Introduction to Web Programming

In the last session, we refactored the Fibonacci sequence functionality into an object-oriented version and implemented a terminal interface.

server.py:

```
class BaseHandler:
  def handle(self, request: str):
    pass


class Server:
  def __init__(self, handlerClass):
    self.handlerClass = handlerClass

  def run(self):
    while True:
      request = input()
      self.handlerClass().handle(request)
```

fib_handle.py:

```
from fib import f
from server import BaseHandler, Server

class FibHandler(BaseHandler):
  def handle(self, request: str):
    n = int(request)
    print('f(n)=', f(n))
    pass

server = Server(FibHandler)
server.run()
```

Simple Web Server

How do we change this to a `Web` interface?

We need to replace the `Server` with an `HTTP protocol` server. Let' s first look at what an HTTP server in Python looks like.

Python's standard library provides a simple web server.

```
python -m http.server
```

Run this in the terminal.

```
$ python -m http.server
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
```

You can see the result by opening it in a browser.

This lists the current directory. When you browse this webpage, go back and check the terminal. It's quite interesting.

```
$ python -m http.server
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
::1 - - [07/Mar/2021 15:30:35] "GET / HTTP/1.1" 200 -
::1 - - [07/Mar/2021 15:30:35] code 404, message File not found
::1 - - [07/Mar/2021 15:30:35] "GET /favicon.ico HTTP/1.1" 404 -
::1 - - [07/Mar/2021 15:30:35] code 404, message File not found
::1 - - [07/Mar/2021 15:30:35] "GET /apple-touch-icon-precomposed.png HTTP/1.1" 404 -
::1 - - [07/Mar/2021 15:30:35] code 404, message File not found
::1 - - [07/Mar/2021 15:30:35] "GET /apple-touch-icon.png HTTP/1.1" 404 -
::1 - - [07/Mar/2021 15:30:38] "GET / HTTP/1.1" 200 -
```

These are the web access logs. `GET` indicates a type of data access operation for the web service, and `HTTP/1.1` indicates that the `1.1` version of the `HTTP` protocol was used.

How do we use this to create our Fibonacci sequence service? Let's look for some example code online and modify it to write a simple web server:

```
from http.server import SimpleHTTPRequestHandler, HTTPServer

class Handler(SimpleHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text')
        self.end_headers()
        self.wfile.write(bytes("hi", "utf-8"))

server = HTTPServer(("127.0.0.1", 8000), Handler)
```

```
server.serve_forever()
```

Does this look familiar? It's almost the same as our previous `Server` implementation. Note that `SimpleHTTPRequestHandler` is not a base class; there's also `BaseHTTPRequestHandler`. `SimpleHTTPRequestHandler` handles some additional content. Adding the Fibonacci sequence handling functionality to this is easy.

Here, `127.0.0.1` represents the local machine address, and `8000` represents the local port. Think of the port like a window in a house, a way to communicate with the outside world. `bytes` converts a string to bytes. `utf-8` is a string encoding method. `send_response`, `send_header`, and `end_headers` are used to output content as specified by the `HTTP` protocol so it can be understood by the browser. This way, we see `hi` in the browser.

Next, let's try getting parameters from the request.

```
from http.server import SimpleHTTPRequestHandler, HTTPServer
from fib import f
from urllib.parse import urlparse, parse_qs

class Handler(SimpleHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text')
        self.end_headers()
        parsed = urlparse(self.path)
        qs = parse_qs(parsed.query)
        result = ""
        if 'n' in qs:
            n = int(qs['n'][0])
            result = str(f(n))
        self.wfile.write(bytes(result, "utf-8"))

server = HTTPServer(("127.0.0.1", 8000), Handler)

server.serve_forever()
```

It's a bit complex, parsing some parameters here.

```
self.path=/?n=3
parsed=ParseResult(scheme='', netloc='', path='/', params='', query='n=3', fragment='')
```

```
qs={'n': ['3']}
ns=['3']
n=3
```

Recursive Improvements

Let's refactor the code a bit.

```
from http.server import SimpleHTTPRequestHandler, HTTPServer
from fib import f
from urllib.parse import urlparse, parse_qs

class Handler(SimpleHTTPRequestHandler):
    def parse_n(self, s):
        parsed = urlparse(s)
        qs = parse_qs(parsed.query)
        if 'n' in qs:
            return int(qs['n'][0])
        return None

    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text')
        self.end_headers()
        result = ""
        n = self.parse_n(self.path)
        if n is not None:
            result = str(f(n))
        self.wfile.write(bytes(result, "utf-8"))

server = HTTPServer(("127.0.0.1", 8000), Handler)

server.serve_forever()
```

We introduced the `parse_n` function to encapsulate the parsing of `n` from the request path.

Now we face an issue. If someone requests the 10000th Fibonacci number, it will take a long time to compute. If someone else requests it again, it will again take a long time. How do we improve the efficiency of this `Web service`?

Note that if `n` is the same, `f(n)` is always the same. We conducted some experiments.

```
127.0.0.1 - - [10/Mar/2021 00:33:01] "GET /?n=1000 HTTP/1.1" 200 -
----------------------------------------
Exception occurred during processing of request from ('127.0.0.1', 50783)
Traceback (most recent call last):
    ...
    if v[n] != -1:
IndexError: list index out of range
```

It turns out the array wasn't large enough, so we changed the size of the `v` array to 10000.

```
v = [-1] * 10000
```

However, when `n` is `2000`, a recursion depth exceeded error occurs:

```
127.0.0.1 - - [10/Mar/2021 00:34:00] "GET /?n=2000 HTTP/1.1" 200 -
----------------------------------------
Exception occurred during processing of request from ('127.0.0.1', 50821)
Traceback (most recent call last):
    ...
    if v[n] != -1:
RecursionError: maximum recursion depth exceeded in comparison
```

Still, it's quite fast.

Why? Because `f(1)` to `f(1000)` only need to be computed once. This means that when calculating `f(1000)`, the `+` operation might only be executed around 1000 times. We know that Python's recursion depth limit is around 1000. This means we can optimize the program this way: if we need to calculate `2000`, we first calculate `1000`. No, this might still cause a recursion depth exceeded error. If we need to calculate 2000, we first calculate 1200. If we need to calculate 1200, we first calculate 400.

After calculating 400 and 1200, calculating 2000 would have a recursion depth of about 800, avoiding the recursion depth exceeded error.

```
v = [-1] * 1000000

def fplus(n):
    if n > 800:
        fplus(n - 800)
        return f(n)
    else:
```

```
        return f(n)


def f(n):
    if v[n] != -1:
        return v[n]
    else:
        if n < 2:
            a = n
        else:
            a = f(n - 1) + f(n - 2)
        v[n] = a
        return v[n]
```

We added the `fplus` function

.

But one might wonder, what if `fplus` is recursively called `1000` times? 1000 * 800 = 800000. When I set `n` to 800000, a recursion depth error occurred again. Further testing showed things were more complex. However, after this optimization, calculating 2000 is very easy.

File Read and Write

It seems we have diverged a bit. Let' s return to the topic of web development. If the first request is for `f(400)`, and the second request is for `f(600)`, we can use the `v` array values from the first request for the second. However, if we restart the program, we lose the cached values. If Fibonacci sequence calculations are slow, especially without the `v` array, there would be a lot of redundant calculations. We want to save the hard-won results.

This introduces the concept of `caching`. Here, the `v` array is a cache, but it only exists during the program' s lifecycle. When the program closes, it disappears. What do we do?

We can save it to a file.

How do we save the `v` array to a file?

```
0 0
1 1
2 1
3 2
4 3
```

...

We can save the `v` array like this. Each line is saved as `n f(n)`. Since `n` grows naturally, we could also save just the `f(n)` values.

```
0
1
1
2
3
...
```

Let's try it.

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

# Open and read the file after appending:
f = open("demofile2.txt", "r")
print(f.read())
```

The second parameter of `open` can be `a`, which appends to the file, or `w`, which overwrites it.

```
file = open('fib_v', 'a')
file.write('hi')
file.close()
```

Run it, and indeed a file named `fib_v` appears.

`fib_v`:

```
hi
```

When we run it again, it becomes:

```
hihi
```

How do we add a newline?

```
file = open('fib_v', 'a')
file.write('hi\n')
file.close()
```

When we print it once, it shows `hihihi` with no visible newline. However, printing again shows the newline.

The first print already added the newline character, but it's at the end, so it's not visible.

How do we read it?

```
file = open('fib_v', 'r')
print(file.read())
```

```
$ python fib.py
hihihi
hi
```

Now, let's modify our Fibonacci program.

```
v = [-1] * 1000000

def read():
    file = open('fib_v', 'r')
    s = file.read()
    if len(s) > 0:
        lines = s.split('\n')
        for i in range(len(lines)):
            v[i] = int(lines[i])

def save():
    file = open('fib_v', 'w')
    s = ''
    start = True
    for vv in v:
        if vv == -1:
            break
        if not start:
            s += '\n'
        start = False
        s += str(vv)
    file.write(s)
    file.close()

def fcache(n):
    x = fplus(n)
```

```
    save()
    return x

def fplus(n):
    if n > 800:
        fplus(n - 800)
        return f(n)
    else:
        return f(n)

def f(n):
    if v[n] != -1:
        return v[n]
    else:
        if n < 2:
            a = n
        else:
            a = f(n - 1) + f(n - 2)
        v[n] = a
        return v[n]

read()
fcache(10)
save()
```

Finally, we have the program. After running the program, the `fib_v` file looks like this:

`fib_v`:

```
0
1
1
2
3
5
8
13
21
```

```
34
55
```

You can see the parsing is a bit cumbersome. The `\n` is the newline character. Is there a simpler, more uniform way to parse it? People invented the `JSON` data format.

JSON

JSON stands for `JavaScript Object Notation`. Here is an example of `JSON`:

```
{"name":"John", "age":31, "city":"New York"}
```

This represents a mapping.

JSON has basic elements:

1. Numbers or strings
2. Lists
3. Mappings

These basic elements can be nested arbitrarily. For example, a list can contain lists, and a mapping can contain lists, and so on.

```
{
  "name": "John",
  "age": 30,
  "cars": ["Ford", "BMW", "Fiat"]
}
```

Writing it in one line or making it look better doesn't change its meaning. We can imagine their computational graph. Spaces do not affect their computational graph.

Next, let's convert the `v` array to a JSON formatted string.

```
import json

v = [-1] * 1000000

def fplus(n):
    if n > 800:
        fplus(n - 800)
        return f(n)
    else:
```

```
        return f(n)

def f(n):
    if v[n] != -1:
        return v[n]
    else:
        if n < 2:
            a = n
        else:
            a = f(n - 1) + f(n - 2)
        v[n] = a
        return v[n]

fplus(100)
s = json.dumps(v)
file = open('fib_j', 'w')
file.write(s)
file.close()
```

When we write it this way, we get an error. `TypeError: dump() missing 1 required positional argument: 'fp'`. In `vscode`, you can see the function definition like this.

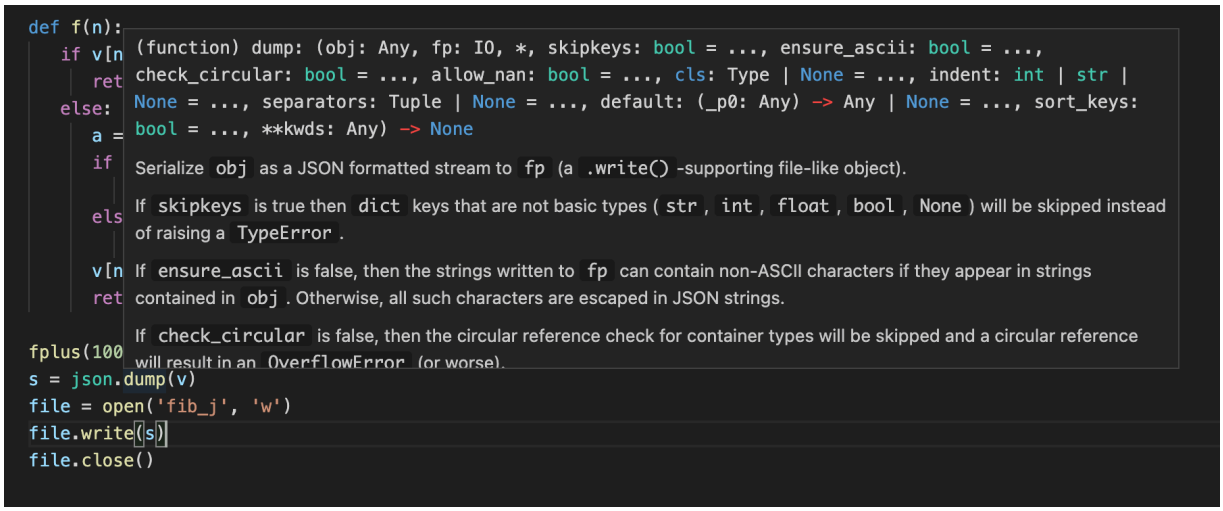

Figure 1: json

You can hover over `dump` with your mouse. Very convenient, right?

```
fplus(10)
```

```
file = open('fib_j', 'w')
json.dump(v, file)
file.close()
```

I changed the computation to 10 because there were too many numbers at 100. It turns out the second parameter of `dump` should be the `file` object.

Now we see the file:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, -1, -1, -1]
```

Notice that many `-1` values are omitted at the end.

```
def read():
    file = open('fib_j', 'r')
    s = file.read()
    sv = json.loads(s)
    for i in range(len(sv)):
        if sv[i] != -1:
            v[i] = sv[i]
def save():
    file = open('fib_j', 'w')
    json.dump(v, file)
    file.close()

read()

for vv in v:
    if vv != -1:
        print(vv)
```

When we do this, it prints:

```
0
1
1
2
3
5
8
13
```

```
21
34
55
```

Now let's check these functions together:

```
def read():
    file = open('fib_j', 'r')
    s = file.read()
    sv = json.loads(s)
    for i in range(len(sv)):
        v[i] = sv[i]


def save():
    sv = []
    for i in range(len(v)):
        if v[i] != -1:
            sv.append(v[i])
        else:
            break
    file = open('fib_j', 'w')
    json.dump(sv, file)
    file.close()


read()
fplus(100)
save()
```

Checking the file shows the correct values are saved neatly.

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711


1, 4052739537881, 6557470319842, 10610209857723, 17167680177565, 27777890035288, 44945570212853, 727234
```

Database

What if the data is large and complex? Using files becomes slow and cumbersome. This introduces the concept of a `database`, which is like a programmable `Excel` sheet that can easily be manipulated with code.

From the official documentation, here's an example.

```
import sqlite3
con = sqlite3.connect('example.db')

cur = con.cursor()

# Create table
cur.execute('''CREATE TABLE stocks
               (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
cur.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
con.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
con.close()

for row in cur.execute('SELECT * FROM stocks ORDER BY price'):
    print(row)
```

`cursor` represents a cursor, similar to a pointer. The above code connects to a database, creates a table, inserts data, commits changes, and closes the connection. The last example shows how to query data.

```
import sqlite3

v = [-1] * 1000000

def create_table(cur: sqlite3.Connection):
    cur.execute('CREATE TABLE vs(v text)')

def read():
    pass

def save():
    con = sqlite3.connect('fib.db')
    cur = con.cursor()
    create_table(cur)
```

```
    for vv in v:
        if vv != -1:
            cur.execute('INSERT INTO vs VALUES(' + str(vv) + ')')
        else:
            break
    con.commit()
    con.close()

fplus(10)
save()
```

Let's try running it.

I already have `sqlite3` installed on my computer.

```
$ sqlite3
SQLite version 3.32.3 2020-06-18 14:16:19
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.

sqlite> .help
.auth ON|OFF             Show authorizer callbacks
.backup ?DB? FILE        Backup DB (default "main") to FILE
.bail on|off             Stop after hitting an error.  Default OFF
.binary on|off           Turn binary output on or off.  Default OFF
.cd DIRECTORY            Change the working directory to DIRECTORY
.changes on|off          Show number of rows changed by SQL
.check GLOB              Fail if output since .testcase does not match
.clone NEWDB             Clone data into NEWDB from the existing database
.databases               List names and files of attached databases
.dbconfig ?op? ?val?     List or change sqlite3_db_config() options
.dbinfo ?DB?             Show status information about the database
.dump ?TABLE?            Render database content as SQL
.echo on|off             Turn command echo on or off
.eqp on|off|full|...     Enable or disable automatic EXPLAIN QUERY PLAN
.excel                   Display the output of next command in spreadsheet
.exit ?CODE?             Exit this program with return-code CODE
.expert                  EXPERIMENTAL. Suggest indexes for queries
```

```
.explain ?on|off|auto?   Change the EXPLAIN formatting mode.  Default: auto
.filectrl CMD ...        Run various sqlite3_file_control() operations
.fullschema ?--indent?   Show schema and the content of sqlite_stat tables
.headers on|off          Turn display of headers on or off
.help ?-all? ?PATTERN?   Show help text for PATTERN
.import FILE TABLE       Import data from FILE into TABLE
.imposter INDEX TABLE    Create imposter table TABLE on index INDEX
.indexes ?TABLE?         Show names of indexes
.limit ?LIMIT? ?VAL?     Display or change the value of an SQLITE_LIMIT
.lint OPTIONS            Report potential schema issues.
.log FILE|off            Turn logging on or off.  FILE can be stderr/stdout
.mode MODE ?TABLE?       Set output mode
.nullvalue STRING        Use STRING in place of NULL values
.once ?OPTIONS? ?FILE?   Output for the next SQL command only to FILE
.open ?OPTIONS? ?FILE?   Close existing database and reopen FILE
.output ?FILE?           Send output to FILE or stdout if FILE is omitted
.parameter CMD ...       Manage SQL parameter bindings
.print STRING...         Print literal STRING
.progress N              Invoke progress handler after every N opcodes
.prompt MAIN CONTINUE    Replace the standard prompts
.quit                    Exit this program
.read FILE               Read input from FILE
.recover                 Recover as much data as possible from corrupt db.
.restore ?DB? FILE       Restore content of DB (default "main") from FILE
.save FILE               Write in-memory database into FILE
.scanstats on|off        Turn sqlite3_stmt_scanstatus() metrics on or off
.schema ?PATTERN?        Show the CREATE statements matching PATTERN
.selftest ?OPTIONS?      Run tests defined in the SELFTEST table
.separator COL ?ROW?     Change the column and row separators
.session ?NAME? CMD ...  Create or control sessions
.sha3sum ...             Compute a SHA3 hash of database content
.shell CMD ARGS...       Run CMD ARGS... in a system shell
.show                    Show the current values for various settings
.stats ?on|off?          Show stats or turn stats on or off
.system CMD ARGS...      Run CMD ARGS... in a system shell
.tables ?TABLE?          List names of tables matching LIKE pattern TABLE
```

```
.testcase NAME           Begin redirecting output to 'testcase-out.txt'
.testctrl CMD ...        Run various sqlite3_test_control() operations
.timeout MS              Try opening locked tables for MS milliseconds
.timer on|off            Turn SQL timer on or off
.trace ?OPTIONS?         Output each SQL statement as it is run
.vfsinfo ?AUX?           Information about the top-level VFS
.vfslist                 List all available VFSes
.vfsname ?AUX?           Print the name of the VFS stack
.width NUM1 NUM2 ...     Set column widths for "column" mode
```

You can see many commands. `.quit` means exit.

If you don't have it, you can download it from the official website, or run `brew install sqlite` to install it.

```
$ sqlite3 fib.db
```

```
sqlite> .schema
CREATE TABLE vs(v text);
```

Initially, I thought it was like `MySQL` where you use `show tables`. Later, I found out in `SQLite`, it's like this. `MySQL` is another database we will learn about in the future.

```
sqlite> select * from vs;
0
1
1
2
3
5
8
13
21
34
55
```

Indeed, we correctly wrote the data. Note we used `text` because our numbers are large, and the database's integer type might not store them.

```
import sqlite3
```

```
v = [-1] * 1000000

def fplus(n):
    if n > 800:
        fplus(n - 800)
        return f(n)
    else:
        return f(n)

def f(n):
    if v[n] != -1:
        return v[n]
    else:
        if n < 2:
            a = n
        else:
            a = f(n - 1) + f(n - 2)
        v[n] = a
        return v[n]

def create_table(cur: sqlite3.Connection):
    cur.execute('CREATE TABLE IF NOT EXISTS vs(v text)')

def read():
    con = sqlite3.connect('fib.db')
    cur = con.cursor()
    create_table(cur)
    i = 0
    for row in cur.execute('SELECT * from vs'):
        v[i] = int(row[0])
        i += 1
    con.close()

def save():
    con = sqlite3.connect('fib.db')
    cur = con.cursor()
```

```
    create_table(cur)
    for vv in v:
        if vv != -1:
            cur

.execute("INSERT INTO vs VALUES('" + str(vv) + "')")
        else:
            break
    con.commit()
    con.close()

read()
for i in range(10):
    print(v[i])
```

We updated the `read` function. But when we ran it, we got an error.

```
$ python fib_db.py
  ...
  File "fib_db.py", line 27, in create_table
    cur.execute('CREATE TABLE vs(v text)')
sqlite3.OperationalError: table vs already exists
```

We can't create the table again; it already exists. We modify the syntax slightly.

```
def create_table(cur: sqlite3.Connection):
    cur.execute('CREATE TABLE IF NOT EXISTS vs(v text)')
```

But we encountered another error.

```
    v[i] = int(row[0])
ValueError: invalid literal for int() with base 10: '1.22001604151219e+19'
```

Why? Because numbers are saved as strings. We modify the code to read the values correctly.

```
def read():
    con = sqlite3.connect('fib.db')
    cur = con.cursor()
    create_table(cur)
    i = 0
    for row in cur.execute('SELECT * from vs'):
```

```
        v[i] = int(row[0])
        i += 1
    con.close()
```

We realized `i` wasn't incrementing.

```
    for row in cur.execute('SELECT * from vs'):
        v[i] = int(row[0])
        i += 1
```

Now it's correct.

```
0
1
1
2
3
5
8
13
21
34
```

Note that when the numbers are large, they are saved like this:

```
4660046610375530309
7540113804746346429
1.22001604151219e+19
1.97402742198682e+19
3.19404346349901e+19
```

Running it again shows this:

```
$ python fib_db.py
Traceback (most recent call last):
  File "fib_db.py", line 35, in read
    v[i] = int(row[0])
ValueError: invalid literal for int() with base 10: '1.22001604151219e+19'
```

We modify:

```
cur.execute("INSERT INTO vs VALUES('" + str(vv) + "')")
```

Notice we use double quotes for `INSERT` and enclose our number strings in quotes. If we had written this earlier, the database would have treated our string as a number. Now, with quotes, it treats it as a string.

It works correctly. But how do we clear the previous erroneous data?

```
$ sqlite3 fib.db
sqlite> delete from vs;
```

Next, we can try other operations. We have shown examples of `insert`, `delete`, and `select`.

Practice

- Students should explore the above steps similarly.