

2.1 Python Professional Course Series: Lab & Assessment

Working with RESTful APIs: Lab & Assessment

Well done, you've reached the end of the course!

In this section, it's time to translate your Python skills and everything you've learned about RESTful APIs into some real-world projects. Specifically, we'll ask you to create:

- an HTTP server availability checker;
- a Vehicle data decoder/encoder;
- a Stock Exchange viewer;
- a Vintage Cars database.

You will also have a chance to do the quiz and the final test to see how well you've mastered the material, and check if you're prepared for the certification exam (available soon).

Completing the quiz and the final test concludes the course. Are you ready?

Lab 1

We want you to write a simple CLI (Command Line Interface) tool which can be used in order to diagnose the current status of a particular http server. The tool should accept one or two command line arguments:

- (obligatory) the address (IP or qualified domain name) of the server to be diagnosed (the diagnosis will be extremely simple, we just want to know if the server is dead or alive)
- (optional) the server's port number (any absence of the argument means that the tool should use port 80)
- use the HEAD method instead of GET — it forces the server to send the full response header but without any content; it's enough to check if the server is working properly; the rest of the request remains the same as for GET.

We also assume that:

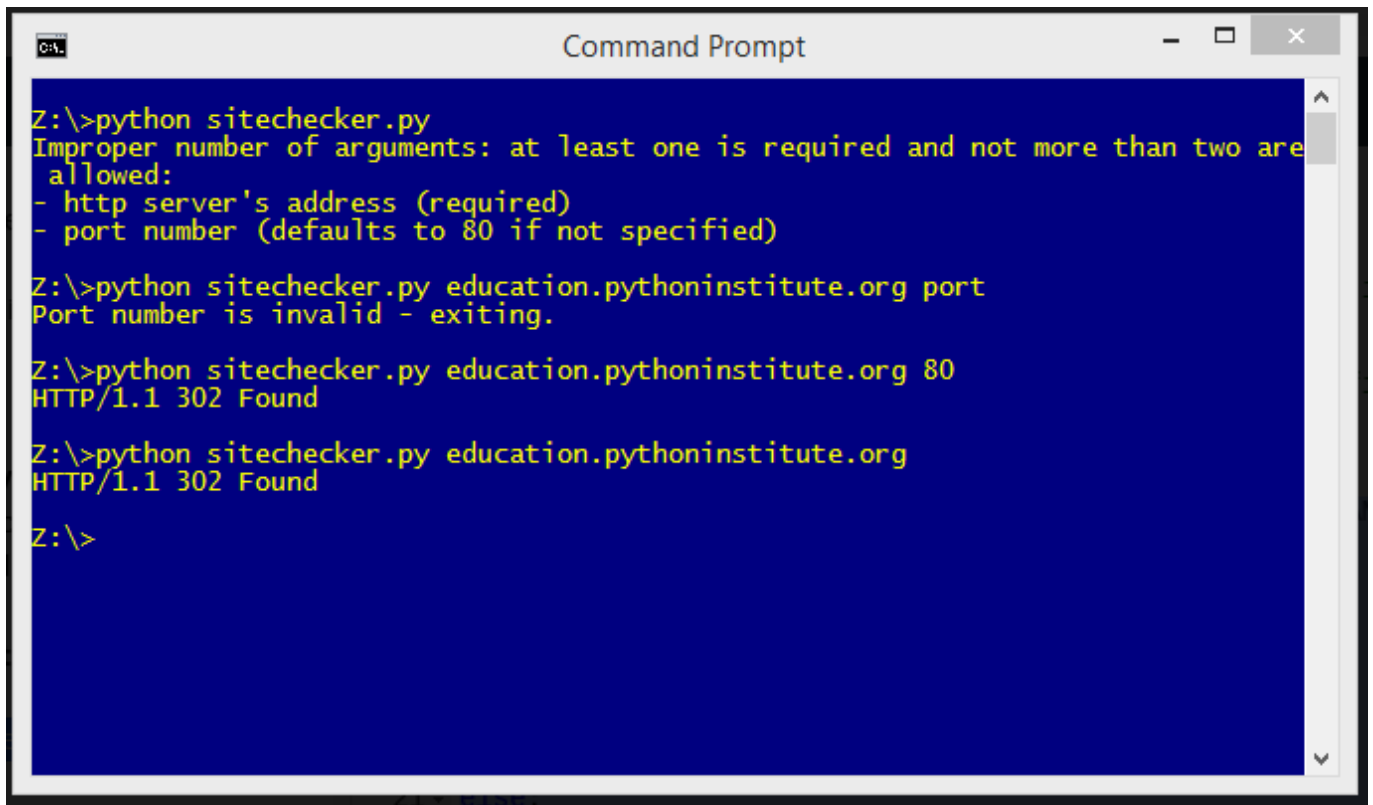
- the tool checks if it is invoked properly, and when the invocation lacks any arguments, the tool prints an error message and returns an exit code equal to 1;
- if there are two arguments in the invocation line and the second one is not an integer number in the range 1..65535, the tool prints an error message and returns an exit code equal to 2;
- if the tool experiences a timeout during connection, an error message is printed and 3 is returned as the exit code;
- if the connection fails due to any other reason, an error message appears and 4 is returned as the exit code;
- if the connection succeeds, the very first line of the server's response is printed.

Hints:

- to access command line arguments, use the argv variable from the sys module; its length is always one more than the actual number of arguments, as argv[0] stores your script's name; this means that the first argument is at argv[1] and the second at argv[2]; don't forget that the command line arguments are always strings!

- returning an exit code equal to n can be achieved by invoking the `exit(n)` function.

Assuming that the tool is placed in a source file name `sitechecker.py`, here are some real-use cases:



```
Command Prompt

Z:\>python sitechecker.py
Improper number of arguments: at least one is required and not more than two are
allowed:
- http server's address (required)
- port number (defaults to 80 if not specified)

Z:\>python sitechecker.py education.pythoninstitute.org port
Port number is invalid - exiting.

Z:\>python sitechecker.py education.pythoninstitute.org 80
HTTP/1.1 302 Found

Z:\>python sitechecker.py education.pythoninstitute.org
HTTP/1.1 302 Found

Z:\>
```

solution

```
import sys
import socket

if len(sys.argv) not in [2, 3]:
    print("Improper number of arguments: at least one is required" +
          "and not more than two are allowed:")
    print("- http server's address (required)")
    print("- port number (defaults to 80 if not specified)")
    exit(1)

addr = sys.argv[1]
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
if len(sys.argv) == 3:
    try:
        port = int(sys.argv[2])
        if not (1 <= port <= 65535):
            raise ValueError
    except ValueError:
        print("Port number is invalid - exiting.")
        exit(2)
else:
    port = 80

try:
    sock.connect((addr, port))
except socket.timeout:
```

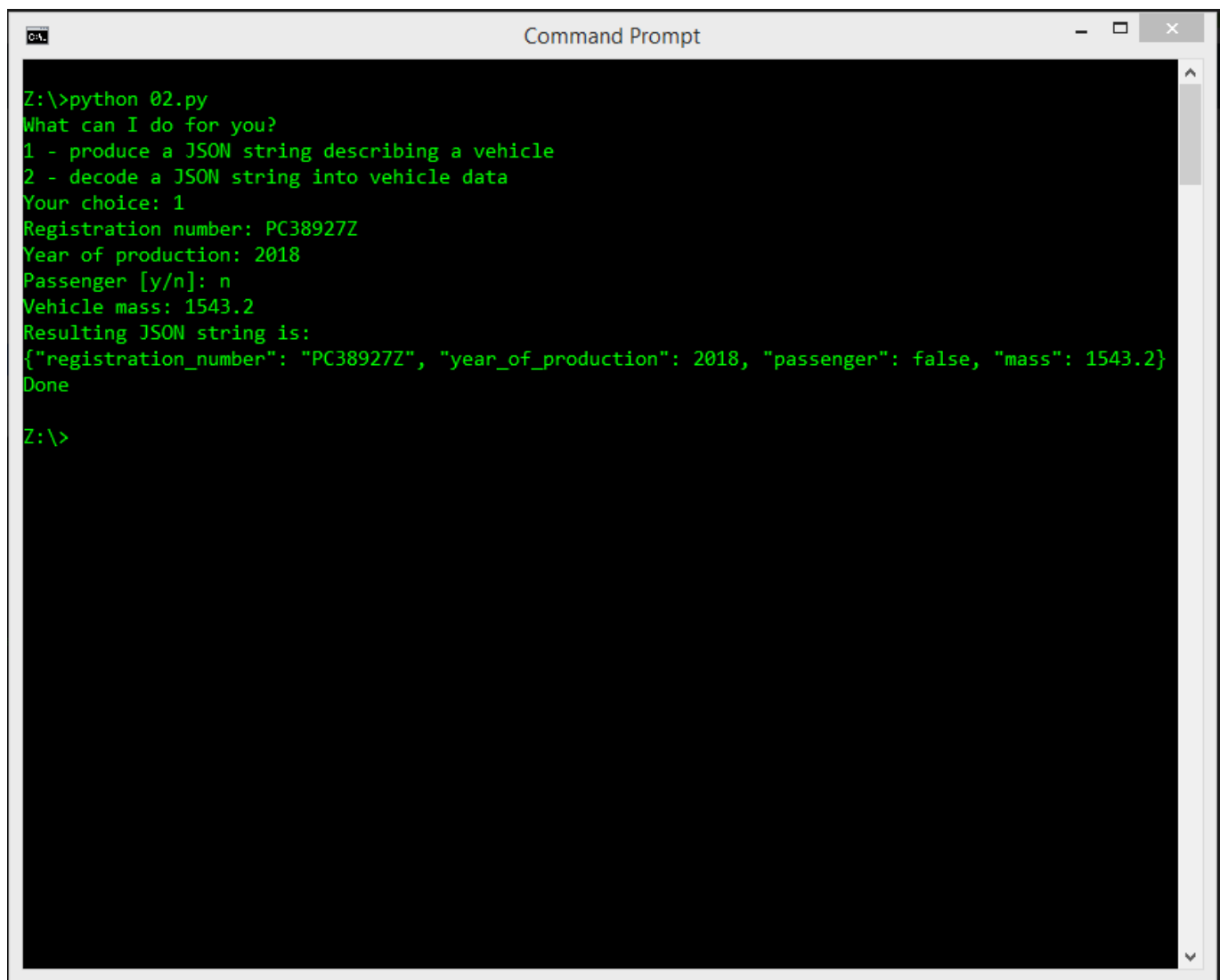
```
print("The server" + addr + "seems to be dead - sorry.")
exit(3)
except socket.gaierror:
    print("Server address" + addr + "is invalid or malformed - sorry.")
    exit(4)

request = b"HEAD / HTTP/1.0\r\nHost: " + \
    bytes(addr, "utf8") + \
    b"\r\nConnection:close\r\n\r\n"

sock.send(request)
answer = sock.recv(100).decode("utf8")
sock.shutdown(socket.SHUT_RDWR)
sock.close()
print(answer[:answer.find('\r')])
```

Lab 2

Take a look at these two screenshots. They present two different use cases of the same program:



```
Command Prompt

Z:\>python 02.py
What can I do for you?
1 - produce a JSON string describing a vehicle
2 - decode a JSON string into vehicle data
Your choice: 1
Registration number: PC38927Z
Year of production: 2018
Passenger [y/n]: n
Vehicle mass: 1543.2
Resulting JSON string is:
{"registration_number": "PC38927Z", "year_of_production": 2018, "passenger": false, "mass": 1543.2}
Done
Z:\>
```

```
Command Prompt

Z:\>python 02.py
What can I do for you?
1 - produce a JSON string describing a vehicle
2 - decode a JSON string into vehicle data
Your choice: 2
Enter vehicle JSON string: {"registration_number": "PC38927Z", "year_of_production": 2018, "passenger": false, "mass": 1543.2}
{'registration_number': 'PC38927Z', 'year_of_production': 2018, 'passenger': False, 'mass': 1543.2}
Done

Z:\>
```

Your task is to write a code which has exactly the same conversation with the user and:

1. defines a class named **Vehicle**, whose objects can carry the vehicle data shown above (the structure of the class should be deduced from the above dialog — call it «reverse engineering» if you want)
2. defines a class able to encode the Vehicle object into an equivalent JSON string;
3. defines a class able to decode the JSON string into the newly created Vehicle object.

Of course, some basic data validity checks should be done, too. We're sure you're careful enough to protect your code from reckless users.

From:
<https://matebcn.eu/> - miguel angel torres egea

Permanent link:
<https://matebcn.eu/info:ursos:pue:python-pcpp1:m4:2.1>

Last update: 19/01/2024 18:59

