Object Oriented Programming
Using
Python

Teemu Matilainen
teemu.matilainen@savonia.fi

Lecture 7#

- **Inheritance continues…**
- **You understand how to use iterators**
- **You know how to use** comprehensions **within own classes**
- **You are able to restrict the visibility of inherited attributes in a subclass**
- **You are able to restrict the visibility of inherited methods in a subclass**

Inheritance … Thesis class… Override…

The constructor within the Thesis class invokes the constructor in the base class Book with the specified arguments for name and author.

In addition, the constructor in the derived class assigns a value to the attribute ‘author’.

Derived class can override method in the base class!

```python
from book import Book
from author import Author

class Thesis(Book):
    def __init__(self, name: str, author: str):
        self.name = name
        self.add_author(author)
        super().__init__(name, "code", 0)
        super().add_author(Author(author, 0))

    def add_author(self, author: str):
        self.author = author
```

```python
from book import Book
from author import Author
from library import Library
from bookContainer import BookContainer
from bookShelf import Bookshelf
from thesis import Thesis
```

```python
############ Thesis ########################
thesis = Thesis(name = "Universe of Arduino", author = "Savonia student")

# Print out the values of the attributes
print(thesis.name)
thesis.summary()
```

```
Universe of Arduino
Book: Universe of Arduino
Authors: 1
Authors names are from each book: ['Savonia student']
```

Access modifiers within base class

NoteBook with private attributes

```python
class Notebook:
    ##########3 A Notebook stores notes in string format ########

    def __init__(self):
        self.__notes = [] # private attribute

    def add_note(self, note):
        self.__notes.append(note)

    def retrieve_note(self, index):
        return self.__notes[index]

    def all_notes(self):
        return ",".join(self.__notes)
```

NoteBook with protected attributes

```python
class Notebook:
    ##########3 A Notebook stores notes in string format ########

    def __init__(self):
        self._notes = [] # protected attribute

    def add_note(self, note):
        self._notes.append(note)

    def retrieve_note(self, index):
        return self._notes[index]

    def all_notes(self):
        return ",".join(self._notes)
```

Access modifiers: - Private

- If a trait is defined as private in the base class, it is not directly accessible in any derived classes. Let's take a look at an example. In the Notebook class below the notes are stored in a list, and the list attribute is private:

```python
from notebookPrivate import Notebook

class NotebookNew(Notebook):
    ######## A better Notebook with search functionality ##########3
    def __init__(self):
        # This works, the constructor is public despite the underscores
        super().__init__()

    # This causes an error
    def find_notes(self, search_term):
        found = []
        # the derived class can't access it directly
        for note in self.__notes:
            if search_term in note:
                found.append(note)

        return found
```

Access modifiers: - Private

- If the integrity of the class is key, making the list attribute notes private makes sense. The class provides the client with suitable methods for adding and browsing notes, after all. This approach becomes problematic if we define a new class NotebookPro, which inherits the Notebook class. The private list attribute is not accessible to the client, but neither is it accessible to the derived classes. If we try to access it, as in the find_notes method below, we get an error:

```python
from notebookPrivate import Notebook

class NotebookNew(Notebook):
    ######## A better Notebook with search functionality ##########3
    def __init__(self):
        # This works, the constructor is public despite the underscores
        super().__init__()

    # This causes an error
    def find_notes(self, search_term):
        found = []
        # the derived class can't access it directly
        for note in self.__notes:
            if search_term in note:
                found.append(note)

        return found
```

```python
from notebookNew import NotebookNew


if __name__ == "__main__":
    nb = NotebookNew()
    nb.add_note("the term is...")
    print(nb.find_notes("term"))
```

```
 9      # This causes an error
10      def find_notes(self, search_term):
11          found = []
12          # the derived class can't access it directly
13          for note in self.__notes:

Exception has occurred: AttributeError ×
'NotebookNew' object has no attribute '_NotebookNew__notes'
  File "C:\Users\sh23740\OneDrive - Savonia-ammattikorkeakoulu\Savonia\Teaching Spring 2024\Object Oriented Programming\Code examples\notebookNew.py", line 13, in find_notes
    for note in self.__notes:
                ^^^^^^^^^^^^
```

Access modifiers:
- Private
- Protected

```
def __init__(self):
    self.__notes = [] # private attribute
```

```
def __init__(self):
    self._notes = [] # protected attribute
```

- Many object oriented programming languages have a feature, usually a special keyword, for protecting traits. This means that a trait should be hidden from the clients of the class, but kept accessible to its subclasses. Python in general abhors keywords, so no such feature is directly available in Python. Instead, there is a convention of marking protected traits in a certain way.

Access modifiers:
- Protected

```python
from notebookProtected import Notebook

class NotebookNew(Notebook):
    ######## A better Notebook with search functionality ##########3
    def __init__(self):
        # This works, the constructor is public despite the underscores
        super().__init__()

    # This causes an error
    def find_notes(self, search_term):
        found = []
        # the derived class can't access it directly
        for note in self._notes:
            if search_term in note:
                found.append(note)

        return found
```

```python
from notebookNew import NotebookNew


if __name__ == "__main__":
    nb = NotebookNew()
    nb.add_note("the term is...")
    print(nb.find_notes("term"))
```

```
['the term is...']
```

Access modifiers: Table

Below we have a table for the visibility of attributes with different access modifiers:

Access modifier	Example	Visible to client	Visible to derived class
Public	**self.name**	**yes**	**yes**
Protected	**self._name**	**no**	**yes**
Private	**self.__name**	**no**	**no**

Iteration?

We use **for** statement to iterate through different data structures, objects and collections of items. A typical use:

```
def count_positives_values(l: list):
    i = 0
    for item in l:
        if item > 0:
            i += 1
    return i

l = [1, 2, 3, 4, -1, -2, -2, -5, -6, -7]

postive_values = count_positives_values(l)
print(postive_values)
```

One more operator example and Iteration

Creating custom iterable classes is achievable as well. This becomes beneficial when the primary function of the class revolves around managing a group of items. For instance, the Bookshelf class demonstrated earlier would be well-suited for this, allowing a for loop to conveniently navigate through the books on the shelf. A similar scenario applies to a student register, where the ability to iterate through the collection of students would be advantageous.

To enable a class to be iterable, it is necessary to implement the iterator methods __**iter**__ and __**next**__. We will delve into the details of these methods in the upcoming example.

One more operator example and Iteration

```python
from book import Book

class Antique_store:
    def __init__(self):
        self._books = []

    def add_book_to_shelf(self, book: Book):
        self._books.append(book)

    def __iter__(self):
        self.count = 0
        return self

    def __next__(self):
        if self.count < len(self._books):
            book = self._books[self.count]
            self.count += 1
            return book
        else:
            raise StopIteration
```

One more operator example and Iteration

The **__iter__** method initializes the iteration variable or variables. In this context, a simple counter containing the index of the current item in the list is sufficient. Additionally, the **__next__** method is also required, responsible for returning the next item in the iterator. In the aforementioned example, this method returns the item at index n from the list within the `Antique_store` object, while also incrementing the iterator variable.

Upon traversing all objects, the **__next__** method triggers the **StopIteration** exception. While the process is akin to raising other exceptions, Python automatically handles this particular exception. Its purpose is to notify the code invoking the iterator (e.g., a for loop) that the **iteration has concluded**.

One more operator example and Iteration

```python
from book import Book
from antique_store import Antique_store

if __name__ == "__main__":
    book_1 = Book("Never ending storie 1", "123", 23)
    book_2 = Book("Never ending storie 2", "123", 23)
    book_3 = Book("Never ending storie 3", "123", 23)
    book_4 = Book("Never ending storie 4", "123", 23)

    a_store = Antique_store()
    a_store.add_book_to_shelf(book_1)
    a_store.add_book_to_shelf(book_2)
    a_store.add_book_to_shelf(book_3)
    a_store.add_book_to_shelf(book_4)

    for book in a_store:
        print(book.name)
```

```
Never ending storie 1
Never ending storie 2
Never ending storie 3
Never ending storie 4
```

Exercise

Create a MyShoppingList class and adjust the class so that it is **iterable** and can thus be used as follows:

```
Beer IV: 10 units
Chips: 5 units
Cheese: 1 units
```

```python
class MyShoppingList:
    def __init__(self):
        self.items = {}

    def add_item(self, item, count):
        if self.items.get(item) == None:
            self.items[item] = count
        else:
            self.items[item] += count

    def __iter__(self):
        #TODO
        pass

    def __next__(self):
        #TODO
        pass
```

```python
if __name__ == "__main__":
    shopping_list = MyShoppingList()
    shopping_list.add_item("Beer IV", 10)
    shopping_list.add_item("Chips", 5)
    shopping_list.add_item("Cheese", 1)

for product in shopping_list:
    print(f"{product[0]}: {product[1]} units")
```

Exercise…

Create a MyShoppingList class and adjust the class so that it is **iterable** and can thus be used as follows:

```python
class MyShoppingList:
    def __init__(self):
        self.items = {}

    def add_item(self, item, count):
        if self.items.get(item) == None:
            self.items[item] = count
        else:
            self.items[item] += count

    def __iter__(self):
        self.index = 0
        self.keys = list(self.items.keys())
        return self

    def __next__(self):
        if self.index < len(self.keys):
            index = self.index
            self.index += 1
            return self.keys[index], self.items[self.keys[index]]
        else:
            raise StopIteration
```

```python
if __name__ == "__main__":
    shopping_list = MyShoppingList()
    shopping_list.add_item("Beer IV", 10)
    shopping_list.add_item("Chips", 5)
    shopping_list.add_item("Cheese", 1)

for product in shopping_list:
    print(f"{product[0]}: {product[1]} units")
```

Exercise…

Create a MyShoppingList class and adjust the class so that it is **iterable** and can thus be used as follows:

```
class ShoppingListIter:
    def __init__(self):
        self.items = {}

    def add_item(self, item):
        self.items[item] = self.items.get(item, 0) + 1

    def __iter__(self):
        self.current_item = None
        self.item_iterator = iter(self.items.items())
        return self

    def __next__(self):
        try:
            self.current_item = next(self.item_iterator)
            return self.current_item
        except StopIteration:
            raise StopIteration
```

```
if __name__ == "__main__":
    shopping_list = MyShoppingList()
    shopping_list.add_item("Beer IV", 10)
    shopping_list.add_item("Chips", 5)
    shopping_list.add_item("Cheese", 1)

for product in shopping_list:
    print(f"{product[0]}: {product[1]} units")
```

iter()

- The iter() function is used to obtain an iterator from an object that implements the iteration protocol.

- If the object has a __iter__() method, the iter() function calls this method to get the iterator.
- If the object doesn't have a __iter__() method but has a __getitem__() method, iter() creates an iterator that accesses elements using integer indices.
- If neither method is present, iter() raises a TypeError.

next()

- The next() function is used to retrieve the next item from an iterator.
- It takes an iterator as its first argument and an optional default value as its second argument.
- If the iterator has more items, next() returns the next item; otherwise, it raises a StopIteration exception if no default value is provided.
- If a default value is provided, next() returns the default value when the iterator is exhausted.

Simple Iteration

```
my_iter = iter([1, 2, 3, 4])

value = next(my_iter, None)
print(value)
value = next(my_iter, None)
print(value)
value = next(my_iter, None)
print(value)
value = next(my_iter, None)
print(value)
```