Introduction to Jupyter

1. What is Jupyter?

Project jupyter is a collection of open-source web applications (Jupyter Notebook, Jupyter Lab and Jupyter Hub) that allow to create and share documents that contain live code, text, equations, visualizations, images, hypterlinks, and code output. Jupyter supports various programming languages, including Python, R, and Julia, making it an excellent tool for data science, statistical modeling, machine learning, and more.

Jupyter utilized the **IPYNB** ("Interactive Python Notebook") file format. IPYNB files are structured using JSON (JavaScript Object Notation) and contain metadata at the root level, including information such as the Jupyter Notebook version, kernel information, and notebook-specific metadata like the notebook name, author, and creation date.

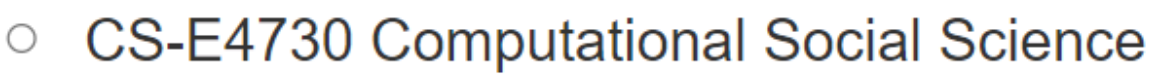
The main content of the notebook is organized into **cells**. Each cell can contain either code, Markdown text, or raw content. Code cells contain executable code snippets. When executed, code cells produce output that can be displayed directly below the cell.

Markdown cells contain formatted text written in Markdown syntax. Markdown cells support various formatting options, including headings, lists, links, images, and more. Markdown cells are rendered as formatted text when the notebook is viewed or exported.

2. How to work on a *.ipynb file

Aalto Jupyter Hub (strongly recommended)

All Aalto students can login into Aalto Jupyter Hub with their Aalto credentials. Then, you can select the course - in this case, **CS-E4730 Computational Social Science (2024)**:



- and click on "Start". This will start a Jupyter environment with all the correct python libraries you will need in the course. Here you can upload any *.ipynb file by clicking on the button "Upload". After the* .ipynb file has been uploaded to Aalto Jupyter Hub, you can double click on it to open the notebook.

Another option is installing Jupyter yourself (your own responsibility):

To install Jupyter Notebook on your computer, you can use Python's package manager, `pip` . Open your terminal or command prompt and run:

```
pip install notebook
```

After the installation completes successfully, you still need to take care of installing all the python libraries that you might need to import in the notebook. Note that TAs on this course

will not help troubleshooting your local installation. We recommend you to use Aalto Jupyter Hub.

3. How to submit programming exercises in this course

In CS-E4730 Computational Social Science course, you will solve some programming exercises in Python. You will find the Jupyter notebooks in A+. You can download the notebooks (*.ipynb files) and then upload them in the CS-E4730 Computational Social Science course space in Aalto Jupyter Hub. The Jupyter notebooks already have most of the code needed for the task. You will need to read the notebook and implement the missing parts. After you have implemented all the needed parts, you need to go back to A+ and follow the instructions there.

Sometimes you will be asked to upload your completed Jupyter notebook to A+. If this is the case, A+ will run unit tests to check that you have indeed solved the tasks correctly. In some cases, you will not be asked to submit your solved Jupyter notebook directly, but you will be asked to answer some questions on A+. However, answering those questions requires that you have already solved the programming task!

But, before starting, let's review a bit of the Python basics!

4. Using Python in Jupyter

To demonstrate how to work with Python in a Jupyter Notebook, we start with the famous Hello world:

In [1]:
```
# This line is a comment where I explain that my next line of python code pri
print("Hello world!")
```

```
Hello world!
```

Python data types

In [2]:
```
# next line assigns the integer value 2 to the variable x
x = 2
# print the value of x
print(x)
# print the value of x + 3
print(x + 3)
# print the value of x - 1
print(x - 1)

# if we divide an integer, we get a float
print(x / 3)

# we can also have strings (either inside single quotes ' or double quotes ")
string_one = 'this is a '
string_two = "test string"
# strings can be combined using +
print(string_one + string_two)
```

```
2
5
1
0.6666666666666666
this is a test string
```

In [3]:
```python
# here is how to check the type of a variable
print(type(x))
```

```
<class 'int'>
```

In [4]:
```python
# as I told you, if we divide an integer, we get a float
print(type(x / 3))
```

```
<class 'float'>
```

Lists, tuples, and dictionaries

A list is a mutable (changeable) ordered collection of elements. Lists are defined using square brackets `[]`, allows for duplicate elements and can contain elements of different data types. Elements in a list can be accessed by index, starting with 0 (first element).

In [5]:
```python
# define an empty list
my_list = []
print("This is my empty list: ", my_list)

# each line adds the element inside parenthesis to the end of the list
my_list.append(200)
my_list.append("abc")
my_list.append(1.4)

print("This is my_list after appending elements: ", my_list)

print("The first element I have appended to my list is: ", my_list[0])
print("After that, I have appended: ", my_list[1])
print("If I don't remember how long my list is, I can use index -1 to get the
```

```
This is my empty list:  []
This is my_list after appending elements:  [200, 'abc', 1.4]
The first element I have appended to my list is:  200
After that, I have appended:  abc
If I don't remember how long my list is, I can use index -1 to get the last e
lement:  1.4
```

In [ ]:
```python
# We can modify the list by replacing one element
my_list[1] = "6000"
print("Here is my list now: ", my_list)

# We can check how long the list is
print("My list has lenght: ", len(my_list))

# We can modify the list by removing one element
second = my_list.pop(1) # pop returns the element at the given index and remo
print("Popped list element: ", second)
print("Now this is my list: ", my_list)
```

```
Here is my list now:  [200, '6000', 1.4]
My list has lenght:  3
Popped list element:  6000
Now this is my list:  [200, 1.4]
```

Tuples are immutable (unchangeable) ordered collections of elements. They are defined using parentheses `()`, allow duplicate elements, and can contain elements of different data types. Elements in a tuple can be accessed by index, similar to lists. Tuples are often used to store fixed collections of items that should not be modified.

In [ ]:
```
my_tuple = (1, 2, 3, 'a', 'b', 'c')
print("My tuple has lenght: ", len(my_tuple))
print("The fourth element in my tuple is: ", my_tuple[3])
```

Dictionaries are mutable (changeable) unordered collections of key-value pairs and do not allow duplicate keys (whereas a value can appear multiple times). They are defined using curly braces `{}`. Since dictionaries are unordered, elements in a dictionary are accessed by key rather than by index. Dictionaries are useful for storing data in a structured format, where each piece of data is associated with a unique identifier (key).

In [ ]:
```
# We can define an empty dictionary
my_dict = {}

# We can add the key "my_first_key" associated with the integer value 1
my_dict["my_first_key"] = 1

# Keys can also be integers. Here we associate 3 to a string value
my_dict[3] = "the value associated to integer 3 is this string"

# Values can be any data type. Here the value is a list
my_dict["another_key"] = [1, 2, 3]

# We can print all the keys
print(my_dict.keys())

# We can print all the values
print(my_dict.values())

# We can print the entire dictionary
print(my_dict)
```

Note: The `values()` method gives a listing of the values in the dictionary in "arbitrary" order. This order may sometime seem like the values are ordered based on the keys, especially if the keys are small integers. However, YOU CANNOT TRUST THIS TO HAPPEN EVERY TIME! Dictionaries are **unordered**!

Control flow in python

In Python, the `if`, `elif` (else if), and `else` statements are used for conditional execution of code. They allow you to control the flow of your program based on certain conditions.

The `if` statement is used to execute a block of code *if* a certain condition is true.

The `elif` statement is used to check additional conditions if the preceding `if` statement or `elif` statements (if any) evaluate to False. You can have multiple `elif` statements, each with its own condition. The `elif` statements are optional and can only appear after an `if`.

The `else` statement is used to execute a block of code if none of the preceding conditions (in the `if` and `elif` statements) evaluate to True. The `else` statement is optional and can appear only once in an `if-elif-else` block.

In [ ]:
```
x = 10
if x > 15:
    print("x is greater than 15")
elif x > 10:
    print("x is greater than 10 but less than or equal to 15")
else:
    print("x is less than or equal to 10")
```

Loops in python

In Python, loops are used to iterate over a sequence of elements or to execute a block of code repeatedly. There are two main types of loops in Python: `for` loops and `while` loops.

`for` loops are typically used when you know the number of times you want to iterate over a sequence or when you want to iterate over the elements of a sequence (like lists, tuples, strings, dictionaries, etc.)

In [ ]:
```
# Define a list of numbers
numbers = [1, 2, 3, 4, 5]

# Now iterate over all the element in the list
for num in numbers:
    # and for each of them, I print it
    print(num)
```

`while` loops are used when you want to execute a block of code repeatedly as long as a condition is true.

In [ ]:
```
# Define a variable to have value 0
count = 0

# while the value of my variable is below 5, ...
while count < 5:
    # ...print the value of my variable
    print(count)
    # ...and add 1 to my variable
    count += 1
    # ...and then go up again and check if the value of my variable is still

# if the value of my variable is 5 or more, we are done
print("Done!")
```

In addition to these basic loops, Python also provides a way to control the flow of a loop using statements like `break`, `continue`, and `else`.

- `break`: Terminates the loop prematurely when a certain condition is met.
- `continue`: Skips the rest of the code block and moves to the next iteration of the loop.

- `else` : Executes a block of code after the loop finishes executing, but only if the loop completed without encountering a break statement

In [ ]:
```
# Iterate over all the numbers from 0 to 10 (not including 10), ...
for num in range(10):
    # ...and if the number is 5,
    if num == 5:
        # ..then exit the loop
        break
    # ...if the number is a multiple of 2,
    if num % 2 == 0:
        # ...then continue without doing anything
        continue
    # next line is printed only if we are still inside the loop and the numbe
    print(num)

else:
    # this line would only be printed if the for loop has finished without en
    print("Loop completed successfully")
```

Functions

Functions are reusable blocks of code that perform a specific task when you call them. They allow you to break down your program into smaller, more manageable pieces, making your code more organized, readable, and maintainable. Functions help promote code reusability, reduce redundancy, and improve the overall structure of your programs.

You define a function using the `def` keyword followed by the function name and parentheses `()`. Any input parameters to the function are placed within the parentheses. To use a function, you call it by its name followed by parentheses `()`. If the function takes parameters, you pass the values for those parameters within the parentheses.

Functions can have zero or more parameters (also called arguments). Parameters are variables that receive values when the function is called. Parameters can have default values. A default value is already specified in the function definition.

Functions can return values using the return statement. This allows functions to send data back to the caller. A function can have multiple return statements, but only one will be executed in each function call. If a function does not explicitly return a value, it implicitly returns None.

It is good practice (but not mandatory) to specify the parameters types and the return types for a function. It is also good practice to include documentation for your functions using docstrings. Docstrings are string literals placed immediately after the function header, and they describe what the function does.

In [ ]:
```
# here I define a function
# the function name is greet
# the function takes name as parameter
# I expect the parameter to be a string (name: str)
# I expect the function to return a string (-> str:)
# This explanation should better be in a docstring!
def greet(name: str) -> str:
```

```
    """
    This text inside the triple quotes is a type of comment, called docstring
    A docstring should tell what the function does. Here, for example, we wan
    to say that this function greets the user by name. Then we can specify th
    parameters and return of this function like this:
    :param name: str, a string with the name of the person we want to greet
    :return: a string with our greetings to the person
    ------------
    Example
    ------------
    >>> greet(name="Adam")
    Hello, Adam!
    """
    return f"Hello, {name}!"


# Let's call the function
print(greet("John"))


# We can also save the returned value of the function
my_greetings_message = greet("Michael")
print(my_greetings_message)  # Output: Hello, Michael!


# A function with default parameter set to "Guest"
# and no types specified
def greet_my_guest(name="Guest"):
    return f"Hello, {name}!"

# If a parameter has default value, we can use it in default mode
print(greet_my_guest())    # Output: Hello, Guest!

# But we can also specify a different value
print(greet_my_guest(name="Juha"))    # Output: Hello, Juha!
```

Generators

Generators in Python are a way to create iterators in a simple and efficient manner. They are functions that allow you to generate a sequence of values over time rather than storing all the values in memory at once. This can be particularly useful when dealing with large amount of data or infinite sequences, because they allow you to generate values on-the-fly without having to store them all in memory. They are also commonly used in combination with loops to iterate over sequences of values in a memory-efficient manner.

The key feature of generators is the `yield` statement. When a function contains a `yield` statement, it becomes a generator. The `yield` statement is used to return a value from the generator and temporarily suspend the function's execution. Later, when the generator is called again, it resumes execution from where it left off.

Here's an example to illustrate how generators work:

In [ ]:

```
def my_generator():
    """
    This is a generator function that yields three values: 1, 2, and 3.
    :return: an integer
    """
```

```
    yield 1
    yield 2
    yield 3

# Create a generator object
gen = my_generator()

# When you call next() on the generator object, the code inside the generator
# until it encounters a yield statement. At that point, the value yielded by
# to iterate over the generator. Subsequent calls to next(gen) resume executi
# until the next yield statement is encountered.
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

In [ ]:

```
def fibonacci():
    """
    A generator to generate an infinite sequence of Fibonacci numbers
    """
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Create a generator object
fib_gen = fibonacci()

# Print the first 10 Fibonacci numbers
for _ in range(10):
    print(next(fib_gen))
```

Classes and objects

Classes and objects provide a way to model real-world entities and encapsulate their behavior and data. They are fundamental concepts in object-oriented programming (OOP) and are widely used in Python for creating reusable and organized code.

A class is a blueprint for creating objects (instances of a class). A class defines the attributes (data) and methods (functions) that belong to the objects created from it. Objects are instances of classes, and each object has its own unique set of attributes and methods.

In the following example, we define a class named `Car` with class attribute `wheels` and instance methods `__init__`, `drive`, `honk`. The `__init__` method is a special method called constructor. It initializes new objects with the provided attributes (`make`, `model`, `year`). The class attribute `wheels` is the same for all objects in the same class (all cars have 4 wheels).

In [ ]:

```
# Define a clas
class Car:
    # Class attribute
    wheels = 4

    # Constructor method (initializer)
    def __init__(self, make: str, model: str, year: int):
        # Instance attributes
        self.make = make
```

```
        self.model = model
        self.year = year

    # Instance method
    def drive(self):
        return f"{self.make} {self.model} is driving."

    # Instance method
    def honk(self):
        return "Beep! Beep!"

# Create two objects (instances) of the Car class,
# passing arguments to the constructor to initialize their attributes.
my_car = Car("Toyota", "Corolla", 2020)
your_car = Car("Honda", "Civic", 2018)

# Access attributes of an the object using dot notation (object.attribute).
print(my_car.make)  # Output: Toyota
print(your_car.year)  # Output: 2018

# Call methods of an object using dot notation (object.method()).
print(my_car.drive())  # Output: Toyota Corolla is driving.
print(your_car.honk())   # Output: Beep! Beep!

# Class attributes can also be accessed with dot notation
print(Car.wheels)  # Output: 4
```

How to read data from files and how to use Python libraries

Python has a vast ecosystem of libraries that cover a wide range of functionalities. Here, we introduce you to some popular Python libraries that are also useful for this course.

In Python, you can work with text data using built-in file object. You can read a file line by line using a file object's `readline()` method inside a loop.

In [ ]:
```
# This code looks for a file named filename.txt, opens it in read mode ('r'),
# and iterates over its lines, printing each line to the console
with open('filename.txt', 'r') as file:
    for line in file:
        print(line.strip())  # strip() removes leading/trailing whitespace an
```

We can also work with JSON data using the built-in `json` module, that provides functions to parse JSON strings into Python objects (deserialization) and serialize Python objects into JSON strings (serialization).

In [ ]:
```
# next line imports the library needed to work with JSON objects
import json

# This code looks for a file named data.json, opens it in read mode ('r'),
# then deserializes it from JSON to a string
with open('data.json', 'r') as file:
    data = json.load(file)

# Print the parsed JSON data
print(data)
```

NetworkX is a Python library for creating, analyzing, and visualizing complex networks (graphs). It provides data structures for representing various types of networks, along with algorithms for network analysis and manipulation. Next cell assumes that networkx is installed in your python environment.

In [ ]:

```
# next line imports the library networkx and assigns it the alias nx
import networkx as nx

# Create an undirected graph, that initially is empty
G = nx.Graph()

# Add nodes to our undirected graph
G.add_node(1)
G.add_nodes_from([2, 3])

# Add edges to our undirected graph
G.add_edge(1, 2)
G.add_edges_from([(2, 3), (1, 3)])

# Print graph information
print("Nodes:", G.nodes())
print("Edges:", G.edges())
```

In [ ]:

```

```