
PCL2–Tutorial 07
Lucas Suomela, Rong Li, Tosca Peruzzi-Vieli, Isabelle Cretton

# Welcome!

# Text Encoding

- Encoding text is translating human-readable text into a machine-readable format.
- Characters can be represented as code points and in binary.
- Early text encodings used 1 byte, or 8 bits per character, allowing for $2^8 = 256$ code points
- The different text encodings determine how those code points are mapped to characters

**Examples**

- ASCII
- ISO-8859-1 (Latin 1)
- ISO-8859-5
- ISO-8859-6
- Windows-1252 (CP-1252)
  → Most common issue when writing and reading files in Windows.

# Unicode

- Problem with prior encodings: very limited number of encodable characters, and multiple writing systems within a file or a string was impossible
- Solution: The Unicode Standard, with the goal to represent any writing system. Currently contains code point mappings for 149'813 characters, and supports 161 writing systems/scripts (Unicode 15.1)
- Will most likely be the standard for a long time, as it can support up to 1'111'998 characters
- Defines three text encodings: UTF-8, UTF-16, UTF-32
  Other encodings based on Unicode exist as well but are uncommon

# unicodedata

```
# Getting the character from its name
char_from_name = unicodedata.lookup('LATIN SMALL LETTER E WITH ACUTE')
print(f"The character for the Unicode name 'LATIN SMALL LETTER E WITH ACUTE'
is '{char_from_name}'.")

# Finding the category of a character
category = unicodedata.category(char)
print(f"The category of '{char}' is {category} (Letter, lowercase).")

# Normalizing an emoji
combined_emoji = '👨‍👩‍👧'  # Family: Man, Woman, Girl
normalized_emoji = unicodedata.normalize('NFC', combined_emoji)
print(f"The normalized form of the combined emoji '{combined_emoji}' is '
{normalized_emoji}'.")
```

***Emoji Handling***: *Many emojis are combinations of multiple Unicode characters. The family emoji (👨‍👩‍👧) is one such example, made by combining multiple human emojis with a zero-width joiner. Normalizing these is useful for ensuring that complex emojis are handled consistently and for displaying the separate characters they are made with.*

# Opening Files in Python

**Best Practices:**

- **Always use 'with open()' when opening files! If you open a file without this statement and assign the content to a variable directly, you must manually close it, otherwise it stays in memory**
- **Be mindful of the file path, think of how the script is run and whether you should then use relative or absolute file paths.**
- **It's not a must, but it's good practice to handle potential errors that might occur during file operations, such as using `try...except` blocks**

**(→ See the demo script on OLAT)**

# Text-based Data Formats: Overview

**CSV:** Comma Separated Values, also called tabular data (Table). CSV is **not standardized** and has dialects. Usually not preferred when sharing and storing data.
**JSON:** JavaScript Object Notation, arbitrary nested data structures, mostly identical to dictionaries in Python. One of the most common data structures used for web services and APIs.
**XML:** Extensible Markup Language, arbitrary nested data. It was the most dominant choice for data exchange before JSON.

**XML**

XML allows for custom tags (definitions), schemes and self-validation making it the preferred choice for many NLP tasks.
It does require more processing power compared to JSON, so considerations in regard to efficiency need to be made when parsing large files.

# Text-based Data Formats: Libraries

**Built-in**

- Python has a built-in encoder and decoder library for JSON and a reader/writer for CSV files, called `json` and `csv` respectively
- Python also has a built-in library for parsing XML files, `xml`, which offers limited functionality compared to external libraries and is vulnerable to maliciously structured data.

**External**

- A better choice for parsing XML files is `lxml`. Like the standard library, it utilizes the **ElementTree API**.
- You can also use `pandas` to read and write to CSV and JSON files with `pandas.read_csv()` and `pandas.read_json()`

# Midterm Exam Info (See here)

**When:** Next Wednesday (24.04) from 10:15-11:00 (45 mins)
**Where:** AND-3-02/06
**How:** paper and non-erasable pen

**NB:**
- Bring your student ID card.
- **No** cheat sheets or laptop allowed.
- Please be on time! You will **not** get additional time if you are late.
- The second half of the lecture will take place as usual.

**The exam will involve:**
- Understanding and applying theoretical foundations
- Explaining terms and concepts
- Understanding and analyzing code snippets (including the standard library, as far as covered in the lecture)

**The exam will not involve:**
- Writing code
- Memorizing function signatures
- Knowledge about external libraries (spaCy, nltk, etc.)

**Question types:**
- Multiple-choice
- Cloze/fill-in-the-blank texts
- Short text answers

# Midterm Exam FAQ

Q: Can you upload previous years' exam questions on OLAT?

A: This year is the first time we have mid-term exam □ !

Q: Do we need to memorize a lot of theory, or only understand concepts? Or only solve / deploy the things practically?

A: Focus on theoretical terms and concepts.

- Understanding 👉 Multiple-choice
- Explaining 👉 Short text answers
- Applying 👉 Analyzing code snippets, but no codes writing

Q: Do we need to know the structures of python packages (e.g. syntax of .toml file etc.)?

A: The idea of the exam isn't to memorize every detail :)

Q: For which topics will there be questions that have more points than other topics?

A: The points should be evenly distributed across each lecture 🤔.

Q: Can we get some practice exam-type questions?

A: Yes! The sample solution will later be uploaded on OLAT.

# Computational Complexity

- A theoretical concept
- Determine how quickly runtime **increases** with increasing input length based on **inherent characteristics** of the program.
- Tells us how **scalable** our algorithms are (e.g., with increasing corpus size, document length, vocabulary size, etc.)

**Time complexity**
**Given an algorithm, how quickly does the number of operations grow when we increase the input length?**

**Space complexity**
**How complex is our program in terms of the memory it takes to run?**

**Big O notation:**
- Keep only highest-order terms, ignore constant factors ("overall impact" assumption)
- we only look at the factor in our expression that has the potential greatest impact on the value that our expression will return.
- Always assumes the worst case scenario
- $O(1) < O(n) < O(n\log n) < O(n^2) < O(2^n)$

See more examples here

| Regular | Big-O | |
|---|---|---|
| 2 | O(1) | --> It's just a constant number |
| 2n + 10 | O(n) | --> n has the largest effect |
| 5n^2 | O(n^2) | --> n^2 has the largest effect |

# O(1)

**Example**: Accessing an element in a list by index.

```
def get_element(list, index):
    return list[index]

# This operation does not depend on the size of the list; it's a single, quick check.
```

An algorithm is said to run in constant time if it takes the same amount of time to process regardless of the input size.

O(1) – Constant Time Complexity

# O(n)

**Example**: Summing all elements in a list.

```
def sum_lst(list):
    total = 0
    for element in list:
        total += element
    return total
```

**Example:** Finding the maximum element in an unsorted list:

```
def find_max(lst):
    max_val = lst[0]
    for num in lst:
        if num > max_val:  # Compare each element with the current max
            max_val = num
    return max_val
```

We need to look at each element once to add up the elements / determine the maximum, so the runtime grows linearly with n, the number of elements in the list.

O(n) – Linear Time Complexity

An algorithm has linear time complexity if the runtime is directly proportional to the size of the input.

# O(n log n)

**Example:** Finding the $k$ longest strings

```
def longest_naive(strings, k=3):
    return sorted(strings, key=len)[-k:]
```

Sorted() takes O(n log n) time.
It is the Timsort, and Timsort is a kind of adaptive sorting algorithm based on merge sort and insertion sort. If you are interested, see here

Algorithms that are more efficient than $O(n^2)$ but less efficient than O(n). This is often seen in efficient sorting algorithms.

O(n log n) – Log-Linear Time Complexity

# $O(n^2)$

**Example:** Checking for duplicates in a list

```
def has_duplicates(lst):
    for i in range(len(lst)):   # Iterate through each element
        for j in range(i + 1, len(lst)):
# For each element, iterate through subsequent elements
            if lst[i] == lst[j]:
                return True
    return False
```

For each of the n elements, we're potentially comparing it to all other n-1 elements (in the worst case), leading to $O(n^2)$.

An algorithm has quadratic time complexity when the time is proportional to the square of the input size.

$O(n^2)$ – Quadratic Time Complexity

# $O(2^n)$

**Example:** Computing the nth Fibonacci Number (With a Recursive Approach)
The Fibonacci sequence is a series where each number is the sum of the two preceding ones.

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

For n > 1, the function recursively calls itself to calculate the two preceding numbers in the sequence. This means that to calculate fibonacci(n), you first need to calculate fibonacci(n-1) and fibonacci(n-2).
Each function call spawns two more calls until you reach the base cases (n is 0 or 1). Thus, there are n levels, and since each level doubles the number of calls, there are roughly $2^n$ calls in total.

Algorithms with exponential time complexity have runtimes that double with each additional input bit.

$O(2^n)$ – Exponential Time Complexity

# OOP – Instance Method and Abstract Method

**Instance methods** are the basic type of methods in Python classes. They can modify object instance state and operate on the data of the objects to which they belong. An instance method takes self as the first parameter. Decorator is **not required** for instance methods.

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

    def greet(self):
        return f"Hello, my name is {self.name}!"

# Use case
john = Person("John")
print(john.greet())
# Output: Hello, my name is John!
```

**Abstract methods** are declared in a class, but they must be implemented by **subclasses**. They are used in abstract base classes which **cannot be instantiated** themselves and are designed to be extended. An abstract class can be considered as a **blueprint** for other classes.

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def drive(self):
        pass

class Car(Vehicle):
    def drive(self):
        return "Driving a car"

# Use case
my_car = Car()
print(my_car.drive())  # Output: Driving a car
```

Abstract methods must be overridden to create concrete implementations in derived classes.

# OOP – Static vs. Class Method

A **static method** does not receive an implicit first argument (self for instance or cls for class). A static method is also a method that **is bound to the class** and not the instance of the class. They are used when some functionality is related to the class but does not need to **access or modify** the class's state.

```
class Geometry:
    @staticmethod
    def area_of_triangle(base, height):
        return 0.5 * base * height

# Use case
print(Geometry.area_of_triangle(10, 20))
```

A **class method** is bound to the class and not the instance of the class. They can modify the class state that applies across all instances of the class, via the cls argument, which is a reference to the class itself.

```
class Counter:
    count = 0

    @classmethod
    def increment(cls):
        cls.count += 1
        return cls.count

# Use case
Counter.increment()  # count is now 1
Counter.increment()  # count is now 2
```

**Class method vs Static Method**

- A class method takes cls as the first parameter while a static method needs no specific parameters.
- A class method can access or modify the class state while a static method can't access or modify it.
- We use @classmethod decorator to create a class method and we use @staticmethod decorator to create a static method.

# Functional Programming – Iterable vs. Iterator

An **iterable** is an object which can be looped over or iterated over in a loop. Examples of iterables include **lists, sets, tuples, dictionaries, strings**, etc. Any object that has an __iter__() method which returns an **iterator**, or a __getitem__() method that can take sequential indexes, is an iterable.

An **iterator** is an object that allows you to iterate over collections of data, consisting of the methods __iter__() and __next__(). The __iter__() returns the iterator object itself and is used once; the __next__() method returns the next value from the iterator. When there are no more items, __next__() raises a **StopIteration** exception.

```
my_list = [1, 2, 3, 4] # a list is an iterable
my_iterator = iter(my_list)  # Or my_list.__iter__()

while True:
    try:
        item = next(my_iterator)  # Or my_iterator.__next__()
        print(item)
    except StopIteration:
        break
```

# Functional Programming – Generator

**Generators** are a type of **iterator**, but they are written as regular functions and use the yield statement to produce a series of values lazily, meaning they generate values on the fly and only when needed. This makes them memory-efficient, especially useful for large datasets.

**Generator expressions** provide a concise way to create generators. They create a generator that yields items on-the-fly, which makes them highly memory-efficient, especially useful for large datasets.

Syntax: similar to list comprehension except it uses parentheses ():
(expression **for** item **in** iterable **if** condition)

**Let's see an example of this generator function:**

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

counter = count_up_to(5)
for num in counter:
    print(num)
```

**Another example about generator expressions:**

```
infile = open("very_large_file.txt")
 generator = (line.strip().upper() for line in infile if line != "\n")
next(generator)
```

# Functional Programming – Decorator

**Decorators** are **higher-order functions** used to modify the behavior of a function. It allows you to add new functionality to an existing object without modifying its structure. Decorators are usually called **before** the definition of a function you want to decorate.

A **higher order function** is a function that takes a function as an argument or that returns a function.

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

my_decorator is a function that takes another function (say_hello) and extends its behavior without explicitly modifying say_hello's definition.

# Functional Programming – Pure Functions

A **pure function** is a function that satisfies two main conditions:

- **Deterministic Output:** For the same input values, always returns the same result.
- **No Side Effects:** Does not alter any external state or data outside of its scope. Does not print anything.

**Why do we need pure functions?**

- Since they always return the same result for the same inputs, they are straightforward to test.
- Their isolated nature makes them easily reusable and predictable.

```
number = 0

class Counter:
    def __init__(self):
        self.count = 0

    def increment(self):
        global number
        number += 1
        return number

    def add(self, x, y):
        return x + y

# Use case
counter = Counter()
print(counter.increment())  # Output: 1
print(counter.increment())  # Output: 2

print(counter.add(5, 3))  # Output: 8
print(counter.add(5, 3))  # Output: 8
```

The increment() method is impure because:

- It modifies a global variable, affecting external state.
- Repeated calls with the same external inputs (no inputs in this case) produce different results.

The add() method is pure because:

- It always returns the same result for the same inputs.
- It does not modify any internal or external state of the class.

# Now It's Your Turn...

**To-Do**

- ❑ Prepare for the Midterm Exam
- ❑ Ask us questions if you are confused!
- ❑ Learn how to speak in binary