Web プログラミング入門

前回、私たちはフィボナッチ数列の機能をオブジェクト指向のバージョンに書き換え、ターミナルインターフェースを実装しました。

`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
```

このコードは、FibHandler というクラスを定義しています。このクラスは BaseHandler を継承しており、handle メソッドを持っています。handle メソッドは、文字列型の request を引数として受け取り、それを整数に変換して n に代入します。その後、f(n) の結果を出力します。ただし、f(n) の具体的な実装はこのコードには含まれていません。最後に pass が記述されていますが、これは何も実行しないことを示しています。

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

シンプルな Web サーバー

では、Web インターフェースに変更するにはどうすればよいでしょうか。

上の Server を HTTP プロトコルの Server に置き換えれば良いのです。まず、Python における HTTP サーバーがどのようなものか見てみましょう。

Python の標準ライブラリには、ウェブサーバーが提供されています。

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

このコマンドは、Python の組み込み HTTP サーバーを起動するものです。デフォルトでは、現在のディレクトリをルートとして、ポート 8000 でサーバーが立ち上がります。ブラウザで http://localhost:8000 にアクセスすると、ディレクトリ内のファイルを閲覧できます。

ターミナルで実行します。

```
$ python -m http.server
HTTPを::のポート8000で提供中 (http://[::]:8000/) ...
```

ブラウザで開くと効果を確認できます。

これで現在のディレクトリがリストアップされました。次に、このウェブページを閲覧している間に、ターミナルに戻ってみてください。すると、面白いことが起こります。

```
$ python -m http.server
HTTPを::のポート8000で提供中 (http://[::]:8000/) ...
::1 - - [07/Mar/2021 15:30:35] "GET / HTTP/1.1" 200 -
::1 - - [07/Mar/2021 15:30:35] コード404, メッセージ ファイルが見つかりません
::1 - - [07/Mar/2021 15:30:35] "GET /favicon.ico HTTP/1.1" 404 -
::1 - - [07/Mar/2021 15:30:35] コード404, メッセージ ファイルが見つかりません
::1 - - [07/Mar/2021 15:30:35] "GET /apple-touch-icon-precomposed.png HTTP/1.1" 404 -
::1 - - [07/Mar/2021 15:30:35] コード404, メッセージ ファイルが見つかりません
::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 -
```

これはウェブアクセスログです。その中で、GET はウェブサービスにおけるデータアクセス操作の一種を表しています。HTTP/1.1 は、HTTP の 1.1 バージョンのプロトコルが使用されていることを示しています。

それを使って私たちのフィボナッチ数列サービスを作成する方法です。まず、オンラインでサンプルコードを探し、少し手を加えて、最もシンプルな Web サーバーを作成します：

```
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()
```

これらは見覚えがあるものばかりですね。ほとんど上で使用した `Server` と同じです。`SimpleHTTPRequestHandler` が基本クラスではなく、`BaseHTTPRequestHandler` というものがあることに気づきました。`SimpleHTTPRequestHandler` は比較的、いくつかの内容を追加処理しています。これらにフィボナッチ数列の処理機能を加えるのは簡単です。

ここでの `127.0.0.1` はローカルマシンのアドレスを表し、`8000` はローカルマシンのポートを表します。ポートはどのように理解すればよいでしょうか。家の窓のようなもので、家と外界がコミュニケーションを取るための窓口です。`bytes` は文字列をバイトに変換することを意味します。`utf-8` は文字列のエンコード方式の一つです。`send_response`、`send_header`、`end_headers` はすべて、`HTTP` プロトコルで規定された内容を出力するためのもので、ブラウザが理解できるようにするためのものです。これにより、ウェブページに `hi` と表示されます。

次に、リクエストからパラメータを取得してみましょう。

```
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 len(qs) > 0:
            ns = qs[0]
            if len(ns) > 0:
                n = int(ns)
                result = str(f(n))
        self.wfile.write(bytes(result, "utf-8"))
```

このコードは、HTTP GET リクエストを処理するためのシンプルなハンドラクラスです。以下にその動作を説明します：

1. `do_GET` メソッドは、GET リクエストが来たときに呼び出されます。
2. `send_response(200)` で、HTTP ステータスコード 200（OK）を返します。
3. `send_header` で、レスポンスのコンテンツタイプを `text` に設定します。
4. `end_headers` でヘッダーの送信を終了します。
5. `urlparse` を使って、リクエストのパスを解析します。
6. `parse_qs` を使って、クエリ文字列を解析します。
7. `result` 変数を空の文字列で初期化します。
8. クエリ文字列が存在する場合、最初のクエリパラメータを取得し、それを整数に変換します。
9. その整数を関数 `f` に渡し、結果を文字列に変換して `result` に格納します。
10. 最後に、`self.wfile.write` を使って、結果をクライアントに送信します。

このコードは、特定の関数 `f` に基づいてクエリパラメータを処理し、その結果を返すシンプルな HTTP サーバーの一部として使用されることが想定されています。

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

server.serve_forever()
```

少し複雑ですね。ここではいくつかのパラメータを解析しています。

```
self.path=/?n=3
parsed=ParseResult(scheme='', netloc='', path='/', params='', query='n=3', fragment='')
qs={'n': ['3']}
ns=['3']
n=3
```

上記のコードは、URL のクエリパラメータを解析する過程を示しています。具体的には、`self.path` に指定された URL のクエリ部分を解析し、`n` というパラメータの値を取得しています。以下に各ステップの説明を日本語で示します。

- `self.path=/?n=3`: URL のパスとクエリパラメータが`/?n=3` として指定されています。
- `parsed=ParseResult(scheme='', netloc='', path='/', params='', query='n=3', fragment='')`: URL が解析され、各部分が分割されています。`query='n=3'`の部分がクエリパラメータを示しています。
- `qs={'n': ['3']}`: クエリパラメータが辞書形式で解析され、`n` というキーに対して `['3']` という値がリスト形式で格納されています。
- `ns=['3']`: `n` の値がリストとして抽出されています。
- `n=3`: 最終的に、`n` の値が `3` として取得されています。

このコードは、URL から特定のクエリパラメータを抽出し、その値を利用するための一連の処理を示しています。

再帰の応用

少しコードをリファクタリングしてみましょう。

```
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 len(qs) > 0:
        ns = qs['n']
        if len(ns) > 0:
          n = int(ns[0])
          return n
      return None

    def do_GET(self):
```

```
        self.send_response(200)
        self.send_header('Content-type', 'text')
        self.end_headers()
```

上記のコードは、URL からクエリパラメータ `n` を解析し、その値を整数として返す `parse_n` メソッドと、HTTP GET リクエストを処理する `do_GET` メソッドを定義しています。`parse_n` メソッドは、URL が与えられると、そのクエリパラメータを解析し、`n` の値を返します。`do_GET` メソッドは、HTTP レスポンスのステータスコード 200 を送信し、コンテンツタイプをテキストとして設定します。

```
result = ""
n = self.parse_n(self.path)
if n is not None:
    result = str(f(n))

self.wfile.write(bytes(result, "utf-8"))
self.wfile.write(bytes(result, "utf-8"))
```

このコードは、`self.parse_n(self.path)` を使ってパスから数値 `n` を解析し、それが `None` でない場合に `f(n)` を計算して結果を文字列に変換しています。その後、その結果を 2 回 `self.wfile.write` を使って UTF-8 エンコードで書き込んでいます。

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

server.serve_forever()
```

`parse_n` 関数を導入して、リクエストパスから解析された `n` をカプセル化します。

现在程序有这样的问题。小王请求了斐波那契数列的第 10000 位，过了一些天，小明又请求了斐波那契数列的第 10000 位。两次，小王和小明都等待了半天，才得到结果。我们该如何提高这个 Web 服务的效率呢。

解决方案

1. **缓存结果**：
    - 使用缓存机制（如 Redis 或 Memcached）来存储已经计算过的斐波那契数列的结果。这样，当有相同的请求时，可以直接从缓存中获取结果，而不需要重新计算。

2. **预计算**：
    - 如果知道某些斐波那契数列的值会被频繁请求，可以预先计算这些值并存储在数据库中。这样，当请求到来时，可以直接从数据库中获取结果。
3. **优化算法**：
    - 使用更高效的算法来计算斐波那契数列。例如，使用矩阵快速幂算法或动态规划算法来减少计算时间。
4. **异步处理**：
    - 将计算任务放入消息队列中，由后台任务异步处理。这样，用户请求不会阻塞，可以立即返回一个任务 ID，用户可以通过这个 ID 来查询计算结果。
5. **分布式计算**：
    - 如果计算量非常大，可以考虑使用分布式计算框架（如 Hadoop 或 Spark）来并行计算斐波那契数列。

代码示例

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# 使用缓存后的斐波那契函数
result = fibonacci(10000)
```

通过以上方法，可以显著提高 `Web` 服务的效率，减少用户等待时间。

`n` が同じ場合、`f(n)` の値は常に同じであることに気づきました。私たちはいくつかの実験を行いました。

```
127.0.0.1 - - [10/Mar/2021 00:33:01] "GET /?n=1000 HTTP/1.1" 200 -
----------------------------------------
リクエストの処理中に例外が発生しました。送信元: ('127.0.0.1', 50783)
トレースバック（最近の呼び出しを最後に）:
```

```
    ...
    if v[n] != -1:
IndexError: リストのインデックスが範囲外です
```

元の配列が十分に大きくなかったので、v 配列を 10000 に変更しましょう。

```
v = []
for x in range(10000):
   v.append(-1)
```

このコードは、空のリスト v を作成し、0 から 9999 までの範囲でループを回して、リスト v に -1 を 10000 回追加しています。結果として、v は -1 が 10000 個含まれたリストになります。

しかし、n が 2000 の場合、再帰の深さがオーバーフローするエラーが発生しました：

```
127.0.0.1 - - [10/Mar/2021 00:34:00] "GET /?n=2000 HTTP/1.1" 200 -
----------------------------------------
リクエストの処理中に例外が発生しました。送信元: ('127.0.0.1', 50821)
トレースバック（直近の呼び出し最後）:
    ...
    if v[n] != -1:
RecursionError: 比較中に最大再帰深度を超えました
```

しかし、これらすべてはかなり迅速に進みました。

なぜなら、f(1) から f(1000) まで、それぞれ一度だけ計算すればよいからです。これは、f(1000) を計算するとき、+ 演算が約 1000 回しか実行されないことを意味します。私たちは、Python の再帰深度が約 1000 であることを知っています。これは、プログラムを最適化するために、2000 を計算したい場合、まず 1000 を計算するという方法を取ることができることを意味します。しかし、この方法でも再帰深度オーバーフローエラーが発生する可能性があります。2000 を計算したい場合、まず 1200 を計算します。1200 を計算したい場合、まず 400 を計算します。

このように 400 と 1200 を計算した後、2000 を計算すると、再帰の深さは約 800 程度になり、再帰の深さによるオーバーフローエラーは発生しなくなります。

```
v = []
for x in range(1000000):
   v.append(-1)
```

```
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:
      a = 0
      if n < 2:
         a = n
      else:
         a = f(n-1) + f(n-2)
      v[n] = a
      return v[n]
```

fplus 関数を追加しました。

しかし、fplus が 1000 回再帰的に呼び出された場合を考えると、1000 * 800 = 800000 となります。n を 80 万に設定した後、再び再帰深度エラーが発生しました。さらに試してみると、事態はさらに複雑であることがわかりました。しかし、この最適化の後、2000 を計算することは非常に簡単になりました。

ファイルの読み書き

話がそれてしまったようです。Web 開発の話題に戻りましょう。最初のリクエストで `f(400)` を、次のリクエストで `f(600)` を求めるとします。2 回目のリクエストでは、1 回目のリクエストで生成された `v` 配列の値を利用することができます。しかし、プログラムを終了して再起動すると、その値は使えなくなります。私たちの方法では、フィボナッチ数列の計算は非常に速いです。しかし、もし計算が遅い場合を考えてみてください。特に、`v` 配列を導入していない場合、大量の重複計算が発生します。このような場合、せっかく得られた結果を保存しておきたいと思うでしょう。

この時、キャッシュの概念が導入されます。`v` 配列はここではキャッシュとして機能します。ただし、これはプログラムのライフサイクル内にのみ存在します。プログラムが終了すると、それ

は消えてしまいます。では、どうすれば良いでしょうか。自然な考えとして、ファイルに保存することが思い浮かびます。

v 配列をファイルに保存するにはどうすればいいですか。

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

私たちの `v` 配列は次のように保存することができます。各行を `n f(n)` として保存します。`n` は自然に増加するため、おそらく `f(n)` の値だけを保存することができるでしょう。

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

試してみてください。

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

このコードは、`demofile2.txt` というファイルを追記モード (`"a"`) で開き、ファイルの末尾に「Now the file has more content!」というテキストを追加し、その後ファイルを閉じます。

ファイルを開いて、追記後に読み取る：

f = open(“demofile2.txt”, “r”) print(f.read())

`open`の2番目の引数には`a`を指定できます。これはファイルの末尾に追加することを意味します。また、`w`を指定すると、

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

上記のコードは、ファイル `fib_v` を追記モード (`'a'`) で開き、そのファイルに文字列 `'hi'` を書き込み、最後にファイルを閉じる処理を行っています。このコードは Python でファイル操作を行う基本的な例です。

実行してみると、確かに `fib_v` というファイルが生成されました。

```
fib_v:

hi
```

もう一度実行すると、このようになります。

```
hihi
```

改行するにはどうすればいいですか。

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

このコードは、ファイル `fib_v` を追記モード (`'a'`) で開き、そのファイルに文字列 `'hi\n'` を書き込み、最後にファイルを閉じる処理を行います。

これは一度だけ出力され、`hihihi` が表示されますが、改行は見えません。しかし、もう一度出力すると、改行されます。これにより、最初の出力時にも改行文字が出力されていたことがわかりますが、末尾にあったため見えなかったのです。

どのように読み取るのでしょうか。

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

このコードは、ファイル `fib_v` を読み取りモードで開き、その内容を読み取って出力します。

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

次に、私たちのフィボナッチプログラムを修正しましょう。

```
v = []
for x in range(1000000):
   v.append(-1)

def read():
   file = open('fib_v', 'r')
   s = file.read()
   if len(s) > 0:
      lines = s.split('\n')
      if (len(lines) > 0):
        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 start == False:
         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:
        a = 0
        if n < 2:
            a = n
        else:
            a = f(n-1) + f(n-2)
        v[n] = a
        return v[n]

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

ついにプログラムを書き終えました。プログラムを実行した後、`fib_v` ファイルは次のようになります。

```
fib_v:

0
1
1
2
3
5
8
13
21
34
55
```

このコードブロックは、フィボナッチ数列の最初の 11 項を表示しています。フィボナッチ数列

は、各項が前の 2 つの項の和となる数列で、0 と 1 から始まります。この数列は、数学や自然界の多くの現象で見られる重要なパターンです。

上記の解析が少し面倒だと感じるかもしれません。`\n` は改行文字です。もっと簡単で統一的な解析方法はないでしょうか。そこで、人々は `JSON` というデータ形式を発明しました。

JSON

JSON の正式名称は `JavaScript Object Notation` です。以下は `JSON` の例です。

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

この JSON データは、名前が「John」、年齢が 31 歳、都市が「New York」であることを示しています。

以上のようにして、一種のマッピングを表現します。

JSON には以下の基本的な要素があります：

1. 数字または文字列
2. リスト
3. マッピング

これらの基本要素は自由にネストすることができます。つまり、リストの中にリストを含めることができます。マッピングの中にもリストを含めることができます。などなど。

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

この JSON データは、以下の情報を表しています：

- **name**: “John”（名前は John）
- **age**: 30（年齢は 30 歳）
- **cars**: [“Ford”, “BMW”, “Fiat”]（所有している車は Ford、BMW、Fiat）

このデータは、John という人物の名前、年齢、そして所有する車のリストを表しています。

一行で書くのと、このように見やすく書くのとでは、意味の違いがあります。おそらく、それらの計算グラフを想像することができるでしょう。スペースは計算グラフに影響を与えません。

次に、v 配列を `json` 形式の文字列に変換します。

```
import json

v = []
for x in range(1000000):
   v.append(-1)

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:
      a = 0
      if n < 2:
         a = n
      else:
         a = f(n-1) + f(n-2)
      v[n] = a
      return v[n]

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

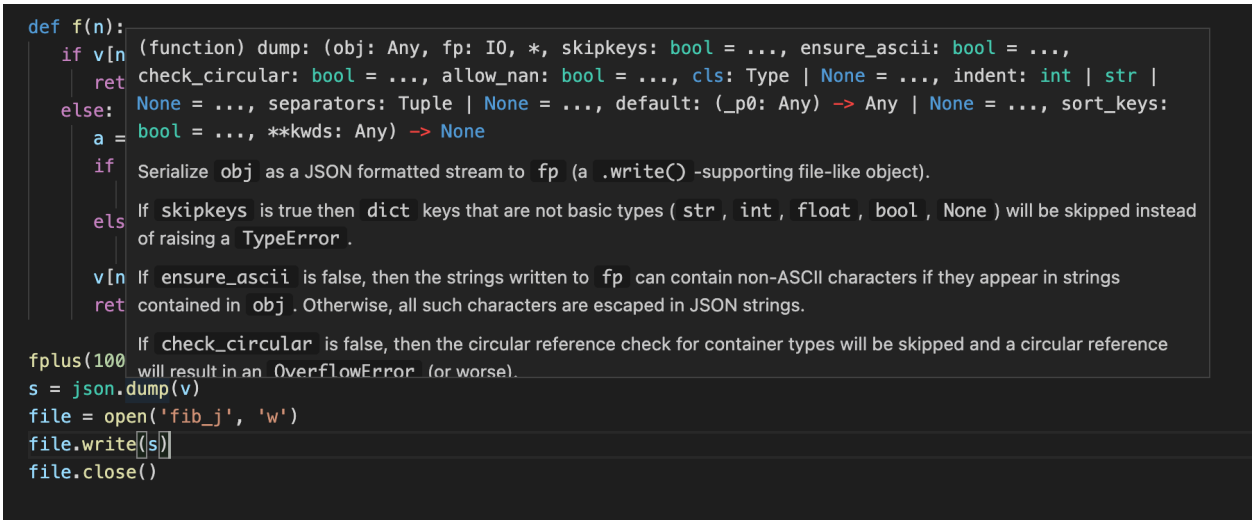

Figure 1: json

このように書いたときに、エラーが発生しました。`TypeError: dump() missing 1 required positional argument: 'fp'`。vscode では、このように関数の定義を確認することができます。

dump の上にマウスを移動するだけで OK です。便利ですよね。

```
fplus(10)
file = open('fib_j', 'w')
json.dump(v, file)
file.close()
```

このコードは、`fplus` 関数に引数 `10` を渡して実行し、その結果を `fib_j` というファイルに JSON 形式で保存するものです。具体的には、`fplus(10)` の結果を変数 `v` に格納し、`json.dump` 関数を使ってその内容をファイルに書き込んでいます。最後に、ファイルを閉じてリソースを解放しています。

100 まで計算して表示するのは少し多いので、ここでは 10 に変更します。元の dump の第二引数に `file` オブジェクトを渡せば良いことがわかりました。

以下のようにファイルを表示できます：

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

この JSON 配列は、フィボナッチ数列の一部と、その後ろに-1 が 3 つ続く形で構成されています。フィボナッチ数列は、各数が直前の 2 つの数の和となる数列で、ここでは 0, 1 から始まっています。最後の 3 つの-1 は、特定の意味を持たせるためのプレースホルダーか、データの終わりを示すマーカーとして使用されている可能性があります。

注意：後ろに多くの-1 が省略されています。

```
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)
```

この場合、以下のように出力されます：

```
0
1
1
2
3
5
8
13
21
34
55
```

以下の関数を一緒に確認しましょう：

```
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()
```

次にファイルを確認すると、確かに正しい値が保存されており、しかも整然としていました。

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

データベース

データが大きく構造が複雑な場合はどうすればいいでしょうか。ファイルで保存する方法は遅くて煩雑になります。そこでデータベースが導入されます。これはプログラマブルな `Excel` シートのようなものです。コードを使って簡単にデータの追加、削除、更新、検索ができる `Excel` シートのようなものです。

公式ドキュメントに例を見つけました。

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

このコードは、Python で SQLite データベースに接続するための基本的なコードです。`sqlite3` モジュールをインポートし、`example.db` という名前のデータベースファイルに接続しています。もし `example.db` が存在しない場合、新たに作成されます。

```
cur = con.cursor()
```

テーブルの作成

cur.execute(’‘’CREATE TABLE stocks (date text, trans text, symbol text, qty real, price real)” ’)

データの 1 行を挿入

cur.execute(“INSERT INTO stocks VALUES (‘2006-01-05’,‘BUY’,‘RHAT’,100,35.14)”)

変更を保存（コミット）する

con.commit()

接続が不要になった場合は、閉じることもできます。

ただし、変更がコミットされていることを確認してください。そうでないと、変更が失われます。

con.close()

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

このコードは、`stocks` テーブルからすべての行を `price` 順に並べ替えて取得し、各行を表示するものです。Python の SQLite データベース操作において、`cur.execute()` メソッドを使用して SQL クエリを実行し、その結果をループで処理しています。

`cursor` はカーソルを表し、ちょうどテキストエディタのカーソルのようなものです。上記のコードは、データベースへの接続、テーブルの作成、データの挿入、変更のコミット、接続のクローズを行うものです。最後の例は、データをクエリするサンプルです。

```
import sqlite3

v = []
for x in range(1000000):
   v.append(-1)

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()

save()
```

上記のコードは、fplus 関数に引数 10 を渡して呼び出し、その後 save() 関数を呼び出しています。このコードの具体的な動作は、fplus と save の実装に依存します。もしこれらの関数がどのように定義されているかが不明であれば、その動作を正確に説明することはできません。

書き終えました。試してみてください。

私のコンピュータにはすでに sqlite3 がインストールされています。

```
$ sqlite3
SQLite version 3.32.3 2020-06-18 14:16:19
使い方のヒントは ".help" と入力してください。
一時的なメモリ内データベースに接続しました。
永続的なデータベースで再開するには ".open FILENAME" を使用してください。
```

```
sqlite> .help
.auth ON|OFF              認証コールバックを表示する
.backup ?DB? FILE         データベースDB（デフォルトは"main"）をFILEにバックアップする
.bail on|off              エラー発生後に停止する。デフォルトはOFF
.binary on|off            バイナリ出力をオンまたはオフにする。デフォルトはOFF
.cd DIRECTORY             作業ディレクトリをDIRECTORYに変更する
.changes on|off           SQLによって変更された行数を表示する
.check GLOB               .testcase以降の出力が一致しない場合に失敗する
.clone NEWDB              既存のデータベースからNEWDBにデータをクローンする
.databases                接続されているデータベースの名前とファイルをリストする
.dbconfig ?op? ?val?      sqlite3_db_config()オプションをリストまたは変更する
.dbinfo ?DB?              データベースのステータス情報を表示する
.dump ?TABLE?             データベースの内容をSQLとしてレンダリングする
.echo on|off              コマンドエコーをオンまたはオフにする
.eqp on|off|full|...      自動EXPLAIN QUERY PLANを有効または無効にする
.excel                    次のコマンドの出力をスプレッドシートで表示する
.exit ?CODE?              リターンコードCODEでこのプログラムを終了する
.expert                   実験的機能。クエリのためのインデックスを提案する
.explain ?on|off|auto?    EXPLAINフォーマットモードを変更する。デフォルト: auto
.filectrl CMD ...         さまざまなsqlite3_file_control()操作を実行する
.fullschema ?--indent?    スキーマとsqlite_statテーブルの内容を表示する
.headers on|off           ヘッダーの表示をオンまたはオフにする
.help ?-all? ?PATTERN?    PATTERNのヘルプテキストを表示する
.import FILE TABLE        FILEからTABLEにデータをインポートする
.imposter INDEX TABLE     インデックスINDEX上に偽装テーブルTABLEを作成する
.indexes ?TABLE?          インデックスの名前を表示する
.limit ?LIMIT? ?VAL?      SQLITE_LIMITの値を表示または変更する
.lint OPTIONS             潜在的なスキーマの問題を報告する
.log FILE|off             ログをオンまたはオフにする。FILEはstderr/stdoutにできる
.mode MODE ?TABLE?        出力モードを設定する
.nullvalue STRING         NULL値の代わりにSTRINGを使用する
.once ?OPTIONS? ?FILE?    次のSQLコマンドの出力のみをFILEに出力する
.open ?OPTIONS? ?FILE?    既存のデータベースを閉じてFILEを再度開く
.output ?FILE?            出力をFILEまたはstdoutに送る（FILEが省略された場合）
.parameter CMD ...        SQLパラメータバインディングを管理する
.print STRING...          リテラルSTRINGを出力する
```

.progress N	毎回Nオペコード後にプログレスハンドラを呼び出す
.prompt MAIN CONTINUE	標準のプロンプトを置き換える
.quit	このプログラムを終了する
.read FILE	FILEから入力を読み取る
.recover	破損したデータベースから可能な限りデータを回復する
.restore ?DB? FILE	DB（デフォルトは"main"）の内容をFILEから復元する
.save FILE	メモリ内のデータベースをFILEに書き込む
.scanstats on|off	sqlite3_stmt_scanstatus()メトリクスをオンまたはオフにする
.schema ?PATTERN?	PATTERNに一致するCREATE文を表示する
.selftest ?OPTIONS?	SELFTESTテーブルで定義されたテストを実行する
.separator COL ?ROW?	列と行の区切り文字を変更する
.session ?NAME? CMD ...	セッションを作成または制御する
.sha3sum ...	データベース内容のSHA3ハッシュを計算する
.shell CMD ARGS...	システムシェルでCMD ARGS...を実行する
.show	各種設定の現在の値を表示する
.stats ?on|off?	統計を表示するか、統計をオンまたはオフにする
.system CMD ARGS...	システムシェルでCMD ARGS...を実行する
.tables ?TABLE?	LIKEパターンTABLEに一致するテーブルの名前をリストする
.testcase NAME	出力を'testcase-out.txt'にリダイレクトし始める
.testctrl CMD ...	さまざまなsqlite3_test_control()操作を実行する
.timeout MS	ロックされたテーブルをMSミリ秒間開こうとする
.timer on|off	SQLタイマーをオンまたはオフにする
.trace ?OPTIONS?	各SQLステートメントを実行時に出力する
.vfsinfo ?AUX?	トップレベルのVFSに関する情報
.vfslist	利用可能なすべてのVFSをリストする
.vfsname ?AUX?	VFSスタックの名前を表示する
.width NUM1 NUM2 ...	"column"モードの列幅を設定する

多くのコマンドがあることがわかります。その中で、.quit は終了を意味します。

もしインストールされていない場合は、公式サイトからダウンロードするか、brew install sqlite を実行してインストールできます。

```
$ sqlite3 fib.db

sqlite> show tables
   ...> ;
エラー: "show" 付近で構文エラー
```

```
sqlite> tables;
エラー: "tables" 付近で構文エラー
sqlite> .schema
CREATE TABLE vs(v text);
```

最初、私は MySQL と同じように使えると思っていました。show tables を使ってどのようなテーブルがあるか確認できると思っていました。しかし、後で SQLite ではそうではないことに気づきました。MySQL は別のデータベースで、これから学ぶ予定のものです。

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

確かに、データを正しく書き込むことができました。注意点として、私たちは text を使用しています。なぜなら、数字が非常に大きいため、データベースの整数型では保存できない可能性があるからです。

```
import sqlite3

v = []
for x in range(1000000):
   v.append(-1)

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

このコードは、n が 800 より大きい場合に、n から 800 を引いた値で再帰的に fplus 関数を呼び出し、その後 f(n) を返します。n が 800 以下の場合には、単に f(n) を返します。

```
def f(n):
   if v[n] != -1:
      return v[n]
   else:
      a = 0
      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 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)
    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])
```

続いて、read 関数を追加しました。しかし、実行するとエラーが発生しました。

```
$ 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
```

上記のエラーメッセージは、Python スクリプト fib_db.py を実行した際に発生したエラーを示しています。具体的には、sqlite3.OperationalError が発生し、table vs already exists というメッセージが表示されています。これは、データベース内に既に vs という名前のテーブルが存在しているため、新たに同じ名前のテーブルを作成しようとした際にエラーが発生したことを意味します。

このエラーを解決するためには、以下のいずれかの方法を検討できます：

1. **既存のテーブルを削除する**: 既存の vs テーブルが不要であれば、DROP TABLE vs; という SQL コマンドを実行してテーブルを削除し、再度スクリプトを実行します。
2. **テーブルの存在を確認する**: テーブルを作成する前に、既に同名のテーブルが存在しないかを確認するコードを追加します。例えば、以下のように IF NOT EXISTS を使用してテーブルを作成することができます。

   ```
   cur.execute('CREATE TABLE IF NOT EXISTS vs(v text)')
   ```

3. **テーブルの再作成を避ける**: テーブルが既に存在する場合には、テーブルを作成しないようにロジックを変更します。

これらの方法のいずれかを適用することで、エラーを回避し、スクリプトを正常に実行できるようになります。

テーブルを作成できません。テーブルは既に存在しています。構文を少し変更してください。

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

しかし、エラーが発生しました。

```
    v[i] = int(row)
TypeError: int() の引数は文字列、バイト列のようなオブジェクト、または数値でなければなりませんが、'tuple' が指定
```

tuple とは何でしょうか。これは、row が tuple を返したことを意味します。それを出力してみましょう。

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

上記のコードは、データベースから vs テーブルのすべての行を選択し、各行を表示して、その値を整数に変換して配列 v の i 番目の要素に格納するものです。

結果は以下の通りです：

```
('0',)
```

このコードスニペットは、Python のタプルを表しています。タプルは不変（イミュータブル）なシーケンスで、複数の要素をカンマで区切って括弧で囲むことで作成されます。この場合、タプルには文字列 '0' が含まれています。タプルは、要素の順序や内容が変更されないことを保証するために使用されます。

実は tuple は配列とよく似ています。ただし、その要素は互いに異なるものであってもよく、配列のようにすべての要素が同じ型である必要はありません。

```
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])
    con.close()
```

このコードは、SQLite データベースからデータを読み取るための Python 関数です。以下にその内容を説明します。

1. con = sqlite3.connect('fib.db'): fib.db という名前の SQLite データベースに接続します。
2. cur = con.cursor(): データベース操作を行うためのカーソルオブジェクトを作成します。

3. `create_table(cur)`: `create_table` 関数を呼び出して、データベースにテーブルを作成します（この関数の定義はコードに含まれていません）。
4. `i = 0`: インデックス変数 `i` を初期化します。
5. `for row in cur.execute('SELECT * from vs')`: `vs` テーブルからすべての行を選択し、各行に対してループを実行します。
6. `v[i] = int(row[0])`: 各行の最初の列の値を整数に変換し、リスト `v` の `i` 番目の要素に代入します。
7. `con.close()`: データベース接続を閉じます。

このコードは、データベースからデータを読み取り、それをリスト `v` に格納するためのものです。ただし、`v` の定義や `create_table` 関数の詳細はコードに含まれていないため、完全な動作を理解するにはそれらの情報が必要です。

このように変更しました。しかし、奇妙なことに、出力は以下のようになりました：

```
55
-1
-1
-1
-1
-1
-1
-1
-1
-1
```

元々、私たちの `i` がインクリメントされていなかったのです。

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

このコードは、データベースから `vs` テーブルのすべての行を選択し、各行の最初の値を整数に変換して配列 `v` に格納しています。変数 `i` は、配列 `v` のインデックスとして使用され、各行の処理後にインクリメントされます。

これで正解です。

```
0
```

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

しかし、数字が大きい場合、データベースに保存される形式は以下のようになっていることに気づきました：

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

(注：上記の数値はそのまま表示されています。これらは特定の計算やデータを示すコードブロックの一部である可能性がありますが、翻訳の対象外としています。)

再実行すると、以下のようになります。

```
$ python fib_db.py
Traceback (most recent call last):
  File "fib_db.py", line 35, in read
    v[i] = int(row[0])
ValueError: 基数10のint()に無効なリテラルです: '1.22001604151219e+19'
```

変更を加える：

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

このコードは、Python で SQL クエリを実行する際に、変数 `vv` の値を文字列として SQL の `INSERT` 文に組み込むものです。具体的には、`vs` テーブルに `vv` の値を挿入する SQL クエリを生成しています。

ここで、INSERT 文の両側のシングルクォートをダブルクォートに変更し、数字の文字列にクォートを追加したことに気づきました。以前の書き方では、データベースは私たちの文字列を数字として扱っていましたが、今ではクォートで囲むことで文字列として認識されます。

その後、正しく動作するようになりました。しかし、以前の誤ったデータをどのようにクリアするかが問題です。

```
$ sqlite3 fib.db
SQLite version 3.32.3 2020-06-18 14:16:19
使い方のヒントは ".help" と入力してください。
sqlite> delete * from vs;
```

次に、他の操作を試してみましょう。増削改検（CRUD）です。ここでは、増削検の例を挙げました。

練習

- 学生は上記のように探索を進めます。