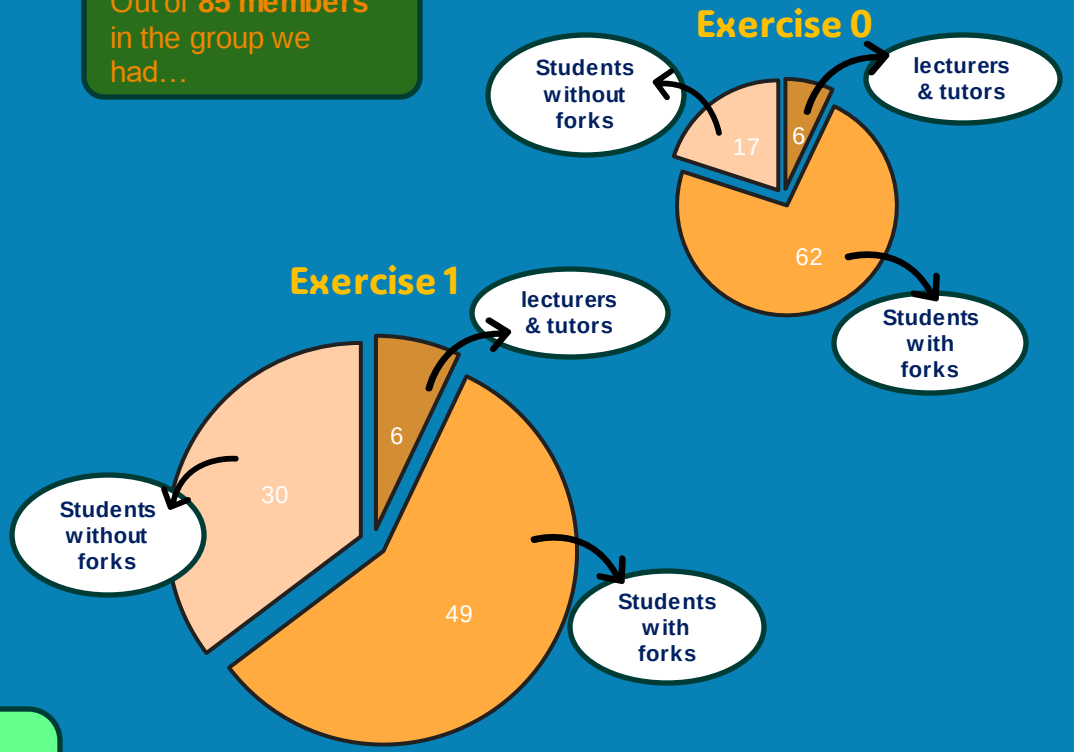

PCL2–Tutorial 03
Lucas Suomela, Rong Li, Tosca Peruzzi-Vieli, Isabelle Cretton

Welcome!

Reminder: Exercises

- Exercise 2 is a **graded** exercise!
- You will receive **1, 0.5 or 0 points**.
- 0 points will be given for: missed deadline, incorrect submission (e.g. no release, wrong access for tutors, wrong exercise submission..)

Out of **85 members** in the group we had…



Most exercises are not graded, and they are not mandatory. We still advise you to do them (even if you don't submit them) to stay on track during the course.

OOP (Object Oriented Programming)

- Object-Oriented Programming (OOP) is a programming paradigm that uses **objects** and **classes** in programming.
- It aims to implement real-world entities like **inheritance**, **polymorphism**, etc. in programming.

recap

Classes:

- Serve as blueprints for creating objects.
- Provide initial values for state (attributes) and implementations of behavior (methods).
- Define the type of objects, categorizing data and functionality.

Objects:

- Are instances of classes, representing a specific implementation.
- Encapsulate variables (attributes) and functions (methods) into self-contained entities.
- Central to Python programming, allowing for data manipulation and interaction.

Inheritance

Inheritance is a mechanism where a new class (subclass) can inherit attributes and methods from an existing class (superclass), promoting code reuse and establishing a hierarchical organization of classes.

What's so great about inheritance?

Code Reusability: Inheritance enables subclasses to use superclass code, reducing duplication and easing maintenance.

Hierarchical Classification: Inheritance forms a class hierarchy, e.g., a **TextProcessor** class with **SpellChecker**, **GrammarChecker**, and **SentimentAnalyzer** subclasses.

Extendibility and Customization: Inheritance allows for easy creation and modification of new classes, enhancing functionality with little change to existing code.

Overriding Methods: Subclasses can modify superclass methods to provide specific implementations while keeping a uniform interface.

```python
# Base class
class TextProcessor:
    def __init__(self, text):
        self.text = text

    def process(self):
        return self.text.lower()

# Subclass inheriting from TextProcessor
class WordCounter(TextProcessor):
    def process(self):
        # Overriding the process method
        return len(self.text.split())

# Another subclass
class Capitalizer(TextProcessor):
    def process(self):
        # Different implementation of the process method
        return self.text.upper()
```

Polymorphism

```python
# Base class
class Shape:
    def area(self):
        raise NotImplementedError("This  method  should  be  overridden  by  subclasses")

# Derived class for circle
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return  math.pi  *  self.radius  **  2

# Derived class for rectangle
class Rectangle(Shape):
    def __init__(self, width,  height):
        self.width  =  width
        self.height  =  height

    def area(self):
        return  self.width  *  self.height
```

(Very explicit example)

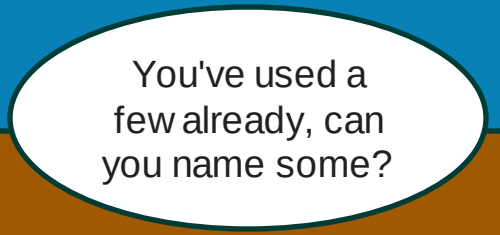
Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables the same interface to be used for different implementations, enhancing flexibility and allowing one interface to support multiple actions.

Polymorphism in NLP classes


```python
# Base class for text analysis
class TextAnalysis:
    def __init__(self, text):
        self.text = text

    def analyze(self):
        raise NotImplementedError("Subclass must implement abstract method")

# Subclass for tokenization
class Tokenizer(TextAnalysis):
    def analyze(self):
        # Simple whitespace tokenization
        return self.text.split()

# Subclass for part-of-speech tagging
class POSTagger(TextAnalysis):
    def analyze(self):
        # Mock implementation for demonstration
        # Normally you would use a library like NLTK or SpaCy here
        return [("This", "DT"), ("is", "VBZ"), ("a", "DT"), ("test", "NN")]

# Subclass for named entity recognition
class NER(TextAnalysis):
    def analyze(self):
        # Mock implementation for demonstration
        return [("This", "O"), ("is", "O"), ("Berlin", "B-LOC")]

# Using the classes in a polymorphic way
texts = ["This is a test sentence.", "Analyze this text for entities."]
strategies = [Tokenizer(text), POSTagger(text), NER(text) for text in texts]

for strategy in strategies:
    print(f"Analysis using {strategy.__class__.__name__}: {strategy.analyze()}")
```

OOP for NLP

In NLP projects, you often deal with complex data structures and algorithms - like language models, tokenizers, and text classifiers. OOP allows for these complex entities to be encapsulated in classes and objects, making the code more modular, reusable, and understandable.



SpaCy Classes

Some notable ones...

nlp (Language Class):

The **Language** class is the main entry point to SpaCy's NLP pipeline. You typically start by loading a language model into an **nlp** object.

Example: **nlp = spacy.load('en_core_web_sm')**

Doc Class:

The **Doc** class is one of the central data structures in SpaCy. It is created by passing a string of text to the **nlp** object. The **Doc** object then contains a sequence of **Token** objects.

Example: **doc = nlp("This is a sentence.")**

Matcher and PhraseMatcher Classes:

These classes are used for rule-based matching of words or phrases within the document.

Example: **matcher = Matcher(nlp.vocab)**

Token Class:

Represents a single word, punctuation symbol, whitespace, etc., in a document. Each **Doc** consists of a sequence of **Token** instances.

Example: **for token in doc: print(token.text)**

NLTK Classes

Some notable ones...

WordNetLemmatizer Class:

Used for lemmatizing words. Lemmatization is the process of grouping together the different inflected forms of a word so they can be analyzed as a single item.

Example: **from nltk.stem import WordNetLemmatizer; lemmatizer = WordNetLemmatizer()**

nltk.corpus Classes:

Provide access to a variety of linguistic corpora, such as **WordListCorpusReader**, **PlaintextCorpusReader**, etc.

Example: **from nltk.corpus import brown; brown.words(categories='news')**

PorterStemmer and LancasterStemmer Classes:

These classes are used for stemming, which involves cutting off the ends of words in the hope of achieving the goal most of the time of correctly identifying the root word.

Example: **from nltk.stem import PorterStemmer; stemmer = PorterStemmer()**

This is just to give you some real-world examples

Exercises

Solve these in your IDE, by yourselves or with your neighbour!
we'll solve them together afterwards....

Exercise 1

Exercise 1: Basic Class Implementation

Task: Create a basic Python class called **BasicTextProcessor**. The class should have:

- An **__init__** method that accepts a string of text and stores it as an attribute.
- A method **lowercase_text** that returns the text in lowercase.
- A method **word_count** that returns the number of words in the text (assume words are separated by spaces).

Exercise 2

Exercise 2: Inheritance and Method Overriding

Task: Extend the **BasicTextProcessor** class by creating a new class called **AdvancedTextProcessor**.

This new class should:

- Inherit from **BasicTextProcessor**.
- Override the **word_count** method to ignore common stopwords (e.g., "the", "is", "at", "which", "on"). You may define a list of stopwords within the class.
- Add a new method **unique_words** that returns the number of unique words in the text.

Objective: Practice using inheritance, understand how to override parent class methods, and introduce new methods in the child class.

Exercise 3

Exercise 3: Abstract Base Classes and Polymorphism

Task: Define an abstract base class called **TextAnalyzer** with an abstract method **analyze**. Then, create three concrete classes that inherit from **TextAnalyzer**:

- **TokenCounter**: Implements the **analyze** method to return the total number of words (tokens) in the text.
- **VowelCounter**: Implements the **analyze** method to return the total number of vowels in the text.
- **UpperCaseConverter**: Implements the **analyze** method to convert all text to uppercase.

Finally, write a function **process_texts** that takes a list of texts and an instance of **TextAnalyzer**, and returns a list of analysis results for each text.

Objective: Learn how to work with abstract base classes, implement polymorphism through method overriding, and understand how to use classes in a flexible and interchangeable manner.

Now It's Your Turn...

To-Do

- ❑ Work on the exercise
- ❑ Issues with GitLab? Please let us know.
- ~~❑ Have a nice dinner date with chatGPT~~
- ❑ Enjoy your weekend!