контейнерів.

__setitem__(self, key, value)¶

Визначає поведінку при присвоєнні значення елементу при використанні синтаксису:

self[key] = value

Частина протокола змінного контейнера. Має піднімати KeyError і TypeError у відповідних випадках.

__delitem__(self, key)¶

Визначає поведінку при видаленні елемента. Частина протокола для змінних контейнерів. Має піднімати відповідний виняток, якщо ключ некоректний.

__iter__(self)¶

Повертає ітератор для контейнера.

__reversed__(self)¶

Визначає поведінку для вбудованої функції reversed(). Має повернути "зворотню" версію послідовності. Реалізується для впорядкованих контейнерів.

__contains__(self, item)¶

Реалізує перевірку належності елемента при використанні синтаксиса:

in/not in

Метод не є частиною протокола послідовності, оскільки Python може самостійно виконувати пошук елемента "перебираючи" всю послідовність.

Замикання (closures)

- Кожного разу, коли ми викликаємо функцію, у неї створюються локальні змінні (якщо, звичайно, вони у неї є), а після завершення — знищуються, при черговому виклику ця процедура повторюється. А чи можна зробити так, щоб після завершення роботи функції, частина локальних змінних не знищувалась, а зберігала б свої значення до наступного запуска? Так, це можна зробити!
- Локальна змінна не буде знищена, якщо на неї десь залишиться “живе” посилання, після завершення роботи функції. Це посилання може зберігати вкладена функція. Функції, які побудовані по такому принципу можуть використовуватись для побудови спеціалізованих функцій, тобто є як би фабриками функцій. Далі розглянемо створення і використання так званих "замикань", які якраз і використовують цю ідею.
- Замикання (closure) — функція, в тілі якої є посилання на змінні, які було оголошено зовні тіла цієї функції в оточуючому коді і які не є її параметрами.

Розглянемо приклад:

```
def outer():
    message = 'Hi there!'

    def inner():
        print(message)
    return inner


f = outer()
f()
```

Отримаємо:

Hi there!

Що тут відбувається?

Функція inner "бачить" змінну message функції outer, вона знайшла її в області видимості enclosing.

Функція outer повертає об'єкт функції inner, результат ми присвоїли змінній f.

Змінна f вказує на об'єкт функції inner, яка, у свою чергу, "пам'ятає" змінну message. Об'єкт функції inner не знищено, отже і не знищено усі його об'єкти, зокрема змінну message.

Фабрика функцій (зовнішній функції будемо передавати параметри)

```
def outer(message):
    def inner():
        print(message)

    return inner


hello_func = outer('Hello')
bye_func = outer('Bye')

hello_func()
bye_func()
```

Отримаємо:

```
Hello
Bye
```

У цьому прикладі функція inner "побачила" змінну message в області видимості enclosing функції outer яка, у свою чергу, є параметром останньої.

Функція outer є як би "фабрикою" інших функцій, при цьому функції вона може "виробляти" з різними властивостями.

Декоратори

Спочатку вказуємо ім'я декоратора після значка @, і на наступному рядку ту функцію, яку треба декорувати вищенаведеним декоратором.

```
>>> def gift_iphone():
...     print('Я — айфон!')
...
>>> gift_iphone = decorator_function(gift_iphone)
```

буде ідентичним такому:

```
>>> @decorator_function
... def gift_iphone():
...     print('Я — айфон!')
```

Декоратор у нас теж буде функцією:

```
>>> def decorator_function(gift_to_wrap_function):
...     def wrap_function():
...         print('Я — святкова обгортка! Я обгорну подарунок!')
...         gift_to_wrap_function()
...         print('Подарунок обгорнуто!')
...     return wrap_function
```

Ланцюжки з декораторів. Синтаксис Python дозволяє одночасне використання декількох декораторів.

```
>>> def bread(func):
...     def wrapper():
...             print('Хліб')
...             func()
...             print('Хліб')
...     return wrapper
...
>>> def salad(func):
...     def wrapper():
...             print('Зеленина')
...             func()
...             print('Зеленина')
...     return wrapper
...
>>> @bread
```

Застосування тих самих декораторів, але без "магічного" символа @:

```
stake = bread(salad(stake))
```

Зауважте що послідовність застосування декораторів має значення.

Шаблони декораторів

Декоратор без параметрів:

```
from functools import wraps
def назва_декоратора(функція_що_декорується):
    @wraps(функція_що_декорується)
    def inner(параметри_функції_що_декорується):
        ...
        функція_що_декорується(параметри_функції_що_декорується)
        ...
    return inner
```

Шаблони декораторів

Декоратор з параметрами:

```
from functools import wraps
def назва_декоратора(параметри_декоратора):
    def decorator(функція_що_декорується):
        @wraps(функція_що_декорується)
        def inner(параметри_функції_що_декорується):
            ...
            функція_що_декорується(параметри_функції_що_декорується)
            ...
        return inner
    return decorator
```