oop

March 6, 2024

1 Lecture 3: Advanced object-oriented programming

- Programming paradigms
- The motivation for OOP
- Advanced concepts: `__special__` methods, class attributes, class methods, static methods
- Inheritance: class hierarchies, abstract classes and methods

1.1 Who am I?

- 2021: BA Computational Linguistics and Comparative Linguistics
- 2024 ( ): MA Computational Linguistics and Neuroinformatics
- Text simplification research @ UZH
- Teaching @ UZH & ZB
- Previously: web development @ Idiotikon (Swiss German dictionary)

andreas@cl.uzh.ch

1.2 Learning objectives

By the end of this lecture, you should:

- Understand the characteristics and advantages of OOP
- Be able to design and implement your own classes
- Understand what inheritance is and what problems it solves
- Understand the concepts of *class attributes/methods*, *parent/child classes*, *abstract classes/methods*, and know their syntax in Python
- Be able to understand and extend existing class hierarchies
- Be able to design and implement your own class hierarchies

Refer to the OOP cheat sheet on OLAT! We will expect you to understand these concepts at the midterm exam.

1.2.1 Quiz: OOP basics

pwa.klicker.uzh.ch/join/asaeub

1.3 Programming paradigms

- Most problems can be solved in many different *ways*
- A programming paradigm is a *way of programming*
- Programming paradigms are often used to characterize different programming languages

1.3.1 Example using functional programming

[ ]:

```python
def simulate_years(balance: float, interest_rate: float, num_years: int) -> float:
    for _ in range(num_years):
        balance = balance * (1 + interest_rate)
    return balance
```

[ ]:

```python
my_balance = 1000.00
my_interest_rate = 0.01

print("My current balance is: ", my_balance)
my_balance = simulate_years(my_balance, my_interest_rate, 10)
print("After 10 years, my balance will be: ", my_balance)
my_balance = simulate_years(my_balance, my_interest_rate, 10)
print("After 10 more years, my balance will be: ", my_balance)
```

1.3.2 Example using object-oriented programming

[ ]:

```python
class BankAccount:
    def __init__(self, balance: float, interest_rate: float):
        self.balance = balance
        self._interest_rate = interest_rate

    def simulate_years(self, num_years: int) -> None:
        for _ in range(num_years):
            self.balance = self.balance * (1 + self._interest_rate)
```

[ ]:

```python
my_account = BankAccount(1000.00, 0.01)
print("My current balance is: ", my_account.balance)
my_account.simulate_years(10)
print("After 10 years, my balance will be: ", my_account.balance)
my_account.simulate_years(10)
print("After 10 more years, my balance will be: ", my_account.balance)
```

1.3.3 Features of object-oriented programming

Encapsulation:

- State is handled inside the object
- State can be hidden from the user

Mutability:

- The object's state can be modified

Interface:

- The methods tell us what we can do with the object

Inheritance:

- Sharing code between classes with similar functionality

1.3.4 Features of functional programming

Pure functions:

- Functions don't have side-effects (same input → same output)

Immutability:

- Objects cannot be modified
- To modify state, create new objects and discard the old ones

Composition:

- Splitting functions into small, independent units (great for unit tests!)
- Maximizing reusability of functions

1.4 The intuition behind OOP

1.4.1 Two types of classes

Classes that **store data** (in attributes):

- a corpus
- a dictionary
- a chat conversation
- …

Classes that **do things** (with methods):

- a tokenizer
- a search engine
- a machine translation system
- …

(plus: classes that do both)

1.5 Example implementation: Word-by-word translation

[ ]:

```python
dictionary = {
    "hello": "hallo",
    "world": "Welt",
}

def translate(sentence: str, dictionary: dict):
    tokens = sentence.split()
    translation = " ".join(dictionary.get(token, token) for token in tokens)
    return translation

print(translate("hello world !", dictionary))
```

We will design two classes:

- `WordByWordTranslator`: can translate text from one language to another
- `ParallelDataset`: can store parallel texts in two languages (e.g., for training or evaluation)

We will get to know the following concepts along the way:

- Encapsulation, public interfaces
- `__special__` methods
- Class methods
- Class attributes
- Properties

1.5.1 `ParallelDataset` class

[ ]:
```python
from typing import Sequence, Iterator

class ParallelDataset:
    def __init__(self, source: Sequence[str], target: Sequence[str]):
        self._source = source
        self._target = target

    def get_parallel_item(self, index: int) -> tuple[str, str]:
        return self._source[index], self._target[index]
```

1.5.2 Encapsulation

- The **underlying data structure** is hidden from the user.
- The user interacts with the object through a **public interface**.
- Methods and attributes starting with an underscore _ are considered **private**.
- Private methods and attributes *should* not be accessed from outside the class (but nothing prevents you from doing so).

[ ]:
```python
dataset_de_en = ParallelDataset(
    ["hello world !", "this is a test"],
    ["hallo Welt !", "das ist ein Test"],
)
dataset_de_en.get_parallel_item(0)
```

1.5.3 Special methods

Special methods define how objects implement certain operations and functionalities which are built into Python's syntax, for example:

- `__init__()`: called when an object is **constructed** using `ClassName()`
- `__str__()`: called when an object is **converted to a string** using `str(obj)`
- `__lt__(other)`: called when an object is compared using `<` (less-than operator)
- `__gte__(other)`: called when an object is compared using `>=` (**greater-than-or-equal operator**)
- `__getitem__(index)`: called when an object is **indexed** using `obj[index]`
- `__iter__()`: called when an object is converted to an iterator (e.g. by using it in a `for` loop)

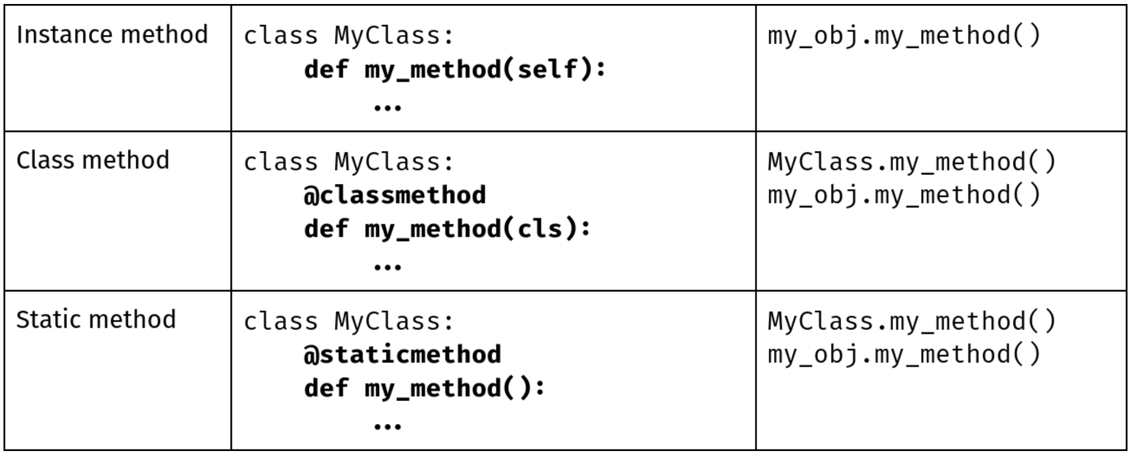
[ ]:

```python
class ParallelDataset:
    def __init__(self, source: Sequence[str], target: Sequence[str]):
        self._source = source
        self._target = target

    def __getitem__(self, index: int) -> tuple[str, str]:
        # Enables indexing with []
        return self._source[index], self._target[index]

    def __len__(self) -> int:
        # Enables len()
        return len(self._source)

    def __iter__(self) -> Iterator[tuple[str, str]]:
        # Enables iteration in for loops
        for i in range(len(self)):
            yield self[i]
```

[ ]:

```python
dataset_de_en = ParallelDataset(
    ["hello world !", "this is a test"],
    ["hallo Welt !", "das ist ein Test"],
)

print(dataset_de_en[0])
print(len(dataset_de_en))
for source, target in dataset_de_en:
    print(source, target)
```

1.5.4 Class methods and static methods



Instance method	class MyClass:
	    def my_method(self):
	        ...
		my_obj.my_method()

Class method	class MyClass:
	    @classmethod
	    def my_method(cls):
	        ...
		MyClass.my_method()
		my_obj.my_method()

Static method	class MyClass:
	    @staticmethod
	    def my_method():
	        ...
		MyClass.my_method()
		my_obj.my_method()

(from the OOP cheat sheet on OLAT)

Class methods are often used as an ``alternative constructor'':

[ ]:

```python
class ParallelDataset:
    def __init__(self, source: Sequence[str], target: Sequence[str]):
        self._source = source
        self._target = target

    @classmethod
    def load(cls, source_filename: str, target_filename: str) -> "ParallelDataset":
        with open(source_filename) as f:
            source = [line.strip() for line in f]
        with open(target_filename) as f:
            target = [line.strip() for line in f]
        return cls(source, target)

    def __getitem__(self, index: int) -> tuple[str, str]:
        # Enables indexing with []
        return self._source[index], self._target[index]

    def __len__(self) -> int:
        # Enables len()
        return len(self._source)

    def __iter__(self) -> Iterator[tuple[str, str]]:
        # Enables iteration in for loops
        for i in range(len(self)):
            yield self[i]
```

[ ]:

```python
dataset = ParallelDataset.load("translation/test.src", "translation/test.trg")

list(dataset)
```

1.5.5 Quiz: encapsulation

pwa.klicker.uzh.ch/join/asaeub

1.5.6 `WordByWordTranslator` class

[ ]:

```python
import sacrebleu

class WordByWordTranslator:
    def __init__(self, dictionary: dict[str, str]):
        self._dictionary = dictionary

    def translate(self, text: str) -> str:
        tokens = text.split()
        translation = " ".join(self._dictionary.get(token, token) for token in tokens)
        return translation

    def evaluate(self, dataset: ParallelDataset) -> float:
        translations = [self.translate(source) for source, _ in dataset]
        targets = [target for _, target in dataset]
        return sacrebleu.corpus_bleu(translations, [targets]).score

    @classmethod
    def load(cls, filename: str) -> "WordByWordTranslator":
        with open(filename) as f:
            dictionary = dict(line.strip().split("\t") for line in f)
        return cls(dictionary)
```

[ ]:

```
translator = WordByWordTranslator.load("translation/en-de.tsv")
translator.translate("this is a test")
```

[ ]:

```
translator.evaluate(dataset)
```

Classes can wrap very different implementations into a common public interface.

1.5.7 GoogleTranslator class

[ ]:

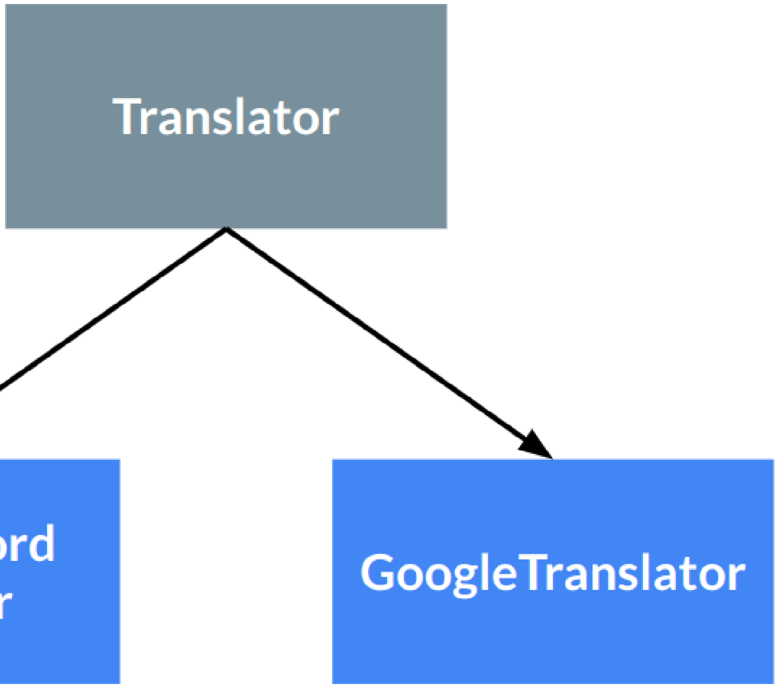
```
import googletrans

class GoogleTranslator:
    def __init__(self, source_lang: str, target_lang: str):
        self.source_lang = source_lang
        self.target_lang = target_lang
        self._translator = googletrans.Translator()

    def translate(self, text: str) -> str:
        translation = self._translator.translate(
            text, src=self.source_lang, dest=self.target_lang
        )
        return translation.text

    def evaluate(self, dataset: ParallelDataset) -> float:
        translations = [self.translate(source) for source, _ in dataset]
        targets = [target for _, target in dataset]
        return sacrebleu.corpus_bleu(translations, [targets]).score
```

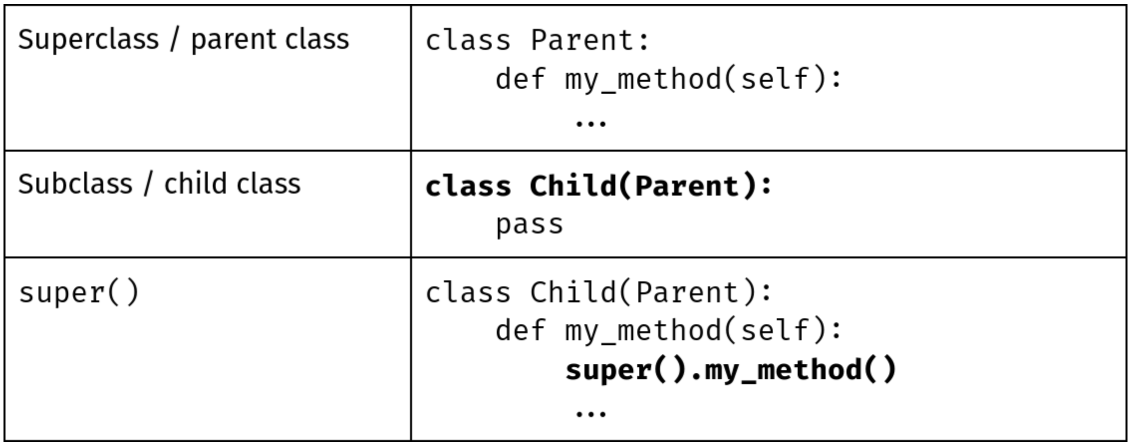
[ ]:

```
translator1 = WordByWordTranslator.load("translation/en-de.tsv")
translator2 = GoogleTranslator("en", "de")

print(translator1.translate("this is a test"))
print(translator2.translate("this is a test"))

print(translator1.evaluate(dataset))
print(translator2.evaluate(dataset))
```

1.5.8 Inheritance

- **Inheritance** allows us to share code and interfaces between similar classes.
- **Child classes** inherit all methods and attributes from their **parent class**.





Superclass / parent class	`class Parent:`
	`    def my_method(self):`
	`        ...`

Subclass / child class	**`class Child(Parent):`**
	`    pass`

super()	`class Child(Parent):`
	`    def my_method(self):`
	`        super().my_method()`
	`        ...`

(from the OOP cheat sheet on OLAT)

Parent class

[ ]:
```
class Translator:
    def __init__(self, source_lang: str, target_lang: str):
        self.source_lang = source_lang
        self.target_lang = target_lang

    def evaluate(self, dataset: ParallelDataset) -> float:
        translations = [self.translate(source) for source, _ in dataset]
        targets = [target for _, target in dataset]
        return sacrebleu.corpus_bleu(translations, [targets]).score
```

Child classes

[ ]:

```
import re

class WordByWordTranslator(Translator):
    def __init__(self, dictionary: dict[str, str], source_lang: str,␣
 ↪target_lang: str):
        super().__init__(source_lang, target_lang)
        self._dictionary = dictionary

    def translate(self, text: str) -> str:
        tokens = text.split()
        translation = " ".join(self._dictionary.get(token, token) for token in␣
 ↪tokens)
        return translation

    @classmethod
    def load(cls, filename: str) -> "WordByWordTranslator":
        source_lang, target_lang = re.search(r"([a-z]+)-([a-z]+).tsv",␣
 ↪filename).groups()
        with open(filename) as f:
            dictionary = dict(line.strip().split("\t") for line in f)
        return cls(dictionary, source_lang, target_lang)
```

[ ]:

```
class GoogleTranslator(Translator):
    def __init__(self, source_lang: str, target_lang: str):
        super().__init__(source_lang, target_lang)
        self._translator = googletrans.Translator()

    def translate(self, text: str) -> str:
        translation = self._translator.translate(
            text, src=self.source_lang, dest=self.target_lang
        )
        return translation.text
```

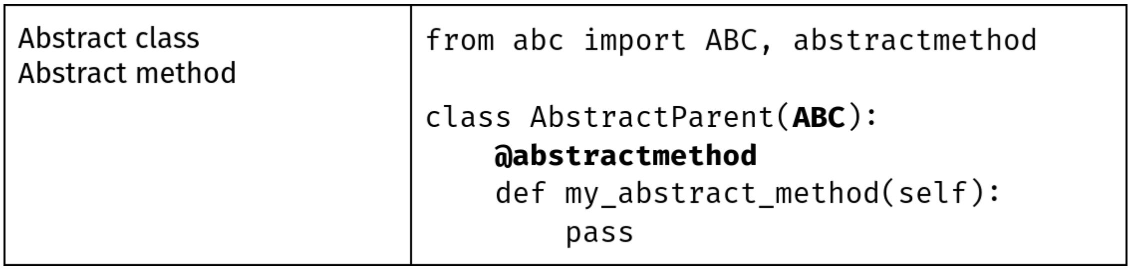
[ ]:

```
translator1 = WordByWordTranslator.load("translation/en-de.tsv")
translator2 = GoogleTranslator("en", "de")

print(translator1.translate("this is a test"))
print(translator2.translate("this is a test"))
```

[ ]:
```python
print(translator1.evaluate(dataset))
print(translator2.evaluate(dataset))
```

1.5.9 Abstract classes

- Prevent the user from instantiating a parent class
- Force all subclasses to implement certain methods



Abstract class
Abstract method
	from abc import ABC, abstractmethod
	
	class AbstractParent(**ABC**):
	    @abstractmethod
	    def my_abstract_method(self):
	        pass

(from the OOP cheat sheet on OLAT)

[ ]:
```python
from abc import ABC, abstractmethod

class Translator(ABC):
    def __init__(self, source_lang: str, target_lang: str):
        self.source_lang = source_lang
        self.target_lang = target_lang

    @abstractmethod
    def translate(self, text: str) -> str:
        pass

    def evaluate(self, dataset: ParallelDataset) -> float:
        translations = [self.translate(source) for source, _ in dataset]
        targets = [target for _, target in dataset]
        return sacrebleu.corpus_bleu(translations, [targets]).score
```

[ ]:
```python
translator = Translator("en", "de")
```

[ ]:
```python
class IncompleteTranslator(Translator):
    pass

translator = IncompleteTranslator("en", "de")
```

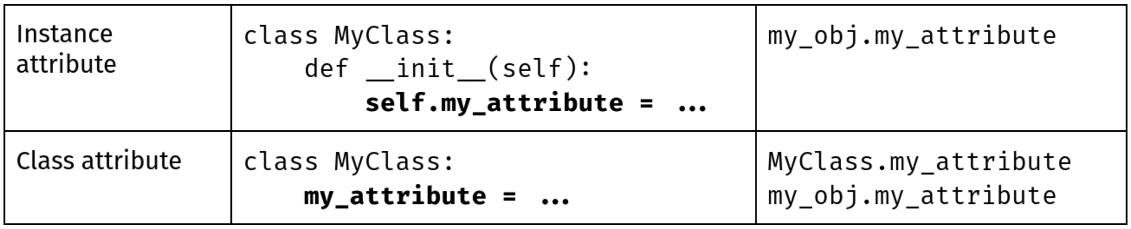
1.5.10 Quiz: inheritance

pwa.klicker.uzh.ch/join/asaeub

1.5.11 Class attributes

- Shared between all instances of a class

- Can be overridden in child classes



Instance attribute	`class MyClass:`
	`    def __init__(self):`
	`        self.my_attribute = ...`
		`my_obj.my_attribute`

Class attribute	`class MyClass:`
	`    my_attribute = ...`
		`MyClass.my_attribute`
		`my_obj.my_attribute`

(from the OOP cheat sheet on OLAT)

Class attributes are often used to store class-specific constants:

[ ]:

```
class Translator(ABC):
    requires_network = False
    ...

class WordByWordTranslator(Translator):
    ...

class GoogleTranslator(Translator):
    requires_network = True
    ...
```

[ ]:

```
WordByWordTranslator.requires_network
```

[ ]:

```
GoogleTranslator.requires_network
```

[ ]:

```
translator = GoogleTranslator()
translator.requires_network
```

1.6 Complete example

- Find the complete implementation on OLAT (`translation_example.zip`).
- Install dependencies using `pip install -r requirements.txt`.
- Next week, we will use this code to create our own Python package.

1.7 A final note on OOP

- OOP can make your code more **readable, maintainable, and extensible**, but it can also make it more **complex and difficult to test**.
- Different programming paradigms lend themselves to different types of problems.
- Use the exercises to practice writing classes and get an intuition for when OOP makes sense and when it doesn't.

1.8 Take-home messages

- **Encapsulation** allows us to hide the implementation details of a class from the user.

- A leading underscore `_` indicates that an attribute or method is **``private''** and should not be accessed from outside the class.
- **Special methods** like `__len__` and `__iter__` allow us to implement built-in Python operators and functionalities for our own classes.
- Different types of methods and attributes:
  - **Instance methods** vs. **class methods** vs. **static methods**
  - **Instance attributes** vs. **class attributes**
- **Inheritance** allows us to share code and interfaces between similar classes.
  - **Child classes** inherit all methods and attributes from their **parent class**.
  - `super()` is used to call the parent class's methods.
  - **Abstract classes** cannot be instantiated and require their child classes to implement certain methods.

Refer to the OOP cheat sheet on OLAT for an overview with examples!

1.9 Exercise 2

This will be the first **graded exercise**!

It will count 20% towards your exercise grade.