



СХОЖІСТЬ ОБ'ЄКТІВ: НАСЛІДУВАННЯ ТА ПОЛІМОРФІЗМ

Питання 8.3

Базове наслідування

- Технічно, кожен клас, який Ви створюєте, використовує наслідування.
 - Всі класи Python є підкласами (породженими класами) спеціального класу object.
 - Якщо явно не наслідуємо від іншого класу, автоматично встановлюється успадкування від object.
 - Можна записати це явно

```
class MySubClass(object):  
    pass
```

- Суперклас (батьківський клас) – це клас, від якого відбувається наслідування.
 - Підклас (subclass, породжений клас) – це клас, що наслідує від суперкласу.

Як застосовується на практиці?

- Доповнення функціональності вже існуючого класу.
 - Приклад: простий менеджер контактів, який зберігає імена та e-mail-адреси людей.
 - У класі контакти зберігаються у списку (class variable)

```
1 class Contact:
2     all_contacts = []
3     def __init__(self, name, email):
4         self.name = name
5         self.email = email
6         Contact.all_contacts.append(self)
```

- Список all_contacts є частиною оголошення класу, тому спільний для всіх екземплярів класу.
 - Список лише один, доступ до нього: Contact.all_contacts.
 - Менш очевидний доступ: self.all_contacts для будь-якого екземпляру класу Contact.
 - Якщо поля (змінної) об'єкту немає, компілятор звернеться до класу.

Нехай деякі з контактів – постачальники товару, і їх треба впорядкувати

```
1 class Supplier(Contact):
2     def order(self, order):
3         print("Якби це була реальна система, ми відправили б"
4               "'{}' замовлення до '{}'".format(order, self.name))
>>> c = Contact("Some Body", "somebody@example.net")
>>> s = Supplier("Sup Plier", "supplier@example.net")
>>> print(c.name, c.email, s.name, s.email)
Some Body somebody@example.net Sup Plier supplier@example.net
>>> c.all_contacts
[<__main__.Contact object at 0xb7375ecc>,
 <__main__.Supplier object at 0xb7375f8c>]
>>> c.order("I need pliers")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Contact' object has no attribute 'order'
>>> s.order("I need pliers")
If this were a real system we would send 'I need pliers' order to
'Sup Plier '
```

- Можемо додати метод `order()` у клас `Contact`.
 - Проте це дасть можливість замовити товар від усіх контактів.
 - Замість цього створимо новий клас `Supplier`, який працює як `Contact`, проте з додатковим методом `order()`.

Розширення функціональності

- Наслідування дає змогу розширяти функціонал вбудованих класів.

```
1 class ContactList(list):
2     def search(self, name):
3         '''Повертає всі контакти, які містять у назві шукане значення'''
4         matching_contacts = []
5         for contact in self:
6             if name in contact.name:
7                 matching_contacts.append(contact)
8         return matching_contacts
9
10 class Contact:
11     all_contacts = ContactList()
12     def __init__(self, name, email):
13         self.name = name
14         self.email = email
15         self.all_contacts.append(self)
16
17 c1 = Contact("John A", "johna@example.net")
18 c2 = Contact("John B", "johnb@example.net")
19 c3 = Contact("Jenna C", "jennac@example.net")
20 print([c.name for c in Contact.all_contacts.search('John')])
```

['John A', 'John B']

Як змінився вбудований синтаксис?

- Створення порожнього списку [] – скорочений синтаксис для list():
 - ```
>>> [] == list()
True
```
- Тип даних list – це клас, який можна розширювати. А сам list наслідує object:
  - ```
>>> isinstance([], object)
True
```

Ще один приклад

- Можна розширити (extend) клас dict:

```
1 class LongNameDict(dict):
2     def longest_key(self):
3         longest = None
4         for key in self:
5             if not longest or len(key) > len(longest):
6                 longest = key
7         return longest
8
9 longkeys = LongNameDict()
10 longkeys['hello'] = 1
11 longkeys['longest yet'] = 5
12 longkeys['hello2'] = 'world'
13 print(longkeys.longest_key())
```

longest yet

Заміщення (Overriding) та функція `super()`

- Наслідування доречне для додавання нової поведінки, проте як змінити вже існуючу?
 - Наш клас `Contact` дозволяє лише ім'я та e-mail.
 - Задамо атрибут `phone` для контакту; якщо хочемо ініціалізувати його при створенні об'єкту, замістимо `__init__()`.
 - Заміщення – це заміна тіла методу суперкласу на нове в межах підкласу (subclass).
 - Спеціального синтаксису не потрібно, новий метод автоматично викликатиметься замість методу суперкласу.

```
class Friend(Contact):  
    def __init__(self, name, email, phone):  
        self.name = name  
        self.email = email  
        self.phone = phone
```

- Версія з заміщенням (класи `Contact` та `Friend` мають дубльований код)

```
class Friend(Contact):  
    def __init__(self, name, email, phone):  
        super().__init__(name, email)  
        self.phone = phone
```


Замістити можна будь-який метод

- `super()` не працює у старих версіях Python.
 - У Python 2 буде викликатись `super(EmailContact, self).__init__()`.
 - Зауважте, що першим аргументом є назва дочірнього класу, а клас перед object.
- Виклик `super()` можна здійснити в будь-якому та будь-де в методі.
 - Робити його першим рядком методу не потрібно.
 - Наприклад, потрібно перевірити або маніпулювати вхідними параметрами до обгортання (форвардингу) їх у суперклас.

Множинне наслідування (Multiple inheritance)

- Підклас, що успадковує від кількох батьківських класів, має доступ до їх функціональності.
 - На практиці багато експертів рекомендують не використовувати.
- Найпростіша та найкорисніша форма множинного наслідування – *домішки (mixin)*.
 - Домішка є суперкласом, який не пристосований до самостійного існування, проте має наслідуватись іншими класами, щоб надати додаткову функціональність.

```
1 class Contact:
2     all_contacts = []
3     def __init__(self, name, email):
4         self.name = name
5         self.email = email
6         Contact.all_contacts.append(self)
7
8 class MailSender:
9     def send_mail(self, message):
10        print("Sending mail to " + self.email)
11        # Тут додається логіка для e-mail
12
13 class EmailableContact(Contact, MailSender):
14     pass
15
16 e = EmailableContact("John Smith", "jsmith@example.net")
17 print(Contact.all_contacts)
18 e.send_mail("Hello, test e-mail here")
```

- Наприклад, хочемо додати в клас Contact можливість відправки e-mail у self.email.
- Надсилання e-mail – поширена задача, яку можемо захотіти використати для багатьох інших класів.
- Клас EmailableContact опирається на класи Contact і MailSender, використовуючи множинне наслідування.
- Ініціалізатор Contact досі додає новий контакт до списку, а домішка може відправляти повідомлення на self.email.

```
[<__main__.EmailableContact object at 0x0000027EAE895B38>]
Sending mail to jsmith@example.net
```

Які альтернативні способи реалізації?

- **Використовувати одиничне наслідування (*single inheritance*).**
 - Додати функцію `send_mail()` у підклас.
 - Недолік: дублювання функціоналу `e-mail` для всіх інших класів, що потребують `e-mail`.
- **Створити окрему функцію для відправки повідомлення.**
 - викликати її з коректною електронною адресою (параметр функції).
- **Використовувати композицію замість наслідування.**
 - Наприклад, `EmailableContact` може містити об'єкт `MailSender` замість наслідування класу.
- **Застосувати *monkey-patching* для класу `Contact`, щоб мати метод `send_mail()` після створення класу.**
 - Визначається функція, яка приймає `self`-аргумент та встановлює його в якості атрибуту для існуючого класу.

Проблеми множинного наслідування

- Нехай існує кілька суперкласів.
 - Як дізнатись, який із них викликати?
 - Як дізнатись, у якому порядку викликати їх?
- Додамо домашню адресу (вулиця, місто, країна та ін.) у клас Friend.
 - Можна передати кожен рядок як параметр у метод `__init__()` класу Friend.
 - Можна зберегти рядки в кортежі чи словнику, а потім передати в `__init__()` одним аргументом.
 - Можна створити новий клас Address, щоб інкапсулювати ці рядки, а потім передати екземпляр цього класу в метод `__init__()`.
 - Перевага: можна прописати додаткову поведінку.
 - Це приклад композиції.
 - Потім такий клас можна аналогічно застосувати для інших сутностей: будівель, організацій тощо.

Проте наслідування теж робочий варіант

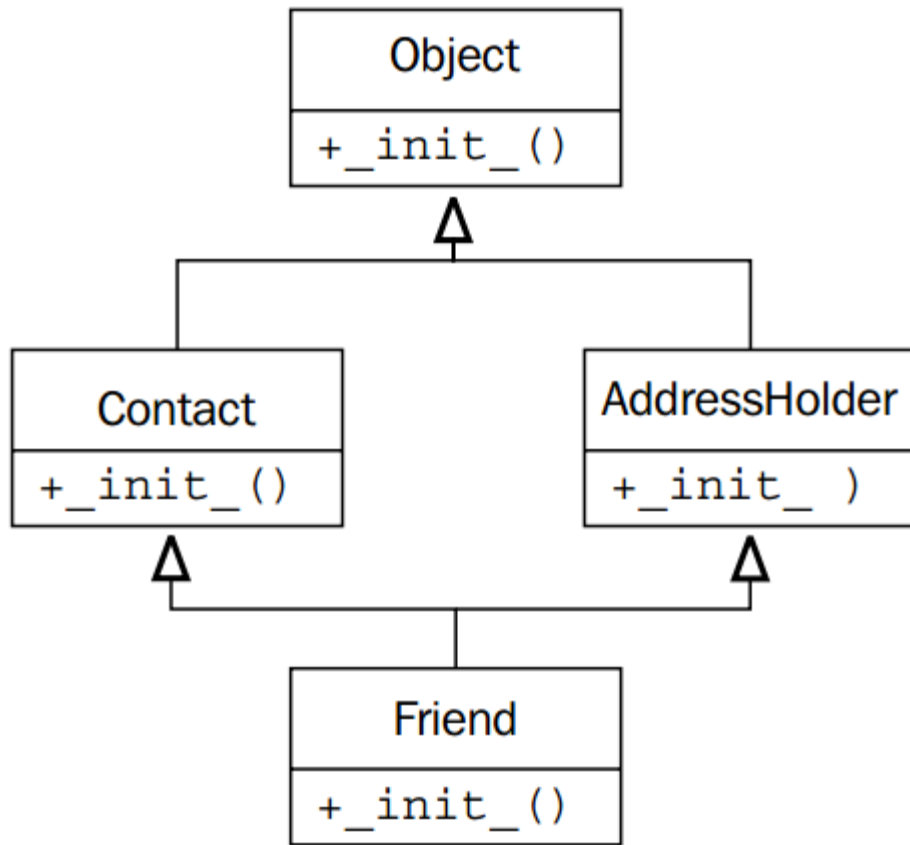
- Додамо новий клас AddressHolder для зберігання адреси

```
class AddressHolder:
    def __init__(self, street, city, state, code):
        self.street = street
        self.city = city
        self.state = state
        self.code = code
```

- Проблемна частина: тепер маємо два батьківських методи __init__(), обидва з яких потрібно ініціалізувати.

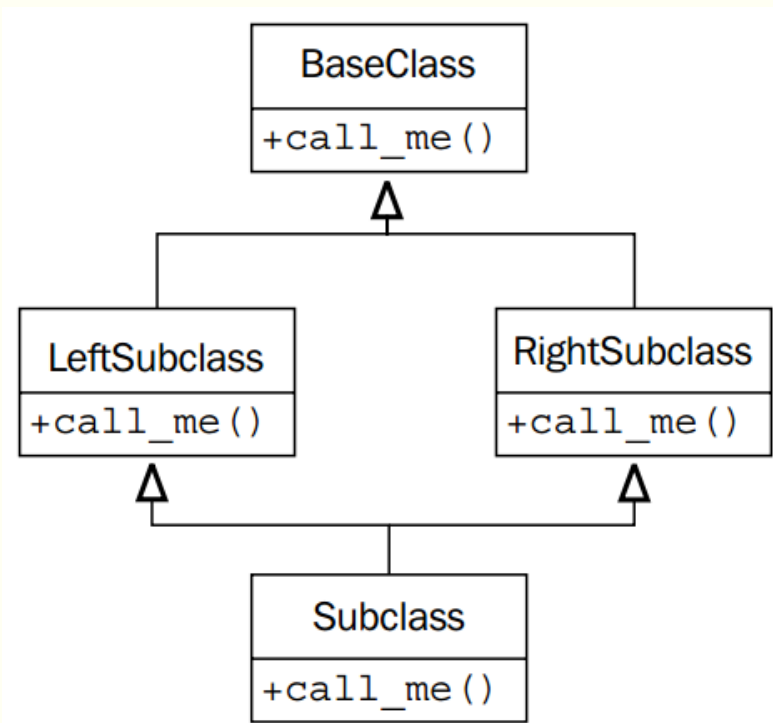
```
class Friend(Contact, AddressHolder):
    def __init__(self, name, email, phone, street, city, state, code):
        Contact.__init__(self, name, email)
        AddressHolder.__init__(self, street, city, state, code)
        self.phone = phone
```

Схема наслідування та проблеми такого підходу



- Суперклас може явно не ініціалізуватись, якщо знехтуємо явним викликом ініціалізатора.
 - Цей приклад може інколи «падати» в поширених сценаріях роботи.
 - Наприклад, вставка даних в БД, з якою немає зв'язку.
- Можливість кількарázового виклику суперкласу object.
 - Батьківський клас двічі **set up**.
 - У деяких ситуаціях об'єкт такого класу може викликати катастрофу.
 - Уявіть, що намагаєтесь двічі підключитись до БД у кожному запиті.
- Базовий клас краще викликати лише один раз
 - Порядок виклику методів можна адаптувати на льоту, змінюючи атрибут `__mro__` (Method Resolution Order) для класу.
 - Додаткова інформація [тут](#).

Схожий приклад



- Приклад показує, як кожний заміщений метод `call_me()` напряду викликає одноіменний батьківський метод.

```
1 class BaseClass:
2     num_base_calls = 0
3     def call_me(self):
4         print("Calling method on Base Class")
5         self.num_base_calls += 1
6
7 class LeftSubclass(BaseClass):
8     num_left_calls = 0
9     def call_me(self):
10         BaseClass.call_me(self)
11         print("Calling method on Left Subclass")
12         self.num_left_calls += 1
13
14 class RightSubclass(BaseClass):
15     num_right_calls = 0
16     def call_me(self):
17         BaseClass.call_me(self)
18         print("Calling method on Right Subclass")
19         self.num_right_calls += 1
20
21 class Subclass(LeftSubclass, RightSubclass):
22     num_sub_calls = 0
23     def call_me(self):
24         LeftSubclass.call_me(self)
25         RightSubclass.call_me(self)
26         print("Calling method on Subclass")
27         self.num_sub_calls += 1
```

Робота з цими класами

```
29 s = Subclass()
30 s.call_me()
31
32 print(s.num_sub_calls, s.num_left_calls,
33       s.num_right_calls, s.num_base_calls)
```

- Результати виводу – базовий клас викликається двічі:

```
Calling method on Base Class
Calling method on Left Subclass
Calling method on Base Class
Calling method on Right Subclass
Calling method on Subclass
1 1 1 2
```

- Це призведе до неявних проблем, наприклад, подвійного внеску грошей на депозит.

Внесемо зміни

```
1 class BaseClass:
2     num_base_calls = 0
3     def call_me(self):
4         print("Calling method on Base Class")
5         self.num_base_calls += 1
6
7 class LeftSubclass(BaseClass):
8     num_left_calls = 0
9     def call_me(self):
10        super().call_me()
11        print("Calling method on Left Subclass")
12        self.num_left_calls += 1
13
14 class RightSubclass(BaseClass):
15     num_right_calls = 0
16     def call_me(self):
17        super().call_me()
18        print("Calling method on Right Subclass")
19        self.num_right_calls += 1
20
21 class Subclass(LeftSubclass, RightSubclass):
22     num_sub_calls = 0
23     def call_me(self):
24        super().call_me()
25        print("Calling method on Subclass")
26        self.num_sub_calls += 1
```

- Хочемо тільки викликати «наступний» метод в ієрархії класів, а не «батьківському» методі.
 - Фактично, метод next() може не відноситись до батьківського чи дочірнього класу щодо поточного класу.
 - Допомогає ключове слово super.

```
Calling method on Base Class
Calling method on Right Subclass
Calling method on Left Subclass
Calling method on Subclass
1 1 1 1
```

Різні набори аргументів

- У методі `__init__()` класу `Friend` спочатку викликали `__init__()` для обох батьківських класів, з різними наборами аргументів:
 - `Contact.__init__(self, name, email)`
 - `AddressHolder.__init__(self, street, city, state, code)`
- Не обов'язково відомо, `super` з якого класу спробує виконати ініціалізацію першим.
 - Потрібна можливість передачі додаткових аргументів, щоб наступні виклики `super` (в інших підкласах) отримували правильні аргументи.
 - Якщо перший виклик `super()` передає аргументи `name` та `email` в `Contact.__init__`, а в `Contact.__init__` потім теж викликається `super()`, він повинен мати можливість передавати аргументи, пов'язані з адресою, в «наступний» метод – `AddressHolder.__init__()`.
- На жаль, вирішення цієї проблеми потрібно планувати з самого початку.
 - Необхідно проектувати так, щоб список параметрів базового класу приймав (асепт) `keyword arguments` для будь-яких параметрів, що не вимагаються кожною реалізацією підкласу.
 - Також потрібно забезпечити вільне прийняття методом неочікуваних аргументів та їх передачу до виклику `super()`, якщо вони необхідні подальшим у порядку наслідування методам.

Синтаксис функцій у Python надає для цього всі інструменти, проте код дещо ускладнюється

```
1 class Contact:
2     all_contacts = []
3     def __init__(self, name='', email='', **kwargs):
4         super().__init__(**kwargs)
5         self.name = name
6         self.email = email
7         self.all_contacts.append(self)
8
9 class AddressHolder:
10     def __init__(self, street='', city='', state='', code='', **kwargs):
11         super().__init__(**kwargs)
12         self.street = street
13         self.city = city
14         self.state = state
15         self.code = code
16
17 class Friend(Contact, AddressHolder):
18     def __init__(self, phone='', **kwargs):
19         super().__init__(**kwargs)
20         self.phone = phone
```

- Які аргументи передавати в Friend.__init__?
 - Тут зазвичай використовують docstring.
- Навіть ця реалізація неефективна, якщо хочемо повторно використовувати змінні в батьківських класах.
 - Коли передаємо змінну **kwargs у super(), словник не включає жодну змінну, які були включені при явній передачі аргументів.
 - Наприклад, у Friend.__init__() виклик super() не має phone у словнику kwargs.
 - Якщо інший клас потребуватиме параметр phone, необхідно **забезпечити його передачу** в словник.
 - Якщо забудемо, суперклас не повідомить про помилку, а просто **присвоїть значення за умовчанням** (тут порожній рядок).

Способи забезпечення передачі значення

- Припустимо, що клас Contact потребує ініціалізації з параметром phone, а класу Friend також буде потрібний доступ до нього.
- Можемо зробити наступне:
 - **Не включати phone як явний аргумент.** Залиште його у словнику kwargs. Клас Friend може переглянути словник – kwargs['phone'].
 - **Зробити phone явним аргументом, проте оновити словник kwargs** перед передачею його в super():
kwargs['phone'] = phone.
 - **Зробити phone явним аргументом, проте оновити словник kwargs** за допомогою методу kwargs.update(). Корисно, якщо є кілька аргументів для оновлення. Створити словник для передачі в update() можна за допомогою конструктора dict(phone=phone), або {'phone': phone}.
 - **Зробити phone явним аргументом, проте передати його явно:** super().__init__(phone=phone, **kwargs).

Поліморфізм

- Різна поведінка залежно від обраного для використання підкласу без явного вказування цього підкласу.
 - Уявіть програму-медіаплеєр, якій потрібно завантажувати об'єкт `AudioFile`, а потім його відтворювати.
 - Додамо в об'єкт метод `play()`, який відповідає за розпакування або виділення аудіо та його передачу на звукову карту чи динаміки.
 - Найпростіший виклик такого методу: `audio_file.play()`
 - Проте процес розпаковки значно відрізняється залежно від типу файлу.
 - `.wav`-файли зберігаються нестисненими, а файли `.mp3`, `.wma`, `.ogg` мають абсолютно різні алгоритми стиснення.
- Спростимо проект за допомогою наслідування та поліморфізму.
 - Кожен тип файлу можна представити окремим підкласом `AudioFile`, наприклад, `WavFile`, `MP3File`.
 - Кожен з цих класів матиме свій метод `play()`, проте реалізуватись буде по-різному.
 - Об'єкту `media player` не потрібно знати, до якого підкласу `AudioFile` будемо звертатись; він викликає `play()` та поліморфно дозволяє об'єкту попіклуватись про фактичні деталі відтворення аудіо.

```

1 class MyAudioFile:
2     def __init__(self, filename):
3         if not filename.endswith(self.ext):
4             raise Exception("Invalid file format")
5         self.filename = filename
6
7 class MyMP3File(MyAudioFile):
8     ext = "mp3"
9     def play(self):
10         print("Відтворення {} в форматі mp3".format(self.filename))
11
12 class MyWavFile(MyAudioFile):
13     ext = "wav"
14     def play(self):
15         print("Відтворення {} в форматі wav".format(self.filename))
16
17 class MyOggFile(MyAudioFile):
18     ext = "ogg"
19     def play(self):
20         print("Відтворення {} в форматі ogg".format(self.filename))
21
22
23 ogg = MyOggFile("myfile.ogg")
24 ogg.play()
25 mp3 = MyMP3File("myfile.mp3")
26 mp3.play()
27 not_an_mp3 = MyMP3File("myfile.ogg")

```

- Усі аудіо файли перевіряють коректність розширення в процесі ініціалізації.
 - Якщо назва файлу не закінчується коректно, викидається виняток.
 - Відсутність фактичного зберігання посилання на змінну ext в AudioFile не усуває можливість отримати доступ до нього в підкласі.

```

відтворення myfile.ogg в форматі ogg
відтворення myfile.mp3 в форматі mp3
Traceback (most recent call last):

```

```

File "<ipython-input-10-3b3c912a01fc>", line 1, in <module>
    runfile('C:/Users/spuasson/Desktop/untitled11.py', wdir='C:/Users/spuasson/Desktop')

```

```

File "C:\ProgramData\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 710, in runfile
    execfile(filename, namespace)

```

```

File "C:\ProgramData\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 101, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

```

```

File "C:/Users/spuasson/Desktop/untitled11.py", line 29, in <module>
    not_an_mp3 = MyMP3File("myfile.ogg")

```

```

File "C:/Users/spuasson/Desktop/untitled11.py", line 4, in __init__
    raise Exception("Invalid file format")

```

```

Exception: Invalid file format

```



ДЯКУЮ ЗА УВАГУ!

Наступне питання: Об'єктно-орієнтоване програмування в дії

Абстрактні базові класи

- Сказати, чи відповідає клас необхідному «протоколу» не завжди просто, тому Python представляє абстрактні базові класи (ABC).
 - Вони визначають набір методів та властивостей, які класу потрібно реалізувати (implement) для того, щоб вважатись duck-type екземпляром цього класу.
 - Клас може розширяти абстрактний базовий клас, щоб створювати свої екземпляри, проте повинен реалізувати всі абстрактні методи.
- На практиці створювати нові абстрактні класи потрібно рідко, проте можна знайти випадки, коли потрібно реалізувати екземпляри вже існуючих абстрактних базових класів.

Використання абстрактного базового класу

- Більшість абстрактних базових класів, що існують у Python Standard Library, знаходяться в модулі `collections`.

- Один з найпростіших – клас `Container`.

```
In [11]: from collections import Container
```

```
In [12]: Container.__abstractmethods__  
Out[12]: frozenset({'__contains__'})
```

- У ньому є лише один абстрактний метод, який необхідно реалізувати: `__contains__()`.

```
In [13]: help(Container.__contains__)  
Help on function __contains__ in module collections.abc:
```

- `__contains__(self, x)`
Викликає один аргумент – значення, яке користувач очікує знайти в контейнері.

Цей метод реалізується в list, str та dict

- Проте можна також визначити простий контейнер, який tells us whether a given value is in the set of odd integers:

```
1 from collections import Container
2
3 class OddContainer:
4     def __contains__(self, x):
5         if not isinstance(x, int) or not x % 2:
6             return False
7         return True
8
9 odd_container = OddContainer()
10 print(isinstance(odd_container, Container))
11 print(issubclass(OddContainer, Container))
```

- Результати виводу:
 - True
 - True