packages_collection_part2

December 10, 2021

0.1 JSON

JSON is a human readable format, which eases data exchange via text files. They can be used to transfer data but can also be used for config files for example. JSON does not depend on the programming language but has its own specification. Mostly JSON behaves like a dictionary. Let's take this example JSON snippet

```
{
  "School": "Example school",
  "City": "Hamburg",
  "old": true,
  "Classes": [
    {
      "ID": "1A",
      "Teachers": [
        "Jane Doe",
        "John Smith"
      ],
      "StudCount": 22
    },
    {
      "ID": "1B",
      "Teachers": [],
      "StudCount": 0
    }
  ]
}
```

[ ]:
```
json_string="""{
  "School": "Example school",
  "City": "Hamburg",
  "old": true,
  "Classes": [
    {
      "ID": "1A",
      "Teachers": [
        "Jane Doe",
        "John Smith"
      ],
      "StudCount": 22
    },
    {
      "ID": "1B",
      "Teachers": [],
      "StudCount": 0
    }
  ]
}
"""
```

Import the package `json` and read in the JSON data with `json.loads`

[ ]:
```
import json
```

[ ]:
```
school = json.loads(json_string)
```

[ ]:
```
print(school)
```

The package returns a datastructure, which consists of (nested) python collection types, an can therefore be accessed and adjusted as well

[ ]:
```
print("School name: " + school["School"])
print("School is old: " + str(school["old"]))
print("Second class of school: " + str(school["Classes"][1]))
```

[ ]:
```
school["Classes"].append({"ID": "1C", "Teachers": ["Max Mustermann", "Maria␣
 ↪Musterfrau"]})
```

[ ]:
```
print(school)
```

[ ]:
```
json.dumps(school, sort_keys=True, indent=4)
```

[ ]:
```
with open("test.json", "w") as f:
    f.write(json.dumps(school,sort_keys=True, indent=4))
```

1 tqdm

Especially if an application has a long runtime it is a good idea to provide feedback to the user so that they can estimate how much time the current application run could still take and to indicate that the application did not freeze and is still doing something

`tqdm` allows you to insert a progress bar into your application, which is automatically printed and updated to the command line. Anything, what can be iterated through, can be wrapped with a `tqdm` object.

[ ]:
```
from tqdm import tqdm
```

[ ]:
```
from time import sleep
for i in tqdm(range(100)):
        #do stuff
        sleep(0.05)
```

2 re

This package offers a lot for handling regular expressions. Cleverly used regular expressions can become powerful if you want to search for patterns (e.g. within file names). A website to check if your regular expression does what you expect it to do, you can give regex101.com a try.

Regular expression can be used to search for a pattern or test if a string matches a given pattern.

Python regular expressions We'll have a look at a selection of interesting/important regex components. For a more complete list, have a look at link. The following list is taken from regex101.com.

Symbol	Meaning
[abc]	A single character of: a, b or c
[^abc]	A character except: a, b or c
[a-z]	A character in the range: a-z
[^a-z]	A character not in the range: a-z
[a-zA-Z]	A character in the range: a-z or A-Z
.	Any single character
a | b	Alternate - match either a or b
\s	Any whitespace character
\S	Any non-whitespace character
\d	Any digit
\D	Any non-digit
\w	Any word character
\W	Any non-word character
(...)	Capture everything enclosed
a?	Zero or one of a
a*	Zero or more of a
a+	One or more of a
a{3}	Exactly 3 of a
a{3,}	3 or more of a
a{3,6}	Between 3 and 6 of a
^	Start of string
$	End of string
\b	A word boundary
\B	Non-word boundary

[ ]:
```
import re
```

[ ]:
```
string = """
Hello the World
This is a simple test message to check if
we can use the regex machanics.

At the moment we haven't done anything, yet.
file_001.txt
file_002.txt
file_003.csv
file_040.csv
"""
test_list=["something 01: Cats$Dogs", "blabla blabla blubb 02: BMW$VW",␣
 ↪"oooommmm....ommmmm....03: Pauli$HSV", "rubbish", "04 04 04 04 04: LoL$Dota"]
```

search/match

[ ]:
```
search_result = re.search("Uetersen", string)
if not search_result:
    print("Nothing found")
```

[ ]:
```
print(type(search_result))
```

[ ]:
```
search_result = re.search("the", string)
if search_result:
    print("Found something")
    start_index = search_result.span()[0]
    end_index = search_result.span()[1]
    str_surroundings = string[max(0, start_index - 10): end_index + 10]
    print(f"Occurence was found at index {start_index}: \"{str_surroundings}\"")
```

[ ]:
```
print(type(search_result))
print(string)
```

[ ]:
```
search_result = re.match("the", string)
print(search_result)
```

findall

[ ]:
```
search_results = re.findall("Uetersen", string)
```

[ ]:
```
if len(search_results) > 0:
    help_func(search_results)
```

[ ]:
```
search_results = re.findall("the", string)
print(search_results)
```

[ ]:
```
search_results = re.findall("\w+.txt", string)
print(search_results)
search_results = re.findall("(\w+).txt", string)
print(search_results)
```

split

[ ]:
```
split_results = re.split("the", string)
print(len(split_results))
print(split_results)
```

sub

[ ]:
```
replaced_string = re.sub(" ", "--",string)
print(replaced_string)
```

match Unlike `search`, `match` is anchored to the beginning of a string. Therefore the number of possible pattern matches is lower and the operation potentially faster.

[ ]:
```
for test_entry in test_list:
    print(test_entry)
```

[ ]:
```
for file in test_list:
    match = re.match(r".*(\d{2}): (\w+)\$(\w+)", file)
    if not match:
        print(f"No match has been found for entry \"{file}\"\n")
    else:
        print(f"Entire match:\t{match.group(0)}")
        for group_index in range(1,4):
            print(f"Group {group_index}:\t{match.group(group_index)}")
        print("")
```

Note A regular expression can be pre-compiled to speed up the execution of `search` or `match`, the usage is still the same. Have a look at link for additional information.

3 subprocess

This package makes it possible to spawn new processes and retrieve their standard or error output. It becomes handy if you want to execute applications/tools/scripts, which do not provide a python API and need to be executed on the shell. There are two possibilities to spawn a new process: 1. `subprocess.run()` 2. `subprocess.Popen()`

Since the second method offers more advanced features, we will focus on the first one. The `run` method expects a list of arguments, which are executed on the shell. Each part of the command, which is individual needs to be its own list cell (e.g. flags). The call is blocking, i.e. it does not return until the command has been completely executed.

[ ]:
```
import subprocess
```

[ ]:
```
subprocess.run(["sleep", "5"])
```

[ ]:
```
output = subprocess.run(["ls", "-l", "-a"])
print(type(output))
print(output)
```

By default the output on `stdout` and `stderr` are not captured. To change this, set the `capture_output` argument to `True`.

[ ]:
```
output = subprocess.run(["ls", "-l", "-a"], capture_output=True)
print(f"{type(output)}\n")
print(f"{output}\n")
# use .decode('utf-8') to improve visuals of output, but this is optional
print(f"{output.stdout.decode('utf-8')}\n")
```

4 tempfile

Are you tired to create new temporary directories by hand? Do you forget after some time a) where the directory was and b) if an obviously deprecated temporary directory can be deleted? The `tempfile` package provides help in this regard, since it allows to create a temporary directory automatically during runtime and makes it easy to delete the directories after the program execution automatically.

`tempfile` can be used to create temporary files as well as directories, both cases are handled in a similar way. Therefore we will focus on creating temporary directories.

[ ]:
```
import tempfile
```

To create a temporary directory use `tempfile.TemporaryDirectory(suffix=None, prefix=None, dir=None, ignore_cleanup_errors=False)`. It can be used as a context manager to ease cleaning up afterwards. Setting `dir` allows to choose the place, where the directory shall be created, otherwise it is created in the default path, where your OS stores temporary files. The directory is automatically deleted after the context manager has been closed or the `TemporaryDirectory` object has been destroyed.

[ ]:
```
with tempfile.TemporaryDirectory(suffix="_end", prefix="start_") as tempd:
    print(f"Path to temporary direcoty: {tempd}")
```

If you want to keep the temporary directory after closing the program, you need to use `tempfile.mkdtemp(suffix=None, prefix=None, dir=None)`.

[ ]:
```
tempd = tempfile.mkdtemp(suffix="_end", prefix="start_")
print(f"Path to temporary direcoty: {tempd}")
```

5 pathlib

A `Path` object represents a path within your filesystem. It is independet of the operating system and simplifies porting your program between Linux and Windows (and Mac). Many methods, which expects a path, also accept `Path` objects.

[ ]:
```
from pathlib import Path
```

[ ]:
```
cwd = Path.cwd()
print(cwd)
```

[ ]:
```
for file in cwd.iterdir():
    print(file)
```

[ ]:
```
for hit in list(cwd.glob('**/*.py')):
    print(hit)
```

[ ]:
```
print(f"Exists:\t\t{cwd.exists()}")
print(f"Is absolute:\t{cwd.is_absolute()}")
print(f"Is directory:\t{cwd.is_dir()}")
```

[ ]:
```
foo = Path("hello")
bar = Path("world.txt")
baz = foo/bar
print(f"Complete path:\t\t\t{baz}")
print(f"Components of path:\t\t{baz.parts}")
print(f"Name of last component:\t\t{baz.name}")
print(f"Extensions of file:\t\t{baz.suffixes}")
print(f"File name without extension:\t{baz.stem}")
```