# Lecture IV - Handling Data in more than one Dimension

Programming with Python

Dr. Tobias Vlćek

# Quick Recap of the last Lecture

## Functions

- Functions are **reusable blocks** of code that perform specific tasks
- They can accept inputs (parameters) and return outputs
- `def` followed by the function name, parameters and a colon
- Help in organizing code and reducing repetition

. . .

```
def greet(name):
    return f"Welcome to this lecture, {name}!"

print(greet("Students"))
```

```
Welcome to this lecture, Students!
```

## Scope

- Scope determines the visibility and lifetime of variables
- Variables defined inside a function are **local** to that function
- Variables defined outside of all functions are **global**
- They can be accessed from anywhere in the program

. . .

```
def greet(name):
    greeting = f"Welcome to this lecture, {name}!"
    return greeting

print(greeting) # This will cause an error
```

>Question: Why does this cause an error?

## Classes

- Classes are **blueprints** for creating objects
- They encapsulate data (attributes) and behavior (methods)
- Help in organizing code and creating objects with similar structures

. . .

```python
class Lectures:
    def __init__(self, name, length_minutes):
        self.name = name
        self.length = length_minutes

    def duration(self):
        return f"Lecture '{self.name}' is {self.length} minutes long!"

lecture_4 = Lectures("4. Data in more than one dimension", 90)
print(lecture_4.duration())
```

```
Lecture '4. Data in more than one dimension' is 90 minutes long!
```

# Tuples

## What are Tuples?

- Tuples are **ordered collections** of items
- They are **immutable** (cannot be changed after creation)
- Help in storing multiple items in a single variable
- Created using the `tuple()` function or the `()` syntax

. . .

```
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple)
```

```
(1, 2, 3, 4, 5)
```

## Tuple Operations

- Tuples support the same operations as strings
- We can use indexing and slicing to access elements
- We can use the + operator to concatenate tuples
- We can use the * operator to repeat a tuple

. . .

>Question: What will the following code print?

```
my_tuple = (1, 2, 3)
print(my_tuple[1:3])
print(my_tuple + (4, 5, 6))
print(my_tuple * 2)
```

```
(2, 3)
(1, 2, 3, 4, 5, 6)
(1, 2, 3, 1, 2, 3)
```

## Tuple Methods

- Tuples support the following methods:
  - `count(x)`: Returns the number of times `x` appears in the tuple
  - `index(x)`: Returns the index of the first occurrence of `x`

. . .

>Question: What will this code print?

```
my_tuple = (1, 2, 3, 2, 4, 2)
print(my_tuple.count(2))
print(my_tuple.index(3))
```

```
3
2
```

## Tuple Data Types

- Tuples can contain elements of different data types

```
my_tuple = ("Peter", 25, "Hamburg")
print(my_tuple)
```

```
('Peter', 25, 'Hamburg')
```

...

```
# This works as well
my_tuple = "Peter", 25, "Hamburg"
print(my_tuple)
```

```
('Peter', 25, 'Hamburg')
```

...

> **Note**
>
> We can also create tuples by listing the elements separated by commas.

## Tuples from Functions

- Functions can return tuples
- This is useful if we want to **return multiple values** from a function

...

```
def get_student_info(name, age, city):
    return name, age, city

student_info = get_student_info("Peter", 25, "Hamburg")
print(student_info)
```

```
('Peter', 25, 'Hamburg')
```

...

>Question: How would you access the age from the tuple?

## Tuple Unpacking

- Allows us to assign the elements of a tuple to variables
- The number of variables **must match** the number of elements
- Use the $*$ operator to assign the remaining elements to a variable

...

```python
def get_student_info(name, age, city):
    return name, age, city
name, *rest = get_student_info("Peter", 25, "Hamburg")
print(f"Name: {name}")
print(f"Other info: {rest}")
```

```
Name: Peter
Other info: [25, 'Hamburg']
```

...

> ⚠ Warning
>
> The output is positional, so we have to be careful with the order of the variables.

# Lists

## What are Lists?

- Lists are **ordered collections** of items
- They are **mutable** (can be changed after creation)
- Created using the `list()` function or the `[]` syntax
- They support the **same operations** as strings and tuples
- Have much more methods and are more versatile than tuples

. . .

```
my_list = [1, 2, 3, 4, 5]
print(my_list)
```

```
[1, 2, 3, 4, 5]
```

. . .

>Question: Any idea why lists support more methods?

## List Methods

- Common methods for lists:
  - `count(x)`: Returns the number of times `x` appears in the list
  - `append(x)`: Adds an element `x` to the end of the list
  - `insert(i, x)`: Inserts an element `x` at index `i`
  - `remove(x)`: Removes the first occurrence of element `x`
  - `index(x)`: Returns the index of the first occurrence of `x`

  - `pop([i])`: Removes the element at index `i` and returns it
  - `sort()`: Sorts the list in ascending order
  - `reverse()`: Reverses the list

## Lists in Action

>Task: Solve the following problem using lists:

```
# Imagine the following shoppping list for this weekend
shopping_list = ["cider", "beer", "bread", "frozen_pizza"]
```

. . .

- First, add some apples to the list for a healthy option

- Next, remove the cider as you already have some at home
- Sort all items in the list alphabetically
- Print each item of the list on a new line

. . .

> **Tip**
>
> You can use the methods and loops we learned so far to solve the problem.

# Sets

## What are Sets?

- Sets are **unordered collections** of unique elements
- They are **mutable** (can be changed after creation)
- Created using the `set()` function or the `{}` syntax
- Supports `+` and `*` operations like lists and tuples
- Unlike lists and tuples, **sets do not support indexing**

. . .

```
my_set = {1, 2, 2, 5, 5}
print(my_set)
```

```
{1, 2, 5}
```

## Set Methods

- Common methods for sets:
  - `add(x)`: Adds an element `x` to the set
  - `remove(x)`: Removes an element `x` from the set
  - `discard(x)`: Removes an element `x` from the set if it is present
  - `pop()`: Removes and returns an arbitrary element from the set
  - `update(other)`: Adds all elements from `other` to the set

## Set Theory

- Additional methods are derived from set theory
  - `union(other)`: New set with elements from both sets
  - `intersection(other)`: New set with common elements
  - `isdisjoint(other)`: `True` if no elements in common
  - `issubset(other)`: `True` if subset of `other`

. . .

> Tip
>
> There are more methods for sets! If you are working intensively with sets, keep that in mind.

## Sets in Action

>Task: Solve the following problem using sets:

```
# You have a list of friends from two different groups
friends_group_1 = ["Neo", "Morpheus", "Trinity", "Cypher"]
friends_group_2 = [ "Smith", "Apoc", "Cypher", "Morpheus"]
```

. . .

- First, find the mutual friends in both groups
- Then create a new set of all friends from both groups
- Count the number of friends in total
- Print each item of the set on a new line

. . .

> Tip
>
> Notice, there is a small error in the given code that you have to fix.

# Dictionaries

## What are Dictionaries?

- Dictionaries are **unordered collections** of key-value pairs
- They are **mutable** (can be changed after creation)
- Keys must be **unique** and **immutable**
- Values can be of any type
- Created using the `dict()` function or the `{}` syntax
- As sets we **cannot access them by index**

. . .

```
who_am_i = {"name": "Tobias", "age": 30, "city": "Hamburg"}
print(who_am_i)
```

```
{'name': 'Tobias', 'age': 30, 'city': 'Hamburg'}
```

## Key-Value Pairs

- We can access them by their keys, though!
- You can think of them as a set of key-value pairs

. . .

```
who_am_i = {"name": "Tobias", "age": 30, "city": "Hamburg"}
print(who_am_i["name"])
```

```
Tobias
```

. . .

> **Note**
>
> Note, how we can use the `[]` operator to access the value of a key?

## Dictionary Operations

- Common operations and methods:
- `in` operation to check if a key is in the dictionary
- `for` loop to iterate over the dictionary
- `keys()` method to return a view of the dictionary's keys

- `values()` method to return a view of the dictionary's values
- `pop(key[, default])` to remove a key and return its value

## Dictionaries in Action

>Task: Solve the following problem using dictionaries:

```
# Create a dictionary with the following information about yourself: name, age, city
i_am = {}
```

...

- Add your favorite color and food to the dictionary
- Remove the city from the dictionary
- Print your name and age in a formatted sentence

# Overview of new Data Types

## Comparison between Data Types

- **Tuple**: Immutable, ordered, duplicates allowed
- **List**: Mutable, ordered, duplicates allowed
- **Set**: Mutable, unordered, no duplicates
- **Dictionary**: Mutable, unordered, no duplicates, key-value pairs

. . .

> Tip
>
> This impacts your code, the operations you can perform and the speed of your program. Thus, it makes sense to understand the differences and choose the right data type for the task.

## When to use which?

- **Tuples**: store a collection of items that should not be changed
- **Lists**: store a collection of items that should be changed
- **Sets**: store a collection of items that should not be changed and duplicates are not allowed
- **Dictionaries**: store a collection of items that should be changed, duplicates are not allowed and require key-value pairs

. . .

> Tip
>
> You can convert between the data types using `tuple()`, `list()`, `set()` and `dict()`. Note, that this is not always possible, e.g. you cannot convert a list to a dictionary without specifying a key.

## Speed Differences

- **Lists** are the most versatile, but slowest
- **Tuples** are generally faster than lists
- **Sets** are generally faster than lists and tuples
- **Dictionaries** depend, but are generally faster than lists and tuples

```
import timeit

# Number of elements in each data structure
```

```python
n = 10000000

# Setup for each data structure, including the test function
setup_template = """
def test_membership(data_structure, element):
    return element in data_structure
data = {data_structure}
"""

setups = {
    'Tuple': setup_template.format(data_structure=f"tuple(range({n}))"),
    'List': setup_template.format(data_structure=f"list(range({n}))"),
    'Set': setup_template.format(data_structure=f"set(range({n}))"),
    'Dictionary': setup_template.format(data_structure=f"{{i: i for i in range({n})}}")
}

# Measure time for each data structure
print(f"Time taken for a single membership test in {n} elements (in seconds):")
print("-" * 75)
for name, setup in setups.items():
    stmt = f"test_membership(data, {n-1})"  # Test membership of the last element
    time_taken = timeit.timeit(stmt, setup=setup, number=1)
    print(f"{name:<10}: {time_taken:.8f}")
print("-" * 75)
print("Note, that theses values are machine dependent and just for illustration!")
```

```
Time taken for a single membership test in 10000000 elements (in seconds):
---------------------------------------------------------------------------
Tuple     : 0.04548883
List      : 0.05264475
Set       : 0.00000233
Dictionary: 0.00000408
---------------------------------------------------------------------------
Note, that theses values are machine dependent and just for illustration!
```

## Comprehensions

- Comprehensions provide a concise way to create data structures
  - **Tuple** comprehensions: `(x for x in iterable)`
  - **List** comprehensions: `[x for x in iterable]`
  - **Set** comprehensions: `{x for x in iterable}`
  - **Dictionary** comprehensions: `{x: y for x, y in iterable}`

...

> **Tip**
>
> The iterable can be **any object that can be iterated over**, e.g. a list, tuple, set, dictionary, etc.

## Iterables

- We have already introduced those!
- We can use the `for` loop to iterate over an iterable

...

```
shopping_list = ["cider", "beer", "bread", "frozen_pizza"]
for item in shopping_list:
    print(item)
```

```
cider
beer
bread
frozen_pizza
```

...

```
who_am_i = {"name": "Tobias", "age": 30, "city": "Hamburg"}
for key, value in who_am_i.items():
    print(f"{key}: {value}")
```

```
name: Tobias
age: 30
city: Hamburg
```

## Nesting

- We can **nest data structures** within each other
- This is useful if we want to store more complex data

...

```
normal_list = [1, 2, 3, 4, 5]
nested_list = ["Hello, World!", normal_list, (1,2)]

print(nested_list)
print(nested_list[2])
```

```
['Hello, World!', [1, 2, 3, 4, 5], (1, 2)]
(1, 2)
```

...

> **Tip**
>
> You can also nest lists within lists within lists, etc.

# I/O

## Input/Output

- A common task in programming is to interact with users
- Remember the `input()` function from the first lecture?
- It is a classical example of **user input**
- An example of **output** is the `print()` function

. . .

```
name = input("Please enter your name: ")
print(f"Hello, {name}!")
```

. . .

> **Note**
>
> Thus, we have already worked with I/O in Python!

## Reading and Writing Files

- We also need to interact with data
- File handling in Python is quite simple:
  - Use `open(file_name, mode)` to open a file
  - Modes: `"r"` (read), `"w"` (write), `"a"` (append)
- Basic operations:
  - Read: `file.read()`
  - Write: `file.write(content)`
  - Close: `file.close()`

## File Handling in Action

```
file = open("hi.txt", "w") # This creates a file called "hi.txt"
file.write("Hello, World!") # This writes "Hello, World!" to the file
file.close() # This closes the file
print("File successfully written")
```

```
File successfully written
```

. . .

>Question: Any ideas how to read the file?

...

```
file = open("hi.txt", "r") # This opens the file "hi.txt"
content = file.read() # This reads the content of the file
file.close() # This closes the file
print(content) # This prints the content of the file
```

```
Hello, World!
```

...

> Tip
>
> Close files with `file.close()` to free up system resources and ensure data is properly saved.

## Easier File Handling with with

- We can also use the `with` statement to open a file
- This ensures the file is properly closed after its handling finishes
- It's a good practice to use it when working with files

```
with open("hi_again.txt", "w") as file:
    file.write("Hello again, World!")

print("File successfully written")
```

```
File successfully written
```

...

>Task: Open the file `hi_again.txt` and print its content using `with`

## Working with other file types

- Naturally, we also want to work with other file types!
- Reading and writing **CSV files** is a common tasks in data analysis
- Excel files are used in many applications and companies
- We will see how to do this **later** in the course

...

> Note
>
> **And that's it for todays lecture!**
> We now have covered the basics of tuples, sets, lists and dictionaries as well as some basic file handling. For now, just remember that advanced reading and writing is possible and that there are libraries that help with this.

Literature {.title}

## Interesting Books

- Downey, A. B. (2024). Think Python: How to think like a computer scientist (Third edition). O'Reilly. Link to free online version
- Elter, S. (2021). Schrödinger programmiert Python: Das etwas andere Fachbuch (1. Auflage). Rheinwerk Verlag.

. . .

> **Tip**
>
> Nothing new here, but these are still great books!

. . .

For more interesting literature to learn more about Python, take a look at the literature list of this course.