Web development

The Python language and Flask

DBMG

Our goal

- To teach you how to use Python to develop web applications? Not really..
- Goal: to teach you how to interact with a database through the web
  - Reading the data inserted by the user in a HTML form
  - Interacting with a DBMS (specifically MySQL): connecting to the database, sending the query, memorizing its result, ...
  - Accessing the tables returned by the DBMS
  - Building the HTML page to be visualized on the browser, combining HTML instructions and the data retrieved from the database

Contents

- Overview of the Python language
  - Structure of a program
  - Template
- Reading the parameters from a HTML form
  - Validating the parameters

Python frameworks (1)

- Flask
  - microframework
  - Jinja2 templates
  - ORM handled by other packages
  - No admin interface
  - No authentication system
- Django
  - Complex applications
  - Object-Relational Mapping (ORM)
  - Model-Template-View
  - Included admin interface
  - Built-in authentication system

Python frameworks (2)

- CherryPy
  - Built-in tools for encoding, sessions, caching, authentication, static content
  - ORM handled by other packages

- Pyramid
  - Object-Relational Mapping (ORM)
  - URL mapping based on Routes configuration
  - Templates management
  - Flexible authentication system

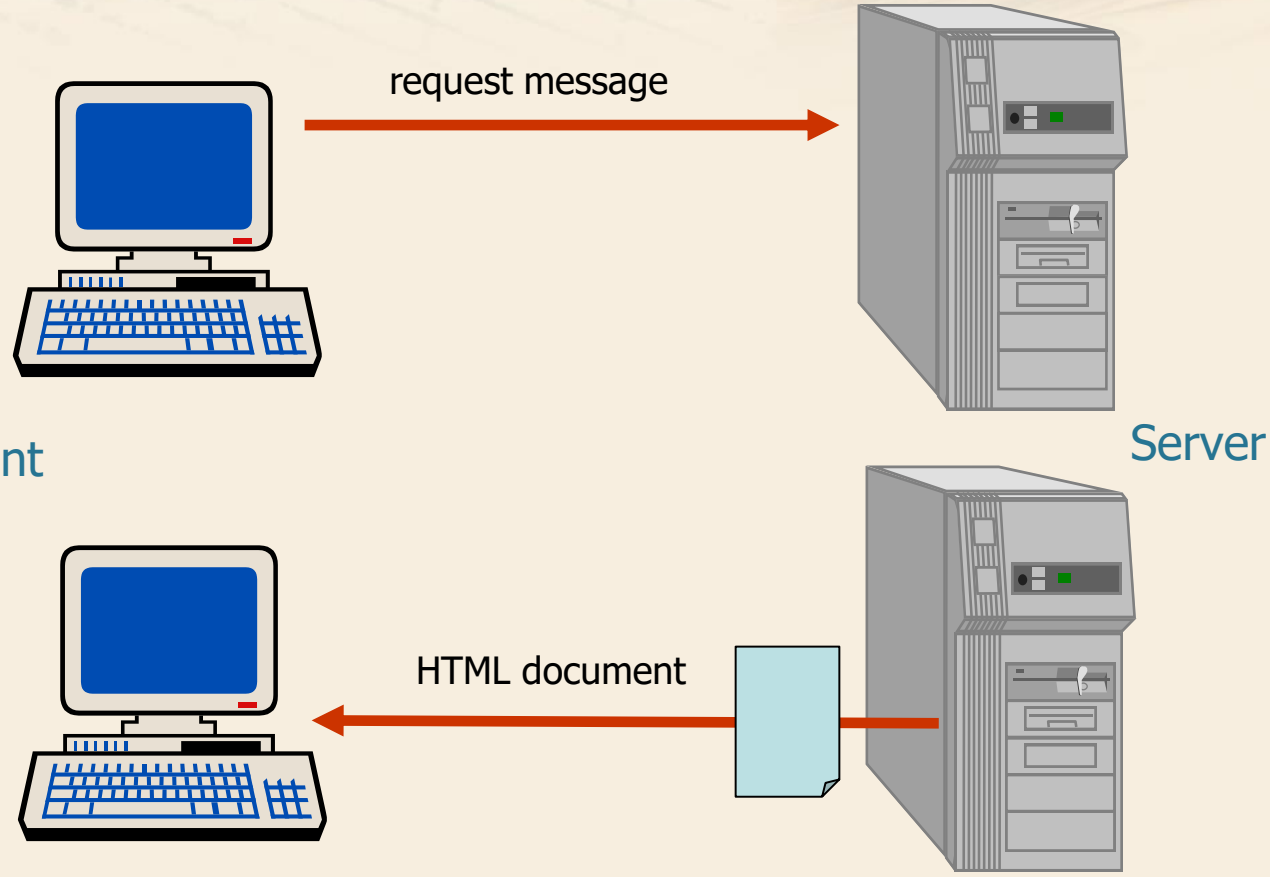
- Web2Py
  - Supports the Model-View-Controller (MVC) architecture
  - Allows to work with relational databases and NoSQL

- many more….

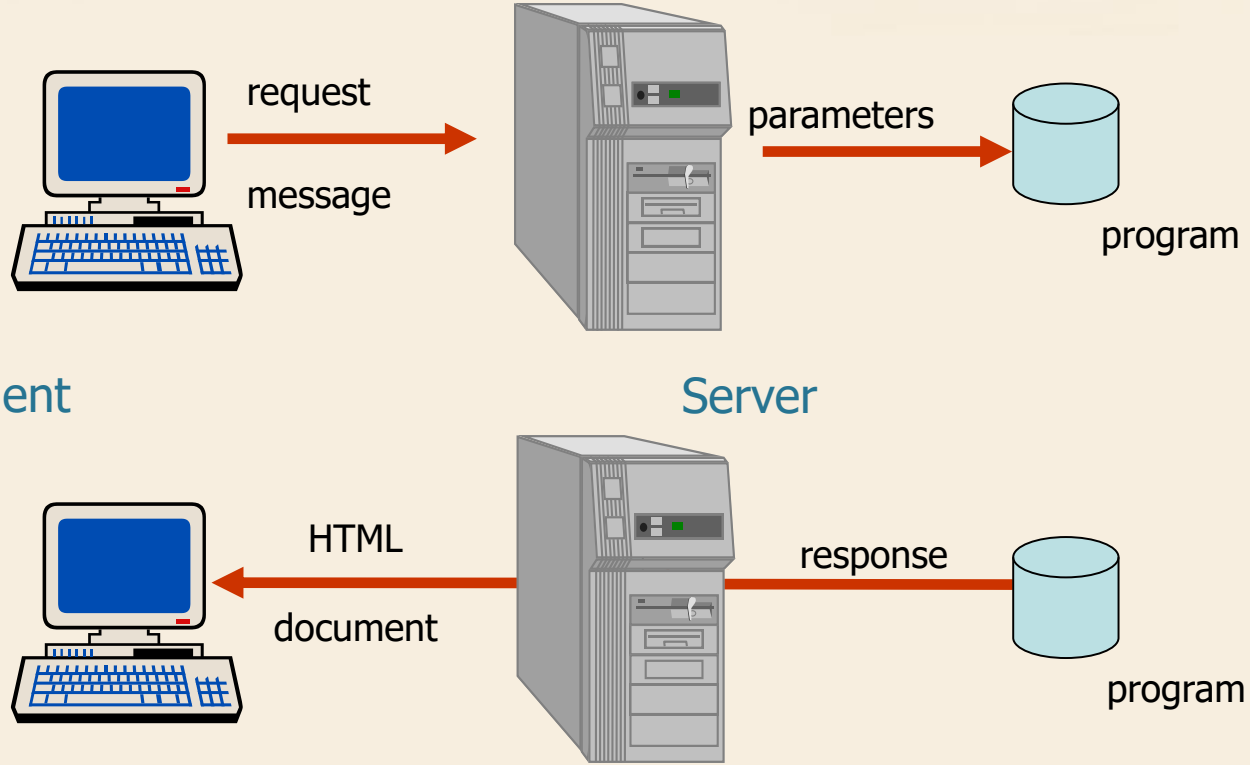
What's Flask

- Born in 2010
- Flask is a *micro-framework* developed in Python to create web applications
  - It offers *basic functionalities* to develop web applications
  - It doesn't require any additional library to work
  - Its functionalities can be extended with many different Python libraries
- Lots of useful resources, e.g.
  - https://flask.palletsprojects.com/en/2.0.x/

Static Web pages



Dynamic Web pages



Main goal

- Flask's main purpose is to develop dynamic web pages in HTML
  - Specifically, to produce HTML code "conditioned" by the results of a computation, that depends on the users' input, or the data stored in the database, …
  - Flask's code is executed on a Web server and the response (HTML document and script's results) are sent to the browser

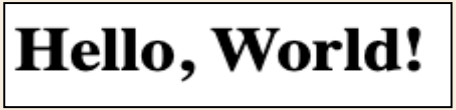
Why Python?

- Available for many platforms, regardless of
  - Hardware (Intel, Spark, Mac, ecc....)
  - Operating system (Linux,Unix, Windows, etc...)
- Python code is "highly portable"
- The Python interpreter is Open Source
  - Free, wide availability of tools, support, developers, and users' community
- Flask is easy to learn once you know Python
- Capable of interacting with various Database Management Systems (MySql, Postgres, Oracle, ...)

First example

- Text file with the .py extension

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<h1>Hello, World! </h1>"
```



First example

- If I open the script in the browser…

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<h1>Hello, World! </h1>"
```

- Why?
  - The browser DOES NOT visualize the result of the execution, but the Python file itself
  - To visualize the result, we need something to execute the code

Another example

- Visualize the current date
- The static way
  - What about tomorrow?

```
<html>
  <body>
    Today is 09/12/2011
  </body>
</html>
```

- The dynamic way
  - Updates in real time

```
from datetime import datetime
from flask import Flask

app = Flask(__name__)

@app.route("/")
def current_date():

    #Get current date in format: DD-MM-YYYY
    today = datetime. today().strftime('%d-%m-%Y')

    return today
```

Code analysis 1/2

- Importing the libraries (modules):
  - from datetime import datetime
    from flask import Flask
- Creating an object of **class** Flask:
  - Flask (__name__)
    - __name__ refers to the module's name
- Specifying **decorators** to bind a URL to a function used to create HTML
  - route("/")
- Defining the function that creates the HTML code
  - def current_date()

```
from datetime import datetime
from flask import Flask

app = Flask(__name__)

@app.route("/")
def current_date():

    #Get current date in format: DD-MM-YYYY
    today = datetime. today().strftime('%d-%m-%Y')

    return today
```

Code analysis 2/2

- Comments:

```
#… single line
''' multi line '''
```

- Variables: `today`
- Functions: `datetime.today()`
- Operators and language constructs: `return`

```
from datetime import datetime
from flask import Flask

app = Flask(__name__)

@app.route("/")
def current_date():

    #Get current date in format: DD-MM-YYYY
    today = datetime. today().strftime('%d-%m-%Y')

    return today
```

Decorators

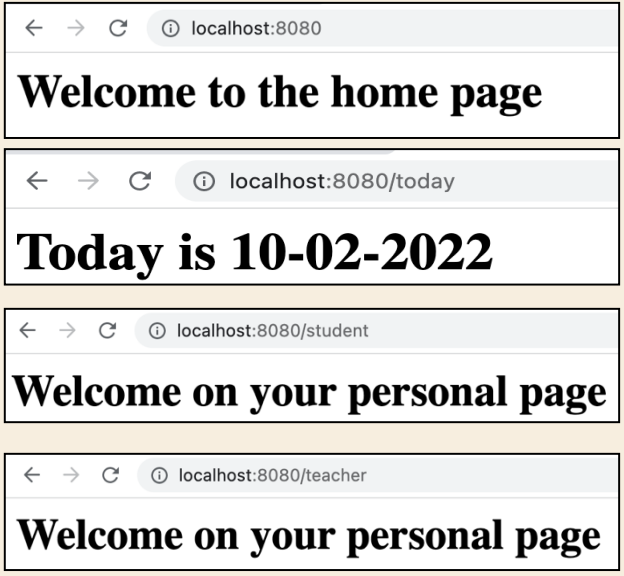
- Decorators are used to add functionalities to a function
- route(): decorator linked to the Flask object
  - It binds a function to a URL
  - It tells the function which HTTP methods to handle (GET/POST)

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<h1>Hello, World! </h1>"
```

Decorators

- A Web application can handle
  - different functions and different decorators
  - the same function can have many decorators

```
from datetime import datetime
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home_page():

    return "<h1>Welcome to the home page </h1>"

@app.route("/today")
def current_date():

    today = datetime. today().strftime('%d-%m-%Y')

    return "<h1> Today is " + today + " </h1>"

@app.route("/student")
@app.route("/teacher")
def personal_page():

    #Get current date in format: DD-MM-YYYY
    today = datetime. today().strftime('%d-%m-%Y')

    return "<h1>Welcome on your personal page </h1>"
```



Web server using Flask

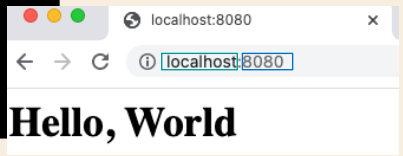
- Flask, in addition to being a library to develop web applications, also includes a **web server**
- The web server allows to run Python scripts **locally** without connecting to an external server
  - The PC becomes client and server
- The web server automatically creates a (local) virtual domain at the address
  - localhost (http://127.0.0.1, http://localhost)
  - At port 5000
  - It's not necessary to be connected to the Internet to run the web server

FLASK : Local web server

- Once installed, Flask can start the web server from within any directory of the PC

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<h1>Hello, World! </h1>"
```

```
[dgiordan@smartdata-danilo:~/web_examples$ export FLASK_APP=1_hello.py
[dgiordan@smartdata-danilo:~/web_examples$ flask run -p 8080
 * Serving Flask app '1_hello.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deplo
yment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
127.0.0.1 - - [10/Feb/2022 12:02:47] "GET / HTTP/1.1" 200 -
```



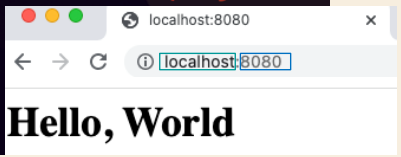
- Once started, the web page is available at the address http://127.0.0.1/ or http://localhost
  - **-p 8080** to specify the port at which the web page is created
    - The port is specified at the end of the address
    - Default port 5000 - **WARNING MACOS**: that port is already used

FLASK : Local web server

- Once installed, Flask can start the web server from within any directory of the PC

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<h1>Hello, World! </h1>"

app.run(port=8080)
```

```
dgiordan@smartdata-danilo:~/web_examples$ python3 1_hello_run.py
 * Serving Flask app '1_hello_run' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
127.0.0.1 - - [10/Feb/2022 17:58:25] "GET / HTTP/1.1" 200 -
```



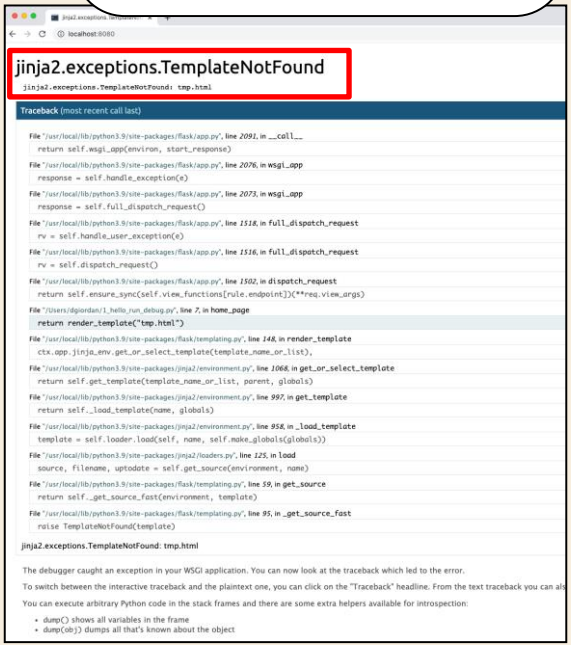
- Once started, the web page is available at the address http://127.0.0.1/ or http://localhost
  - **port=8080** to specify the port at which the web page is created
    - The port is specified at the end of the address
    - Default port 5000 - **WARNING MACOS**: that port is already used

FLASK : Local web server

- Flask can start the web server in **debug** mode
  - Useful to modify the page and see the changes without having to restart the server
  - To identify possible programming **errors**

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<h1>Hello, World! </h1>"

app.run(port=8080, debug=True)
```

```
dgiordan@smartdata-danilo:~/web_examples$ python3 1_hello_run_debug.py
 * Serving Flask app '1_hello_run_debug' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 257-448-331
127.0.0.1 - - [10/Feb/2022 18:06:34] "GET / HTTP/1.1" 200 -
 * Detected change in '/Users/dgiordan/web_examples/1_hello_run_debug.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 257-448-331
127.0.0.1 - - [10/Feb/2022 18:06:54] "GET / HTTP/1.1" 200 -
 * Detected change in '/Users/dgiordan/web_examples/1_hello_run_debug.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 257-448-331
```



Limitations of using Flask by itself

- Flask it's not meant to generate the entire content of a web page, but mainly the dynamic part of it

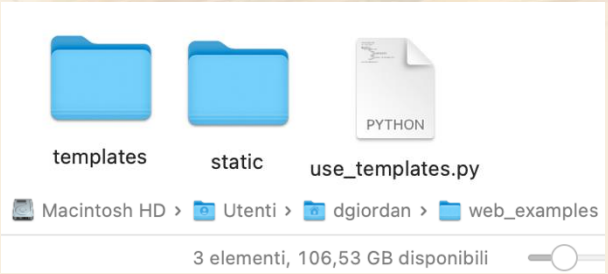
```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<html>\
                <head>\
                   <title>=Home Page </title>\
                </head>\
                <body>\
                   <h1> Welcome to the home page! </h1>\
                </body>\
              <html>"

app.run(port=8080, debug=True)
```

Template

- In addition to being able of generating the entire HTML code of a web page, Flask allows to interact with **templates**
  - Templates allow to have a basic static web page, written in HTML, and insert ONLY the dynamic part using Flask
    - This simplifies the creation of the web pages, since **only** the dynamic elements will be managed by Flask
    - Based on the language and templates' engine **Jinja2**

Template

- To use templates, it's necessary to create a folder
  - templates
- It's possible to create other folders to add static resources
  - CSS, images, etc.



use_template.py

```
from flask import Flask,
render_template

app = Flask(__name__)

@app.route("/")
def hello_world():
    return render_template("home.html")
app.run(port=8080, debug=True)
```

+

home.html

```
<html>
<head>
     <title> Home Page </title>
</head>
<body>
     <h1> Welcome to the home page </h1>
</body>
</html>
```

Jinja2

- To make a template dynamic, **Jinja2** is used
- **Jinja2** code is inserted in-between HTML code, using **delimiters**
  - **{% ... %} to insert instructions and control structures**
  - **{{ ... }} to handle variables**

home.html

```
<html>
<head>
     <title> Home Page </title>
</head>
<body>
     {% if page_name %}
          <h1> Welcome to the {{page_name}} </h1>
     {% endif %}
</body>
</html>
```

Jinja2: Passing variables

- Flask passes the **variables** as parameters of the render_template function
- Variables can be of any type
  - **Strings, numbers, arrays, dictionaries**

use_template.html

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def hello_world():
    return render_template("home.html", page_name="custom name")

app.run(port=8080, debug=True)
```

+

home.html

```
<html>
<head>
     <title> Home Page </title>
</head>
<body>
     {% if page_name %}
          <h1> Welcome to the {{page_name}} </h1>
     {% endif %}
</body>
</html>
```

=

Jinja2: Variables

- As well as visualizing variables passed by Flask, it's also possible to define variables in Jinja2
  - Useful to perform operations inside the template
  - The **set** instruction is used

use_template.html

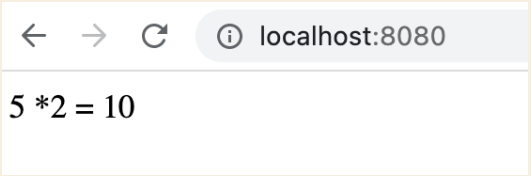
```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def hello_world():
    return render_template("home.html", number = 2)

app.run(port=8080, debug=True)
```

+

home.html

```
<body>
    {% set ris = 5* number%}
    5 * {{number}} = {{ris}} <br>
</body>
```

=



Jinja2: Strings

- To concatenate strings, you can simply insert the string variables directly inside the HTML code

use_template.html

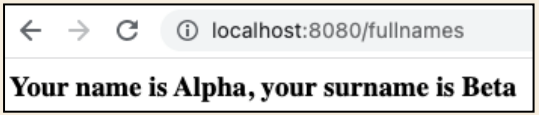
```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/fullnames")
def programs():
    return render_template("fullnames.html", name = "Alpha", surname = "Beta")

app.run(port=8080, debug=True)
```

+

fullnames.html

```
<body>
    <h3> Your name is {{name}}, your surname is {{surname}} </h3>
</body>
```

=



Main functions for arrays and dictionaries

- **array | length**, **dict | length** : returns the number of elements in the array or dictionary
- **array | sort**: sorts the array. Optional parameters can be used to specify how to sort.
- **dict | dictsort**: sorts the dictionary. Optional parameters can be used to specify how to sort.
- key in array, key in dict: verifies whether an element ***key***, exists in the array ***array*** *or in the dictionary* ***dict***

Control structures

- The control instructions and structures allow the conditional execution of portions of the program
- They also allow the iterative execution of portions of the program
- They evaluate certain conditions
- Control structures in Jinja2
  - if, if..else, if..elseif
  - for

Conditions

- A condition is an expression that returns a boolean value (true or false)
  - They are written using comparison operators and logical operators
- The following evaluate as False
  - The boolean value False
  - The integer number 0 and the float number 0.0
  - An empty string (“”) e and the “0” string
  - An empty array
- All the other values evaluate to true

The if and if...else constructs

- If the condition in the IF block is true, the block of operations is executed

```
@app.route("/")
def hello_world():
    return render_template("home.html", page_name = "custom name")
```

+

```
{% if page_name %}
     <h1> Welcome to the {{page_name}} </h1>
{% endif %}
```

- If the condition in the IF block is true, the block of operations is executed, otherwise the ELSE branch is executed

```
@app.route("/")
def hello_world():
    return render_template("home.html")
```

+

```
{% if page_name %}
     <h1> Welcome to the {{page_name}} </h1>
{% else %}
     <h1> Welcome to the Home Page! </h1>
{% endif %}
```

The if…elseif construct

- Allows the definition of more options

```
@app.route("/")
def hello_world():
    return render_template("home.html", page_name = "about")
```

+

```
</body>
   {% if page_name=="home" %}
        <h1 style="color:blue"> Welcome to the Welcome to the Home Page! </h1>
   {% elif page_name== " about" %}
        <h1 style="color:red"> Welcome to the About Page! </h1>
   {% else %}
        <h1> Welcome to the {{page_name}}! </h1>
   {% endif %}
</body>
```

=

The for cycle

- Allows the repetition of a block of instructions, by defining directly
  - The initialization instructions, executed only once at the very entrance of the cycle
  - The condition, that must be true to execute the block of instructions
  - The update instruction, executed at the end of each iteration

```
<body>
    {% for i in range(1,11) %}
        {% set ris=5*i %}
        5 *{{i}} = {{ris}} <br>
    {% endfor %}
</body>
```

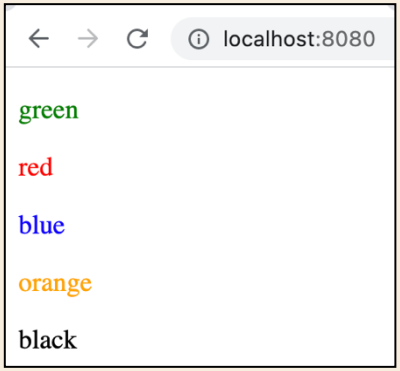
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50

The for cycle

- Similarly to Python, Jinja2 allows to iterate over
  - **arrays**
  - dictionaries

```python
@app.route("/")
def hello_world():
    colors = ["green", "red", "blue", "orange", "black"]
    return render_template("home.html", colors = colors)
```

```
</body>
   {% for color in colors %}
        <p style="color:{{color}}"> {{color}} </p>
   {% endfor %}
</body>
```

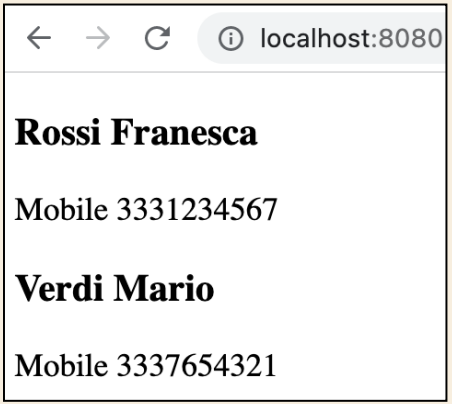


The for cycle

- Similarly to Python, Jinja2 allows to iterate over
  - **arrays**
  - dictionaries

```
@app.route("/")
def hello_world():
    agenda = [
                {"surname": "Rossi",
                 "name": "Francesca",
                 "mobile": 3331234567
                },
                {"surname": "Verdi",
                 "name": "Mario",
                 "mobile": 3337654321
                }
              ]
    return render_template("home.html", agenda = agenda)
```

```
</body>
  {% for record in agenda %}
      <h3> {{record.surname}} {{record.name}} </h3>
      Mobile {{record.mobile}}
  {% endfor %}
</body>
```

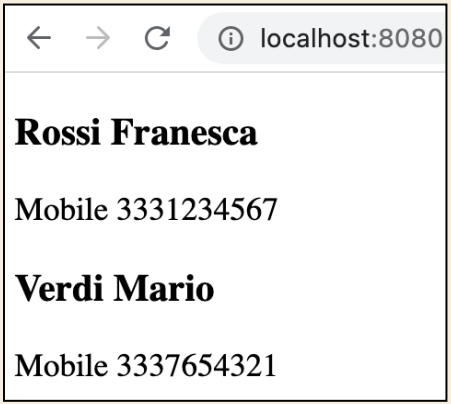


The for cycle

- Similarly to Python, Jinja2 allows to iterate over
  - arrays
  - **dictionaries**

```
@app.route("/")
def hello_world():
    agenda = {
                "one": {"surname": "Rossi",
                  "name": "Francesca",
                  "mobile": 3331234567
                 },
                "two": {"surname": "Verdi",
                  "name": "Mario",
                  "mobile": 3337654321
                 }
               }
    return render_template("home.html", agenda = agenda)
```

```
</body>
  {% for record in agenda %}
    <h3> {{agenda[record].surname}} {{agenda[record].name}} </h3>
    Mobile {{agenda[record].mobile}}
  {% endfor %}
</body>
```



Template extension (inheritance)

- As well as creating new templates, it's also possible to **extend** existing templates
  - By creating a **base template** containing the skeleton of the page and all the elements in common between different pages
  - The **child templates** inherit the base template, and only specify a subset of **blocks** to be modified

Template extension: Base template

- The base template defines the dynamic portions by means of
  - **{% ... %} to insert instructions and control structures**
- The blocks of content defined inside child templates are identified by
  - **{% block <nome> %}**
  - **{% endblock <nome> %}**

layout.html

```
<html>
<head>
{% if title %}
     <title> {{ title }} </title>
{% else %}
     <title> Default Title </title>
{% endif%}
</head>
<body>
    {% block content %}
    {% endblock content %}
</body>
</html>
```

Template extension: Child templates

- Child templates identify the base template by means of the instruction
  - **{% extends "name" %}**
- Child templates define which block of data they want to modify
  - **{% block <name> %}**
  - **{% endblock <name> %}**

home.html

```
{% extends "layout.html" %}
{% block content %}
    <h1> Welcome to the Home Page! </h1>
{% endblock content %}
</body>
</html>
```

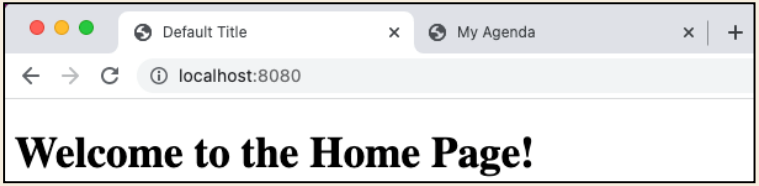
agenda.html

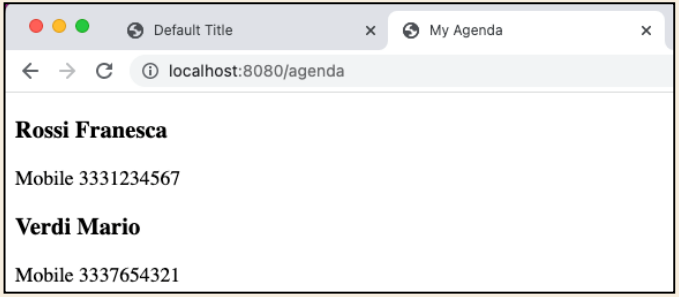
```
{% extends "layout.html" %}
{% block content %}
  <h3> {{agenda[record].surname}} {{agenda[record].name}} </h3>
  Mobile {{agenda[record].mobile}}
{% endblock content %}
```

Template extension: Flask

- The extension is transparent to Flask
  - Child templates are retrieved directly
- Child templates accept as arguments
  - A variable number of parameters
  - Different parameters

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/agenda")
def agenda_page():
   return render_template("home.html")

@app.route("/agenda")
def agenda_page():
agenda = {
                    "one": {"surname": "Rossi",
                 "name": "Francesca",
                 "mobile": 3331234567
                },
                    "two": {"surname": "Verdi",
                 "name": "Mario",
                 "mobile": 3337654321
                }
              }
   return render_template("agenda.html", agenda = agenda,
                          title = "My Agenda")
app.run(port=8080, debug=True)
```





Flask and HTML forms

```
<form name="UserData" action="processlogin" method="GET">
    Input Elements
</form>
```

- "form" tag with some attributes
  - Name: name of the form
  - Action: name of the program that will process the data
  - Method: HTTP method used to pass the form parameters to the program (can be "GET" or "POST")
- Inside the form there are multiple input elements

Accessing the data of the form

- To send the form data to the response page (**action**)
  - Both the form page and the response page **must be handled** by the web server
- To access the parameters passed by a form, it's necessary to import the **request** object from Flask
  - **from** flask **import** request
- The functions specify the **method** accepted to receive data inside the decorator
  - route("/answer", **methods**=['GET','POST'])
  - If the method is not specified, Flask only accepts parameters from requests of type GET

Accessing the data of the form

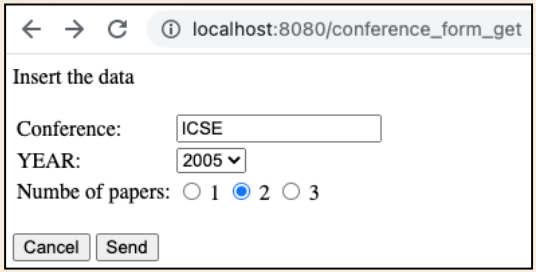
- GET method
  - To access the parameters passed by GET in the URL (?key=value) it's possible to use the args attribute inside the **request** object
  - **request**.args.get("<parameter>")
- POST method
  - To access the parameters passed by POST it's possible to use the **request** object directly, as an associative array
  - **request**.form["<parameter>"], **request**.files["<parameter>"]

Accessing the data of the form

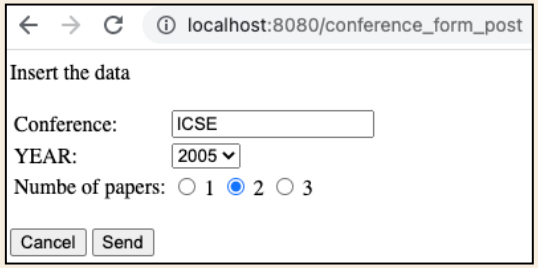
- If both the GET and POST methods can be used
  - Before accessing the parameters, it's necessary to verify the method used in the request object
  - request.method == 'GET'
- The request object can also be used to obtain other kind of information
  - Cookie, sessions, etc.

Example: Form

GET



POST



conference_form_get.html

```
<html>
<head>
   <title>GET form</title>
</head>
<body>
   <p> Insert the data </p>
   <form method="GET" action="conference">
   <table>
      <tr>
          <td> Conference: </td>
          <td> <input type="text" name="conf" size="20"> </td>
      </tr>
      <tr>
          <td> YEAR: </td>
          <td> <select name="year">
                  <option value="2005"> 2005 </option>
                  <option value="2006"> 2006 </option>
                </select> </td>
      </tr>
      <tr>
          <td> Number of papers: </td>
          <td> <input type="radio" name="num" value="1"> 1
                <input type="radio" name="num" value="2" checked> 2
                <input type="radio" name="num" value="3"> 3 </td>
       </tr>
     </table> <br />
     <input type="reset" value="Cancel">
     <input type="submit" value="Send">
</form>

</body>
</html>
```

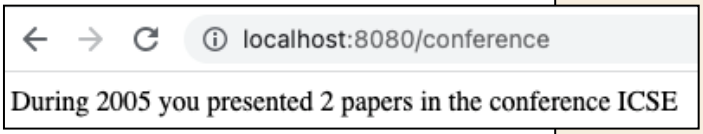
Example: Receiving FORM parameters

File conferences.py

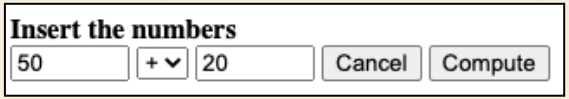
```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route("/conference_form_get")
def get_form():
   return render_template("conference_from_get.html")

@app.route("/conference_form_post")
def post_form():
   return render_template("conference_from_post.html")

@app.route("/result", method=["GET","POST"])
def read_form():

    if(request.method == 'GET'):
        conf = request.args.get('conf')
        year = request.args.get('year')
        num  = request.args.get('num')
    else:
        conf = request.form['conf']
        year = request.form['year']
        num  = request.form['num']

    output_string = "During %s you presented %s papers in the conference %s"%(year,num,conf)

    return render_template("conference.html", output_string = output_string)

app.run(port=8080, debug=True)
```

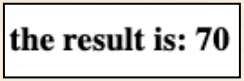
conference.html

```
<html>
<head>
   <title>View form parameters</title>
</head>
<body>
     <p> {{ output_string}} </p>
</body>
</html>
```



Example: calculator






```
<html>
<head>
   <title>Calculator</title>
</head>
<body>
   <p> Insert the numbers </p>
   <form method="GET" action="result">
      <input type="text" name="val1" size="8" maxlength="8">
      <select name="year">
           <option value="sum"> + </option>
           <option value="sub"> - </option>
           <option value="mul"> * </option>
           <option value="div"> / </option>
      </select>
      <input type="text" name="val2" size="8" maxlength="8">
      <input type="reset" value="Cancel">
      <input type="submit" value="Compute">
</form>

</body>
</html>
```

Example: calculator

```python
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route("/calculator")
def get_form():
   return render_template("calculator.html")

@app.route("/result", method=["GET"])
def read_get_form():

    val1 = request.args.get('val1')
    val2 = request.args.get('val2')
    op   = request.args.get('op')

    if(val1=="" or val2==""):
        return render_template('result.html', error_message = "Missing one or more numbers. Please check the input.")
    if(val1.isnumeric() == False or val2.isnumeric() == False):
        return render_template('result.html', error_message = "Only integer number are supported. Please check the input.")

    val1 = int(val1)
    val2 = int(val2)
    if(op == "div" and val2 == 0):
        return render_template('result.html', error_message = "Error try division by zero.")

    result = 0
    if(op == "sum"):
        result = val1 + val2
    if(op == "sub"):
        result = val1 - val2
    if(op == "mul"):
        result = val1 * val2
    if(op == "div"):
        result = val1 / val2

    return render_template('result.html', result = result)

app.run(port=8080, debug=True)
```

result.html

```html
<html>
<head>
   <title>Result</title>
</head>
<body>
   {% if error_message %}
       <h3 style="color:red"> {{ error_message}} </h3>
   {% else %}
       <h3> the result is: {{ result }} </h3>
    {% endif %}
</body>
</html>
```

Example: multiple choice

Please select the programming language you know

☐ C ☐ C++ ☐ Perl

☑ HTML ☑ Python ☐ Java

Cancel Send

You know 2 programming languages:

- HTML
- Python

Example: multiple choice

- HTML form
  - Use the langs array instead of 6 distinct variables

```
<html>
<head>
   <title>Programming Language</title>
</head>
<body>
   <p> Please select the programming language you know </p>
   <form method="GET" action="known">
   <table>
      <tr>
          <td> <input type="checkbox" name="langs" value="C"> C </td>
          <td> <input type="checkbox" name="langs" value="C++"> C++ </td>
          <td> <input type="checkbox" name="langs" value="Perl"> 3 </td>
       </tr>
       <tr>
          <td> <input type="checkbox" name="langs" value="HTML"> HTML </td>
          <td> <input type="checkbox" name="langs" value="Python"> Python </td>
          <td> <input type="checkbox" name="langs" value="Java"> Java </td>
       </tr>
       <tr>
           <td> <input type="reset" value="Cancel">
           <td> <input type="submit" value="Send">
        </tr>
   </table>
   </form>
</body>
</html>
```

Example: multiple choice

- Python script
  - The languages array contains all the selected values (HTML, Python in this example)

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route("/programs")
def get_form():
   return render_template("programs.html")

@app.route("/known", method=["GET"])
def known():

  languages = request.args.getlist('langs')
  return render_template('known.html', result = languages)

app.run(port=8080, debug=True)
```

```
<html>
<head>
   <title>Known Programming Language</title>
</head>
<body>
   {% if languages | length == 0 %}
       <h3>You don't know any programming language.</h3>
   {% else %}
       <p> You know {{languages | length }} programming languages </p>
       <ul>
       {% for l in languages %}
            <li> {{l}} </li>
       {% endfor %}
       </ul>
    {% endif %}
</body>
</html>
```

You know 2 programming languages:

- HTML
- Python

Validating the inserted values

- Before processing the data provided by the user, it's always convenient to **validate** them
  - Avoid processing invalid data
    - E.g., inserting an email address not correctly formatted, or an unexpected value
  - Useful to avoid potential cyberattacks
    - E.g., inserting SQL queries in a form field to visualize the content of the DB (SQL injection)

Data validation

- Verify that the age inserted by the user respects the service constraints (be at least 16 years old)

```
from flask import Flask, render_template, request

app = Flask(__name__)

def check_age(age):
    if(age == ""): return False
    if(age.isnumeric() == False): return False
    if(int(age) < 16): return False

    return True

@app.route("/register")
def register_form():
    return render_template("register.html")

@app.route("/registered", method=["GET"])
def process_registration():

    surname = request.args.get('surname')
    age     = request.args.get('age')

    if(checkAge(age) == False):
        return render_template('error_page.html', errror = 'Invalid Age')

    return render_template('registered.html')

app.run(port=8080, debug=True)
```

Verify the insertion of the age field

Verify that age is a number

Verify the age constraint

Data validation

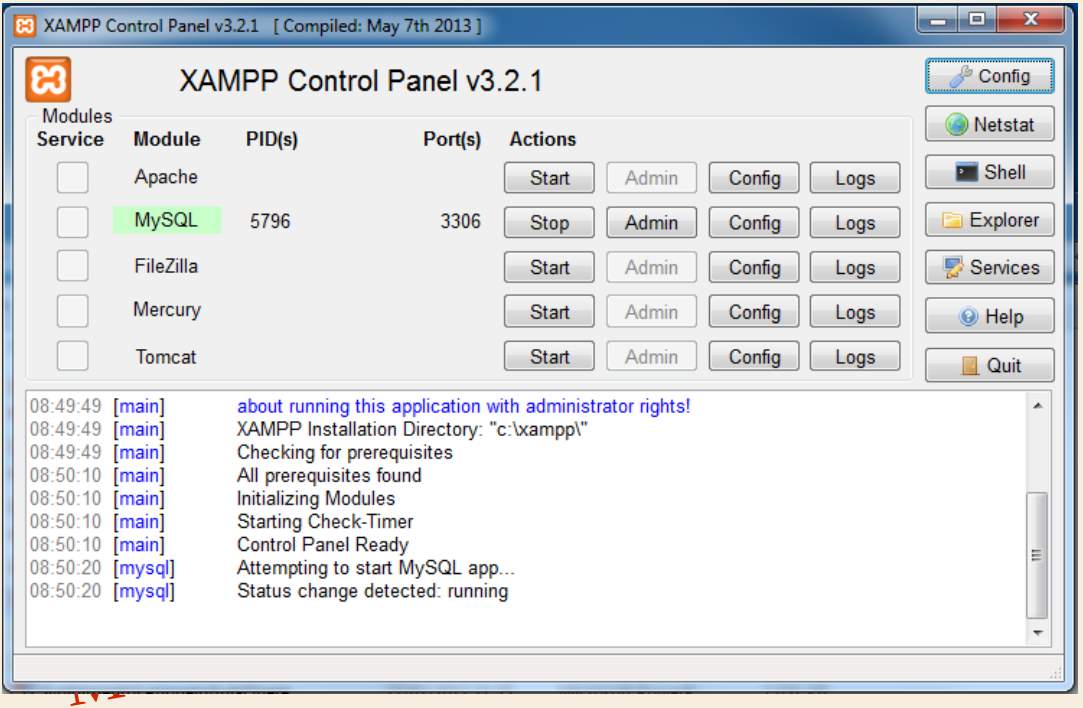
- Before processing the data it's necessary to
  - Verify that the user inserted the data
  - Do a formal verification to ensure **the type** is correct
    - The formal verification can be done exploiting functions and libraries provided by Python, regular expressions, etc.
  - Validate potential constraints
    - E.g., the minimum age requirement

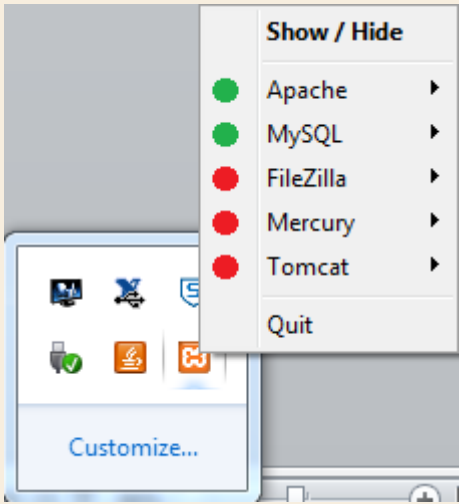
A brief digression: XAMPP

- XAMPP is a web development platform (web-database development environment) that includes
  - A web server (Apache)
  - A database management system (MySQL)
  - An interpreter for PHP e PERL scripts
  - **A graphical administrator for MySQL db (phpMyAdmin)**
- It allows to run locally a MySQL server

XAMPP: services administration

- It allows to manage services
  - Graphical user interface





XAMPP: DB administration

- It allows to manage databases
  - Graphical user interface