Edinburgh Napier University

SET09103 - Advanced Web Technologies

Topic 06 - Datastores

Dr Simon Wells

1 Overview

GOAL: Our aim in this practical is to elaborate on our simple Flask web-app up from before to increase our ability to use different features from HTTP and to take advantage of features that Flask might have. In this topic, that means specifically the following:

1. JSON,
2. SQLite3

Again, the 'Hello World' web-app from the first topic will be the foundation for the majority of the work that we do in the rest of the trimester so it is important to be comfortable with it. If you're still unsure, try it again from the beginning and consider asking a demonstrator to explain things a little.

Don't just go through the practical once and then move on. Instead, try the following approach:

1. Go through the practical once exactly as presented.
2. Go through the lab at least once more so that you are used to everything involved. We will build on the knowledge from this lab and assume it in subsequent topics so get comfortable with it before moving on.

The practicals are built around the following basic structure:

1. An introduction to give you a high-level overview.
2. Some exercise activities that you should iterate through until **you** feel confident with the material covered.
3. Some challenges to give you a starting place for play and exploration.
4. Some additional resources & documentation to help you find more information and start filling in any gaps.

IMPORTANT: If something doesn't work, or you're not sure whether something has worked, then stop for a moment, think and consider what output you are aiming for rather than just plowing on. If you move ahead before understanding something then, if you ahve made a mis-step, you will be compounding error upon error, making it more difficult to extricate yourself. A good rule of thumb when coding is to *aim to always keep things working.* So if something breaks, or doesn't give you the result you want, you should unwind things back to a *known good state.*

2 Activities: Exercises

The last topic added both HTTP methods and HTTP status codes to our collection of tools for building our own sites. So we should now be able to create a simple working web app that incorporates multiple routes to our resources, and enables us to interact with them using different methods and to return results with appropraite codes. However, we are still using quoted strings to hold the content of our Web-pages. What we need is a better mechanism to store and return Web pages as regular HTML files. Once we can do that, we'll add other kinds of static file based resources, before adding Jinja2 code to our HTML files to enable them to be dynamically generated with different content at run-time.

2.1 Getting Started

Log into your lab machine. For this practical you will need a web browser, a text editor, the windows terminal (cmd.exe), and the file manager open as you will probably use a mix of all of them at various points. You will also need uv installed as we covered in the previous lab as well as your working hello world Flask app as a starting point. Check that uv[1] works before proceeding.

So far we've ...

[1]You might need to add it to your path again.

2.2 Data Storage with SQLite3

Many web-apps are increasingly data driven. Whilst there are discussions to be had about the correct balance of static and dynamic functionality[2], it is often the case that a database is an important part of managing the data associated with a web site.

The easiest way to get started with data storage for this module is to use SQLite3 as our datastore. SQLite3 is a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine. It is also considered to be the most used database engine in the world, embedded within many apps, the default datastore on Android, and providing database functionality to support many websites. Whilst you might not have heard of SQLite, it is probably one of the most widely deployed databases on the planet It is slightly different to MySQL or the variants like MariaDB and PostGres that are commonly taught, mainly because those are client server architectures, requring you to run a separate database process that you connect to. Instead SQLite is implemented as a library that implements everything you need for creating, updating, querying and otherwise interacting with a database, but as part of your application. In our case, that means that our database will run as part of the Flask web-app process. The database itself is stored in a file on disk so it is easy to backup, move, or delete, as necessary.

What we'll do is to interact with SQLite as a database in SQLite's own tool to get to grips with it. Then we'll move to interacting with SQLite from Python. Finally, we'll integrate SQLite, Python, and Flask to create a simple database backed Flask web-app.

First we need to install SQLite3. This should work as a side-effect of installing the Python SQLite library.

```
$ uv pip install sqlite3
```

Now it's installed you can run the command line SQLite3 client by executing the following command:

```
$ sqlite3
```

If this doesn't work then we will need to get hold of an sqlite executable. On lab machines we can do this in the same way as we did for Astral UV, by downloading a package and extacting the .exe that we want to use. The easiest method is to go to the SQLite download page[3] and to select the sqlite-tools-win-x64-3500400.zip link from the Precompiled Binaries for Windows section. Note that if there is an update the the number just before .zip might change but the basic naming scheme is fairly constant. You can then extract the sqlite3.exe file from the zip and put it in the same place that you put uv, i.e. an apps folder that is on your Windows path. Note that if you are on a different operating system or on your own machine then you might need to install sqlite by following the instructions on the SQLite project page.

This gives us a command line, a bit like the Python REPL, which we can use to directly manipulate and work with SQLite databases. I recommend exploring the SQLite3 documentation[4] a little and learning about manipulating SQLite databases solely within the SQLite tools before proceeding to using other languages. The command line SQLite3 client looks like this:

```
$ sqlite3
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

[2]A completely static website, e.g. one that has been designed to be static or has been '*flattened*' can be **very** scalable. The equipment that powered Github pages was for many years very modest yet served tens of thousands of static web-pages for many open source projects. *NB.* Github pages also hosts the web pages for this module as well as hosting our public source code repository.

[3]`https://www.sqlite.org/download.html`

[4]`https://sqlite.org/index.html`

To exit the SQLite3 command line client and return to our terminal we can type .exit then hit return like so[5]:

```
$ sqlite3
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .exit
$
```

You can also find out a little more about using this tool by typing .help into the tool. We can leave the SQLite tool by typing .exit which should return us to the regular bash shell. Remember that a database can be used independently of any given software or website so it is useful to be able to interact with our database independently of Python and Flask. To do this we could use the command line client mentioned above, or else there are a lot of third party tools for managing SQLite databases[6].

2.2.1 SQLite & Python

However we probably really want to work with SQlite from a programming langauge, like Python, because this gives us a way to link our website, or really any piece of software we are writing, to an SQLite database. Using SQLite3 from Python is quite straightforward if you have worked with a relational database before. You will need to import the Python SQLite3 interface library, set a location where the SQLite database will be stored, and initialise a connection to the DB. First though you need to create a database file. I usually store database files in a "var" sub-directory of my project directory. So go ahead and create a directory called var then create a file within that directory and call it sqlite3.db like so:

```
$ mkdir var
$ type nul > var/sqlite3.db
```

NB. On Linux and MacOS you can just use *touch var/sqlite3.db* to create the db file. Keep a note of the pathname to your sqlite3.db file as you will need to tell your Python code where to find it. This file will contain all of your data but for the moment is an empty placeholder.

We can now use that empty file we just created, sqlite3.db, from Python. Let's explore this for a moment. Instead of diving right into Flask let's just use the Python REPL and manipulate our new database a little to get a feel for interacting with it. For example, start a Python shell (by typing *uv run python* in the shell) then:

```
>>> import sqlite3
>>> DB = 'var/sqlite3.db'
>>> conn = sqlite3.connect(DB)
>>> cursor = conn.cursor()
```

In this case we have performed our imports, then created a DB object that stores the location and name of our database file, e.g. sqlite3.db which is stored in the relative *var* directory. We then connected to our database file and stored the connection in the variable called *conn* before retrieving a cursor that we can use to work with our database connection.

Now we have retrieved a cursor we can use it to insert and remove data from our database as well querying the data stored in it. First though, we need to set up a table. Let's create a table to store music called *albums*.

```
>>> cursor.execute(""" CREATE TABLE albums
... (title text, artist text, release_date text, publisher text, media_type text)
... """)
<sqlite3.Cursor object at 0x1054ec570>
>>> conn.commit()
```

[5]Alternatively, you can send an end-of-file signal using the <ctrl>-d key combination which should also shut down the SQLite3 command line client. <Ctrl>-c and the related quit signal <ctrl>-c are useful tricks to have up your sleeve if you want to kill a piece of command line software that you're stuck without resorting to the nuclear option of closing the entire window and ending the session.

[6]Some of these tools are graphical tools for those of you who like that kind of thing ;)

Notice that we used a multi-line Python string which is wrapped in triple quotes. Now we can add some data to our database, lets add a couple of albums:

```
>>> cursor.execute('INSERT INTO albums VALUES ("Greatest Hits", "Roy Orbison",
    "30.11.1977", "SWRecords", "vinyl")')
<sqlite3.Cursor object at 0x1054ec570>
>>> conn.commit()
```

We can use a simple SQL query to see the contents of our fledgeling database as follows:

```
>>> for row in cursor.execute("SELECT rowid, * FROM albums ORDER BY artist"):
...     print(row)
...
```

We can also use SQL to carry out more complex queries but the following should give us a flavour of what we can do:

```
>>> sql = "SELECT * FROM albums WHERE artist=?"
>>> cursor.execute(sql, [("Roy Orbison")])
<sqlite3.Cursor object at 0x1054ec570>
>>> cursor.fetchall()
[(u'Greatest Hits', u'Roy Orbison', u'30.11.1977', u'SWRecords', u'vinyl')]
```

In this case we just defined an SQL statement which we stored in the sql variable then we executed that statement using the cursor.execute function and a supplied term "Roy Orbison" to search for. As this matched a record in the database we had a record in the list that cursor.fetchall() returns. We can also use wildcards and the SQL "like" keyword to find close but not exact matches to our query term, e.g.

```
>>> term = "orbi"
>>> cursor.execute("SELECT * FROM albums WHERE artist LIKE '%{term}%'".format(term=term))
<sqlite3.Cursor object at 0x1054ec570>
>>> cursor.fetchall()[(u'Greatest Hits', u'Roy Orbison', u'30.11.1977', u'SWRecords', u'
    vinyl')]
```

For more advanced SQL queries we should look to the SQL documentationbecause this module isn't really about SQL query construction. For now however we can move on to look at how to integrate an SQLite3 database with Flask.

2.2.2 Using SQLite3 with Flask

Now we have seen how SQLite3 works with Python as a general data storage mechanism we can now look at how, with the addition of a few simple functions, we can provide a robust mechanism for storing our web-app's data.

We'll start with defining a schema file (schema.sql) that can be used to initialise our database as follows:

```
DROP TABLE if EXISTS albums;

CREATE TABLE albums (
    title text,
    artist text,
    media_type text
);
```

This schema just deletes any existing table called 'albums' then create a new table called 'albums' with three text fields to store the 'title', 'artist' and 'media type' of the albums. We can now use the schema in a Flask file. Let's look at a basic Flask app (datastore.py) that includes a single route and some datastorage using SQLite3:

```
from flask import Flask, g
import sqlite3

app = Flask(__name__)
db_location = 'var/test.db'

```

```python
def get_db():
    db = getattr(g, 'db', None)
    if db is None:
        db = sqlite3.connect(db_location)
        g.db = db
    return db

@app.teardown_appcontext
def close_db_connection(exception):
    db = getattr(g, 'db', None)
    if db is not None:
        db.close()

def init_db():
    with app.app_context():
        db = get_db()
        with app.open_resource('schema.sql', mode='r') as f:
            db.cursor().executescript(f.read())
        db.commit()

@app.route("/")
def root():
    db = get_db()
    db.cursor().execute('insert into albums values ("American Beauty", "Grateful
        Dead", "CD")')
    db.commit()

    page = []
    page.append('<html><ul>')
    sql = "SELECT rowid, * FROM albums ORDER BY artist"
    for row in db.cursor().execute(sql):
        page.append('<li>')
        page.append(str(row))
        page.append('</li>')

    page.append('</ul></html>')
    return ''.join(page)
```

There are four things to notice here:

1. The init_db function loads our schema file and initialises a new database. We could call init_db() each time we restart the app but this would re-initialise the data each time. Instead we will create a separate external file that calls this function whenever we need it *we shall come back to this in a moment.*

2. The get_db function can be called from within a route to get access to our database connection, e.g.

   ```
   db = get_db()
   ```

   and we can then get a cursor using

   ```
   db.cursor()
   ```

3. The close_db_connection function is a *decorated* function that is automatically called to close the db connection whenever necessary. This is usually when a request end which causes that current context of the flask app to end.

4. Our root function which defined the '/' route does two things, it stores a new album in the database each time the route is accessed. Secondly, this function executes a simple SQL query to retrieve all the entries from the albums table then constructs a small HTML file to display those entries.

If we now run this flask app then we will get an error when we hit the '/' route so there is one last thing to do. The error occurs because we don't call the init_db function from anywhere. We probably only want to do this once when we deploy our web-app, otherwise we will lose all of the contents of the DB so we can create an external Python script, 'init_db.py', that will import the init_db function then execute it to initialise a new database.

```python
from datastore import init_db
init_db()
```

We can now call our script before we start our flask app for the first time to initialise the database as follows:

```
$ uv run python init_db.py
```

This should be enough SQLite3 and Flask integration to get started storing data. Obviously there is lots more to learn about the functionality that SQLite3 offers but that is an exercise for your self-directed study.

There are many data storage mechanisms and the availabilty of high-quality and performant datastores has increased greatly in recent years with the advent of the NoSQL approach. This approach suggests that there are **N**ot **O**nly **SQL** based approaches to storing data but many approaches, and the one that you choose should be based upon a sound assessment of the nature of the problem that you are tackling as well as the knowledge and experience of your development team. So, for example, there are a number of column-oriented, document oriented, graph-oriented, and key-value datastores, and many hybrids, which can be very useful when developing new web-apps. For example, when trying out various solutions to a problem it can be useful to use a schemaless datastore to hold your initial attempts at building a data model so that you do not waste time prematurely attempting to identify good database schema or relational models.

2.3 JSON

Whilst storing data in a database like SQLite is really useful, it is not the only thing that we need to be aware of when it comes to data. When form data is POST'ed to a web server the body of the request is a JSON file containing the form data. Similarly, if we want to store a small amount of structured data, without the overhead of a full database, then writing JSON to a text file can be really useful. When we retrieve data from an API then it is very lkely to be structured as JSON. So learning to work with JSON is really quite important.

JSON is the Javascript Object Notation and started out as a way to serialise, that is transform, usually save, Javascript objects in a text form. Essentially JSON strips away the object's methods, then converts the data into a textual layout which can be written to disk or streamed for consumption elsewhere. Since its creation JSON has expanded beyond its initial use in JavaScript to become an easy to use general purpose data description language. As a result, JSON is used a lot, and in many surprising places. Whenever you want to save a data with structure then JSON is a good option as it can be written to disk and read back in very easily.

JSON is built on a simple concept in which all of the contents of a JSON document are either encapsulated within a single object, using curly braces '{' and '}', or are members of a list using '[' ']'. Whether the document starts with a list or an object, their contents are basically drawn from the same set, further lists, abd objects, and actual data, in the form of strings and numbers. We'll cover that in a few moments in the JSON overview.

Whilst JSON might have started with Javascript it is used in most languages now. Python includes builtin functionality for JSON support which can be accessed using the *import json* command. It turns out that the concepts in JSON, lists and dictionaries and primitive datatypes as a way to represent encapsulated object hierarchies aligns well with similar concepts in many languages. So it is easy, for example, create an object in Javascript, serialise it to JSON, the parse that JSON into Python and have the same data now represented natively in Python for further computation.

2.3.1 JSON Overview

JSON is built around two very common data structures:

- A collection of name:value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

Because objects and lists appear in so many places, albeit under various names, they can be considered to be universal data structures and many modern programming languages support them which means that JSON aligns neatly with many modern programming languages. In JSON, the object and list are treated as follows:

- An object is an unordered set of name/value pairs. An object begins with a left brace, {, and ends with a right brace, '}'. Each name is followed by a colon ':' and the name/value pairs are separated by a comma.
- An array is an ordered collection of values. An array begins with a left bracket '[' and ends with a right bracket ']'. Values are separated by a comma.
- A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested.
- A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string.
- A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used.
- Whitespace can be inserted between any pair of tokens. Excepting a few encoding details, that completely describes the language.

2.3.2 JSON Railroad Diagrams

When you're trying to get to grips with JSON, it can help to follow railroad diagrams. These are fairly intuitive and by following the 'rail' you can see how the parts of JSON can be combined to form a valid JSON document.

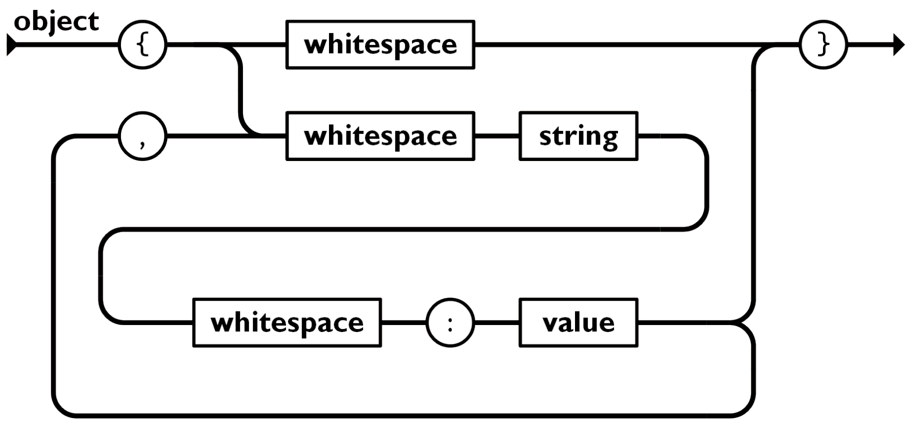

Figure 1

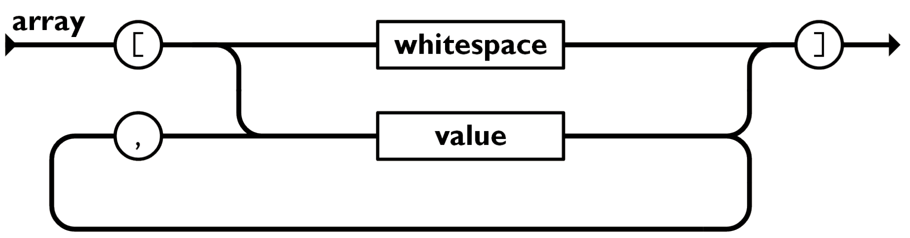

Figure 2

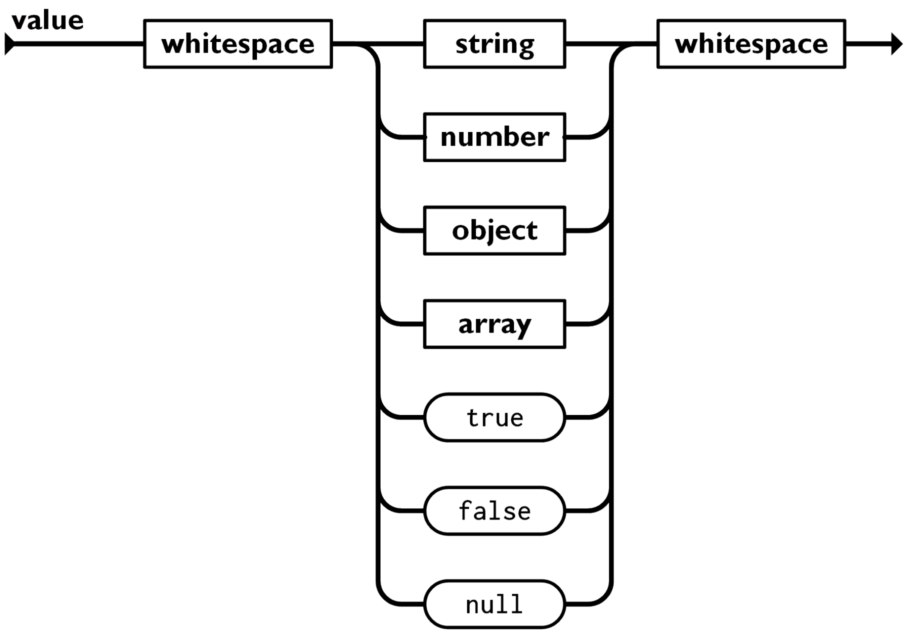

Figure 3

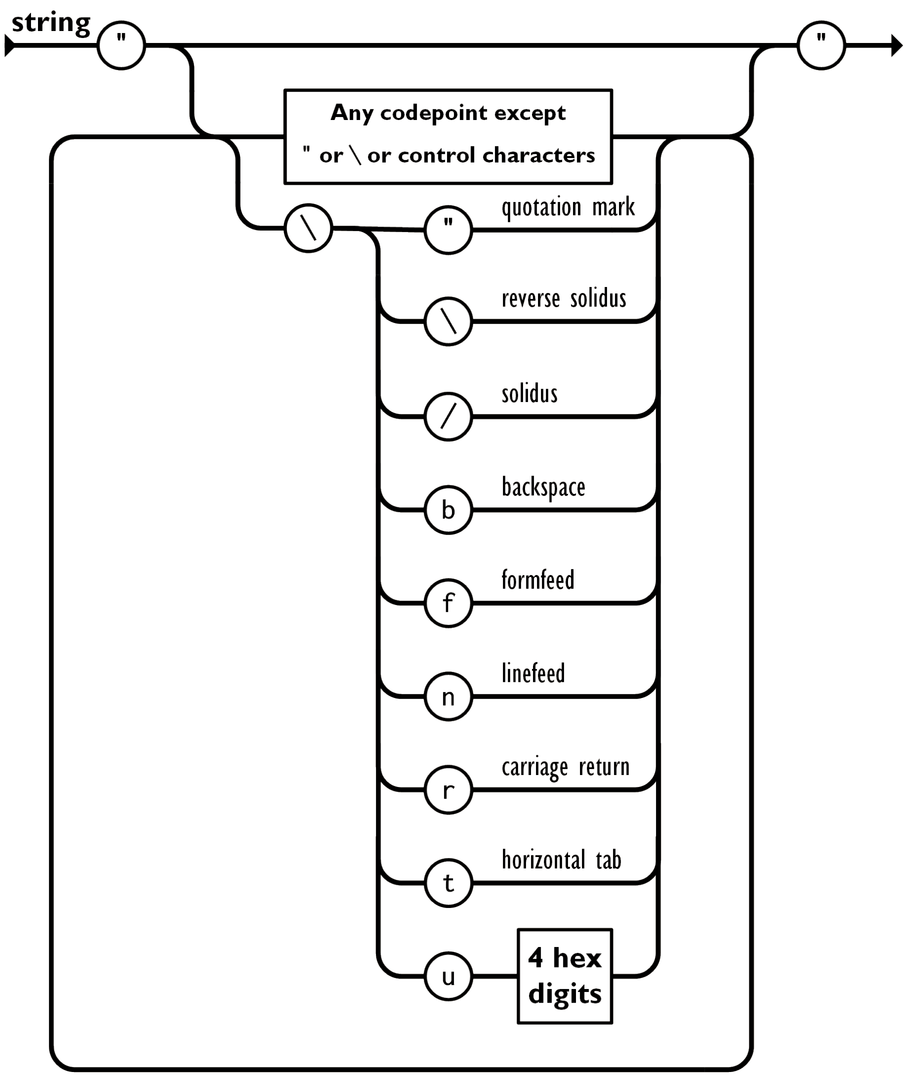

Figure 4

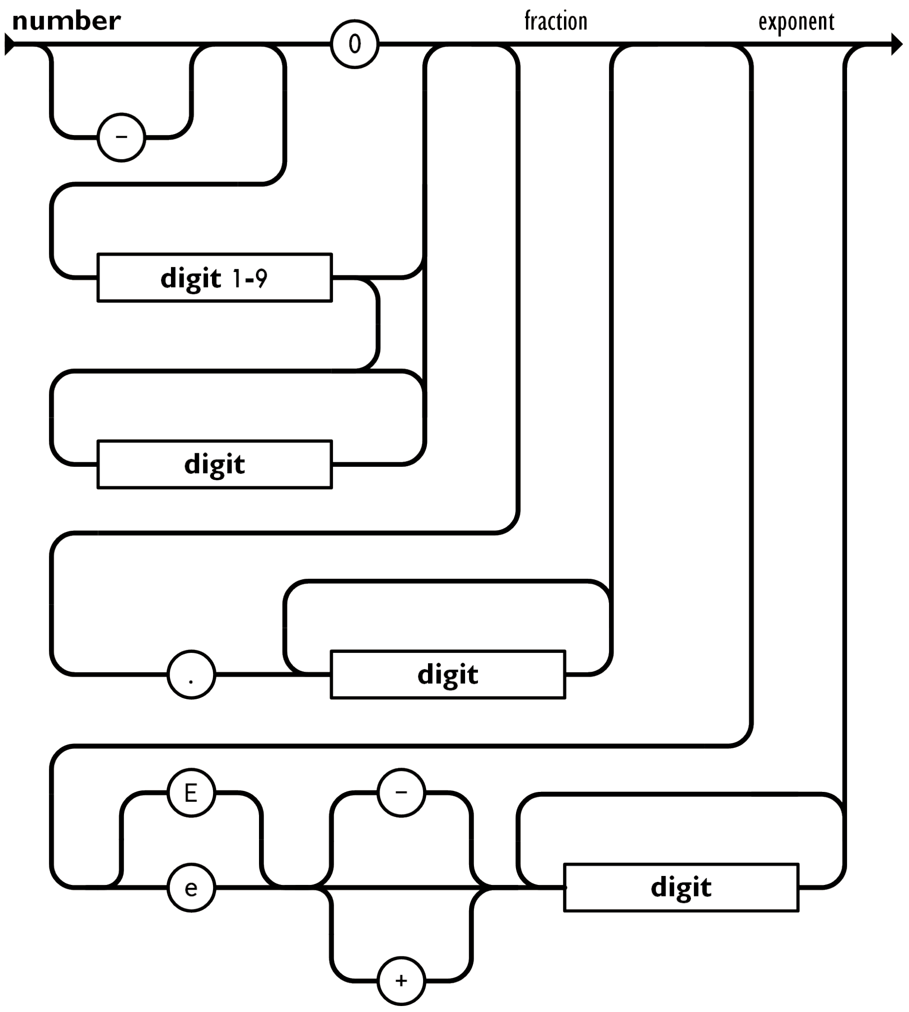

Figure 5

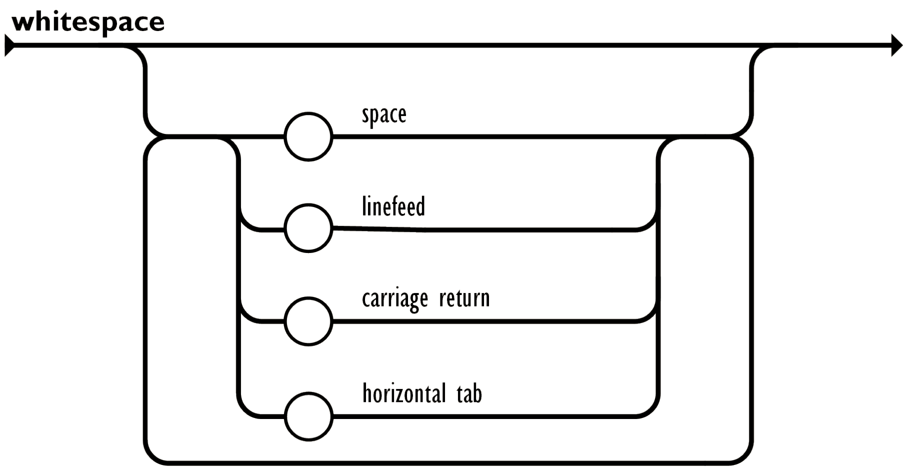

Figure 6

2.3.3 JSON & Python

The following initial exercise was performed in the python command line REPL[7] rather than in a Python script or Flask app but converting the idea of the code examples into a python script or using similar functionality in a Flask app is a useful mini-challenge to perform):

[7]remember that you can start a Python REPL (Read-Evaluate-Print loop by running *uv run python*

We can create a Python dictionary like this:

```
    >>> d = {'firstname':'simon', 'lastname':'wells'}
    >>> print(d)
    {'lastname': 'wells', 'firstname': 'simon }
```

We can import the JSON library and use it to create a JSON string representation of a Python dict as follows:

```
import json
    >>> s = json.dumps(d)
    >>> print(s)
    {"lastname": "wells", "firstname": "simon }
```

Notice that the dict and the string appear very similar in terms of output when they are printed? Internally they arent the same, one is just a string, but the other, the dict can be accessed by key, e.g.

```
    >>> print (d['lastname'])
    'wells'
```

whereas doing the same with the string gets us an error, e.g.

```
    >>> print(s[lastname])
    Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    NameError: name 'lastname' is not defined
```

We can do the reverse, i.e. get a dict back from a JSON encoded string like follows:

```
    >>> d2 = json.loads(s)
```

Now we can compare the original dict, d, with the new one, d2:

```
    >>> print(d2)
    >>> {u'lastname': u'wells', u'firstname': u'simon'}
    >>> print(d)
    {'lastname': 'wells', 'firstname': 'simon }
```

The main difference is the u which indicates that the strings in the dict are explicitly unicode encoded. We can also access our elements of the two dicts to compare them, e.g.

```
    >>> print(d['lastname'])
    'wells'
    >>> print(d2['lastname'])
    u wells'
```

So that is converting between Python dictionaries and JSON encoded strings. Now we want to serialise our string to a file then read them back in. To write our string s to the file testfile.json we can do this:

```
    >>> with open('testfile.json', 'w') as outfile:
    ...     outfile.write(s)
```

Now open testfile.json in a text editor, like vim, or just use cat to show the contents and check it is the same as the dict/string that we had earlier. I made a small edit to testfile.json in the text editor to add a new key:value pair, e.g. middlename':none so that the information loaded back in differs from the original string written out. Now we just need to read the json file back into a dict within our program so that we can manipulate it, e.g.

```
    >>> with open('testfile.json') as infile:
    ...     new_dict = json.load(infile)
    ...     infile.close()
    ...     print(new_dict)
    ...
    {u'middlename': None, u'lastname': u'wells', u'firstname': u'simon }
```

The only real challenge now is to design the right dictionary structure to capture the information you want to store, i.e. data about your collection.

NB. `https://jsonlint.com/` is really useful for testing that a bit of JSON is valid and `https://docs.python.org/2/library/json.html` documents the Python JSON library

3 Activities: Challenges

Taking any of the Flask apps that you've created so far, including those you made during earlier labs, as a starting point, or else creating something from scratch:

1. Go back over your previous challenges and sites and see if any of them might make good use of a database to store their data. A good place to start might be your quiz web-app. What if the questions and answers were stored in a database? Perhaps add a new "admin" page that can be used to manage questions in the database, adding and removing them as required.

If you're still not confident with creating a new Flask web-app from scratch[8] don't just edit and re-use your existing ones. The setting up aspect is still part of the practise. For each of the exercises, create a new project folder in your set09103 folder. You will need to create a new virtual env each time and install pip. Each project folder should have a unique name but don't call it app or flask as this will cause errors. Doing this will give you practise in setting up new Flask projects so that it becomes straightforward. We'll be doing that a lot over the coming weeks so making it less of a chore is a good idea. One tactic that can help is to create yourself a crib sheet containing the *invariant* commands that you use each time you set up a new Flask app for development. Over a short period of time, and a few repetitions of practise, you **will** get past this.

If you are confident in creating a new Flask app as needed then it is fine to reuse your existing web-apps as a skeleton for new activities and challenges. Note also that once your have set up a virtualenv it can be reused for different Flask apps, you just need to pass the name of the flask app python file to the "*uv run ...*" invocation to tell uv that you want to run a different flask app.

4 Finally...

If you've forgotten what you can do with HTML and want a little more practise or to look up the details of some tags then see the following:

- HTML Tutorial: `https://www.w3schools.com/html/`
- HTML Exercises: `https://www.w3schools.com/html/html_exercises.asp`
- HTML Element Reference: `https://www.w3schools.com/tags/`

Similarly, if you aren't yet entirely comfortable with Python then see the following:

- Python Tutorial: `https://www.w3schools.com/python/default.asp`
- Python Language Reference: `https://www.w3schools.com/python/python_reference.asp`

Use the following to find out more about HTTP verbs and status codes:

- HTTP Verbs: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods`
- HTTP Status Codes: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status`

Remember that the best way to become really familiar with a technology is to just use it. The more you use it the easier it gets. Building lots of "throw away" experimental apps, scripts, and sites is a good way to get to achieve this so let your imagination have free reign.

[8]you should be able to do the entire pprocess from the first lab topic in less than five minutes by now.