Python – Základy práce s Web Servismi

Postavenie jednoduchého python serveru

- Použijeme knižnicu flask

```
from flask import Flask

app = Flask(__name__)

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

- Uložíme ako app.py
- Spustíme pomocou `flask run`

```
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production
deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

- Keď zavoláme náš server (`curl localhost:5000 -i`):

```
HTTP/1.1 200 OK
Server: Werkzeug/3.1.3 Python/3.13.7
Date: Tue, 18 Nov 2025 12:35:02 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 20
Connection: close

<p>Hello, World!</p>
```

- Uvidíme prichádzajúcu požiadavku v termináli:

```
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production
deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [18/Nov/2025 07:33:40] "GET / HTTP/1.1" 200 -
```

Dekorátory – intermezzo

- Mechanizmus, ktorý umožňuje:
  - pridať k funkcii dodatočné správanie *pred* alebo *po* jej vykonaní
  - úplne nahradiť jej správanie vlastnou logikou
  - funkcii meniť vstupy a výstupy

- Ide o volanie jednej funkcie (dekorátor) na inú funkciu (dekorovaná funkcia), pričom výstupom je **nová upravená funkcia**

- Použitie cez @ nad cieľovou metódou ktorú chceme dekorovať aby bola úprava prehľadná a oddelená od samotnej logiky

- Vo Flasku sa dekorátory používajú na definovanie HTTP endpointov (`@app.route(...)`) - oddelíme technické detaily od samotnej logiky aplikácie

Dekorátory – vlastný dekorátor

- Definovanie vlastného dekorátoru:

```
def decorator(func):
    print("Some logic before")
    return func

@decorator
def example():
    print("Example function is called")

if __name__ == "__main__":
    example()
```

- Keď náš script pustíme dostaneme nasledovný výpis:

```
Some logic before
Example function is called
```

Dekorátory – poradie volania

- Keď si spustíme nasledovný ukážkový script:

```
def first(func):
    print("Executing first")
    return func

def second(func):
    print("Executing second")
    return func

@first
@second
def example_function():
    print("Executing example function")

if __name__ == "__main__":
    print("Running main")
    example_function()
```

- Môžeme vidieť, že dekorátory sa vykonajú ešte pred tým ako sa zavolá samotná funkcia

```
Executing second
Executing first
Running main
Executing example function
```

- Treba na postupnosť myslieť kvôli importovaniu a prípadným globálnym premenným

Dekorátory – wrappery

- Ak chceme vykonať úkon aj po skončení metódy / poslať dekorátoru argumenty - wrapper

```
def result_greater_than(target: int):
    def decorator(func):
        def wrapper(*args):
            result: int = func(*args)
            if result > target:
                print(f"[*] {result} is greater than {target}")
            else:
                print(f"[*] {result} is not greater than {target}")
            return result
        return wrapper
    return decorator
```

- Tento dekorátor následne vieme použiť napríklad nad metódou na sčítanie čísel

```python
@result_greater_than(10)
def summary(*args):
    return sum(args)

if __name__ == "__main__":
    summary(1,2,3,4,5,6,7)
    summary(1,1,1,1,1)
```

- Po spustení tohto scriptu dostaneme nasledujúci výstup

```
[*] 28 is greater than 10
[*] 5 is not greater than 10
```

- Vieme využiť na opakované kontroly argumentov prípadne transformácie výstupov metód

Postavenie jednoduchého python serveru – pokračovanie

- Upravíme náš app.py súbor aby niečo robil

```
from flask import Flask, request

app = Flask(__name__)

@app.route("/sum", methods=['POST', 'GET'])
def summary():
    json_data = request.get_json(silent=True) or dict()
    arg = request.args
    numbers = json_data.get("numbers", arg.getlist("numbers")) or []
    return {"summary": sum([int(x) for x in numbers])}
```

- *methods* môže byť ľubovoľné pole HTTP metód obsahujúce napríklad `['GET','POST','PUT','DELETE']`

- Keď pošleme POST požiadavku na náš server:

```
curl -X "POST" -H "Content-Type: application/json" -d
'{"numbers":[1,2,3,4]}' http://localhost:5000/sum
```

- Dostaneme odpoveď:

```
{"summary":10}
```

- Rovnako keď pošleme GET požiadavku na náš server:

```
curl -X GET 'http://localhost:5000/sum?numbers=1&numbers=2'
```

- Dostaneme odpoveď:

```
{"summary":3}
```

Odosielanie požiadaviek

- Knižnica *requests*
- Využijeme triedu *Session* na udržiavanie používateľskej relácie a nastavených cookies
- Odpoveď môžeme získať pomocou
  - .content - dostaneme *bytes* - ak server vracia raw http data
  - .json() – dostaneme *dict* - v prípade, že server vracia json dáta

```
import requests

session = requests.Session()

data = {"numbers": [1, 2, 3, 4]}
result = session.post("http://localhost:5000/sum", json=data)

print(result.content.decode())
print(result.json())
```

- Keď tento script pustíme uvidíme nasledovný výpis:

```
{"summary":10}

{'summary': 10}
```

- Vieme rovnako posielať aj požiadavky bez uchovávania cookies – namiesto *session* dáme len *requests.post(...)*
- Vieme odosielať všetky HTTP Verbs - get(), post(), put(), delete() ...

Zhrnutie

- Naprogramovanie jednoduchého api serveru
- Implementácia vlastných dekorátorov
- Odosielanie požiadaviek pomocou knižnice requests