第6回
BAU Study Session

【テーマ】

・Python：四則演算アルゴリズム

・Python：SQLiteで日経平均DBを作ってみる

日時　：2020年7月1日

参加者：古川(発表者)・森(発表者)・有川

四則演算アルゴリズム

四則演算アルゴリズム
～ ルール定義 ～

- 数式の文字列を解析して解を求める

- 数式に出てくるのは「0-9」「+」「-」「*」「/」のみ（カッコは無し）
  例：“-123+4*5-67/8+90”

※これをPythonで考えてみよう

四則演算アルゴリズム
～ まずは戦略 ～

"-123+4*5-67/8+90"

①数値と演算子を分解する

「-」「123」「+」「4」「*」「5」「-」「67」「/」「8」「+」「90」

②乗算、除算を計算する

「-」「123」「+」「20」「-」「8.375」「+」「90」

③加算、減算を計算する

「-21.375」

四則演算アルゴリズム
～ ①数値と演算子を分解する ～

```
target = “-123+4*5-67/8+90"

import re

# 数値と演算子を分離
parts = [target[m.span()[0]:m.span()[1]]
          for m in re.finditer('\+|-|\*|/|[0-9]+\.?[0-9]*', target)]
```

```
['-', '123', '+', '4', '*', '5', '-', '67', '/', '8', '+', '90']
```

四則演算アルゴリズム
～ 分けて書いてみる ～

```
target = "-123+4*5-67/8+90"

import re
poses = [m.span() for m in re.finditer('\+|-|\*|/|[0-9]+', target)]
```

```
[(0, 1), (1, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9), (9, 11), (11, 12),
(12, 13), (13, 14), (14, 16)]
```

```
parts = []
for pos in poses:
  parts.append(target[pos[0]:pos[1]])
```

```
['-', '123', '+', '4', '*', '5', '-', '67', '/', '8', '+', '90']
```

四則演算アルゴリズム
～ ②乗算、除算を計算する ～

```
def maltidiv(in_parts) :
  no = 0
  for part in in_parts:
    if part in ["*", "/"]:
        newpart = []
        if parts[no] == "*":
          newpart.append( float(parts[no-1]) * float(parts[no+1]) )
        elif parts[no] == "/":
          newpart.append( float(parts[no-1]) / float(parts[no+1]))
        newparts = parts[0:no-1] + newpart + parts[no+2:]
        return newparts, True
    no = no + 1
  return in_parts, False
```

四則演算アルゴリズム
～ ②乗算、除算を計算する ～

```
cont = True
while cont:
  print(parts)
  parts, cont = maltidiv(parts)
```

```
['-', '123', '+', '4', '*', '5', '-', '67', '/', '8', '+', '90']
['-', '123', '+', 20.0, '-', '67', '/', '8', '+', '90']
['-', '123', '+', 20.0, '-', 8.375, '+', '90']
```

四則演算アルゴリズム
～ ③加算、減算を計算する ～

```
value = 0.0
sign = ""
for part in parts:
  if part in ["+", "-"]:
    sign = part
  else:
    if sign == "+":
      value = value + float(part)
      print(F"[{sign}][{part}] => value=[{value}]")
    elif sign == "-":
      value = value - float(part)
      print(F"[{sign}][{part}] => value=[{value}]")
    else:
      value = float(part)
value
```

```
[-][123] => value=[-123.0]
[+][20.0] => value=[-103.0]
[-][8.375] => value=[-111.375]
[+][90] => value=[-21.375]
-21.375
```

四則演算アルゴリズム (C#版)
～ 戦略 ～

① 数式の文字列を逆ポーランド記法に変換

例：10 + 20 * 30 + 40 - 50 * 2 / 4

↓ ↓ ↓

10 20 30 * + 40 + 50 2 * 4 / -

② 逆ポーランド記法から計算結果を出力

四則演算アルゴリズム (C#版)
～ ① 逆ポーランド記法に変換 ～

1. 値と演算子に分ける

index	0	1	2	3	4	5	6
values	10	20	30	40	50	2	4
operations	+	*	+	-	*	/	

2. 値と演算子をキューに格納

index	0	1	2	3	4	5	6
valQueue	10	20	30	40	50	2	4
opQueue	+[1]	*[1]	+[2]	-	*[2]	/	
rpnList							
opStack							

一つ目の + を + [1]
二つ目の * を * [2]
などと表現している

四則演算アルゴリズム (C#版)
～ ① 逆ポーランド記法に変換 ～

3. 値と演算子をキューから取り出し、リストとスタックに格納

index	0	1	2	3	4	5
valQueue	20	30	40	50	2	4
opQueue	*[1]	+[2]	-	*[2]	/	
rpnList	10					
opStack	+[1]					

4. *[1] は +[1] より優先度が高いのでスタックに積む

index	0	1	2	3	4
valQueue	30	40	50	2	4
opQueue	+[2]	-	*[2]	/	
rpnList	10	20			
opStack	*[1]	+[1]			

四則演算アルゴリズム (C#版)
～ ① 逆ポーランド記法に変換 ～

5. +[2] は *[1] より優先度が低いので *[1] を取り出す +[1] とは同等なのでこれも取り出す。その後 +[2] をスタックに積む

index	0	1	2	3	4
valQueue	40	50	2	4	
opQueue	-	*[2]	/		
rpnList	10	20	30	*[1]	+[1]
opStack	+[2]				

6. +[2] は - と同等なので取り出す。その後 - をスタックに積む

index	0	1	2	3	4	5	6
valQueue	50	2	4				
opQueue	*[2]	/					
rpnList	10	20	30	*[1]	+[1]	40	+[2]
opStack	-						

四則演算アルゴリズム (C#版)
～ ① 逆ポーランド記法に変換 ～

7. *[2] は - より優先度が高いのでスタックに積む

index	0	1	2	3	4	5	6	7
valQueue	2	4						
opQueue	/							
rpnList	10	20	30	*[1]	+[1]	40	+[2]	50
opStack	*[2]	-						

8. / は *[2] と同等なので取り出す。 / は - より優先度が高いのでスタックに積む

index	0	1	2	3	4	5	6	7	8	9
valQueue	4									
opQueue										
rpnList	10	20	30	*[1]	+[1]	40	+[2]	50	2	*[2]
opStack	/	-								

四則演算アルゴリズム (C#版)
～ ① 逆ポーランド記法に変換 ～

9. 最後の数字と、スタックをリストに追加

index	0	1	2	3	4	5	6	7	8	9	10	11	12
valQueue													
opQueue													
rpnList	10	20	30	*[1]	+[1]	40	+[2]	50	2	*[2]	4	/	-
opStack													

四則演算アルゴリズム (C#版)
～ ② 逆ポーランド記法から計算結果を出力 ～

・逆ポーランド記法の先頭から順にみていき、数値はすべてスタックへ格納。
・演算子が出てきたらスタックから二つ取り出し、その演算子と数値を計算して再びスタックへ格納。
・これを繰り返す。

1. 結果を格納するスタックを用意

index	0	1	2	3	4	5	6	7	8	9	10	11	12
rpnList	10	20	30	*[1]	+[1]	40	+[2]	50	2	*[2]	4	/	-
resultStack													

2. 演算子が出てくるまで数値をスタックへ入れる

index	0	1	2	3	4	5	6	7	8	9	10	11	12
rpnList	*[1]	+[1]	40	+[2]	50	2	*[2]	4	/	-			
resultStack	30	20	10										

四則演算アルゴリズム (C#版)
～ ② 逆ポーランド記法から計算結果を出力 ～

3. 演算子[*]が出てきたのでスタックから数値二つを取り出して計算し、結果をスタックへ

20 * 30 = 600

index	0	1	2	3	4	5	6	7	8	9	10	11	12
rpnList	+[1]	40	+[2]	50	2	*[2]	4	/	-				
resultStack	600	10											

4. 演算子[+]が出てきたのでスタックから数値二つを取り出して計算し、結果をスタックへ

10 + 600 = 610

index	0	1	2	3	4	5	6	7	8	9	10	11	12
rpnList	40	+[2]	50	2	*[2]	4	/	-					
resultStack	610												

四則演算アルゴリズム (C#版)
～ ② 逆ポーランド記法から計算結果を出力 ～

5. 演算子が出てくるまで数値をスタックへ入れる

index	0	1	2	3	4	5	6	7	8	9	10	11	12
rpnList	+[2]	50	2	*[2]	4	/	-						
resultStack	40	610											

6. 演算子[+]が出てきたのでスタックから数値二つを取り出して計算し、結果をスタックへ

610 + 40 = 650

index	0	1	2	3	4	5	6	7	8	9	10	11	12
rpnList	50	2	*[2]	4	/	-							
resultStack	650												

四則演算アルゴリズム (C#版)
～ ② 逆ポーランド記法から計算結果を出力 ～

7. 演算子が出てくるまで数値をスタックへ入れる

index	0	1	2	3	4	5	6	7	8	9	10	11	12
rpnList	*[2]	4	/	-									
resultStack	2	50	650										

8. 演算子[*]が出てきたのでスタックから数値二つを取り出して計算し、結果をスタックへ

50 * 2 = 100

index	0	1	2	3	4	5	6	7	8	9	10	11	12
rpnList	4	/	-										
resultStack	100	650											

四則演算アルゴリズム (C#版)
～ ② 逆ポーランド記法から計算結果を出力 ～

9. 演算子が出てくるまで数値をスタックへ入れる

index	0	1	2	3	4	5	6	7	8	9	10	11	12
rpnList	/	-											
resultStack	4	100	650										

10. 演算子[/]が出てきたのでスタックから数値二つを取り出して計算し、結果をスタックへ

100 / 4 = 25

index	0	1	2	3	4	5	6	7	8	9	10	11	12
rpnList	-												
resultStack	25	650											

四則演算アルゴリズム (C#版)
～ ② 逆ポーランド記法から計算結果を出力 ～

11. 演算子[-]が出てきたのでスタックから数値二つを取り出して計算し、結果をスタックへ

650 - 25 = 625

index	0	1	2	3	4	5	6	7	8	9	10	11	12
rpnList													
resultStack	625												

四則演算アルゴリズム (C#版)
～ おまけ ～

逆ポーランド記法から結果を求める方法で考えましたが

Pythonと同じ方法でもソースをGithubにUPしてます。（Class：Calculator2）

下記は、文字列から数値と演算子に分割して配列にする際のコードです。

```
// 数値または演算子で分割してリストにする
var elements = Regex.Split(expression.Replace(" ", ""), "([0-9]+|\\+|-|\\*|/)")
                    .Where(x => !string.IsNullOrWhiteSpace(x)).ToList();
```

SQLiteで日経平均DBを作ってみる

SQLiteで日経平均DBを作ってみる
～ 準備編（NK225ダウンロード）～

```
### 株価の取得は前回と同じ ###

!pip install pandas_datareader
from pandas_datareader import data
import pandas as pd

#期間設定

start = ‘2015-06-01'
end = '2020-07-01'

#日経225平均を取得

df = data.DataReader('^N225', 'yahoo', start, end)
```

SQLiteで日経平均DBを作ってみる
～ DB作成 / テーブル作成&インポート ～

```
# SQLite3パッケージをインポート
import sqlite3

#DB作成
dbname = 'STOCK.db'
conn = sqlite3.connect(dbname)

#テーブルを作成＆インポート
df.to_sql('N225', conn, if_exists = 'replace')


conn.close()
```

```
to_sqlのオプション：
if_exists  : append, replace, fail
```

SQLiteで日経平均DBを作ってみる
～ SQLでデータ検索 ～

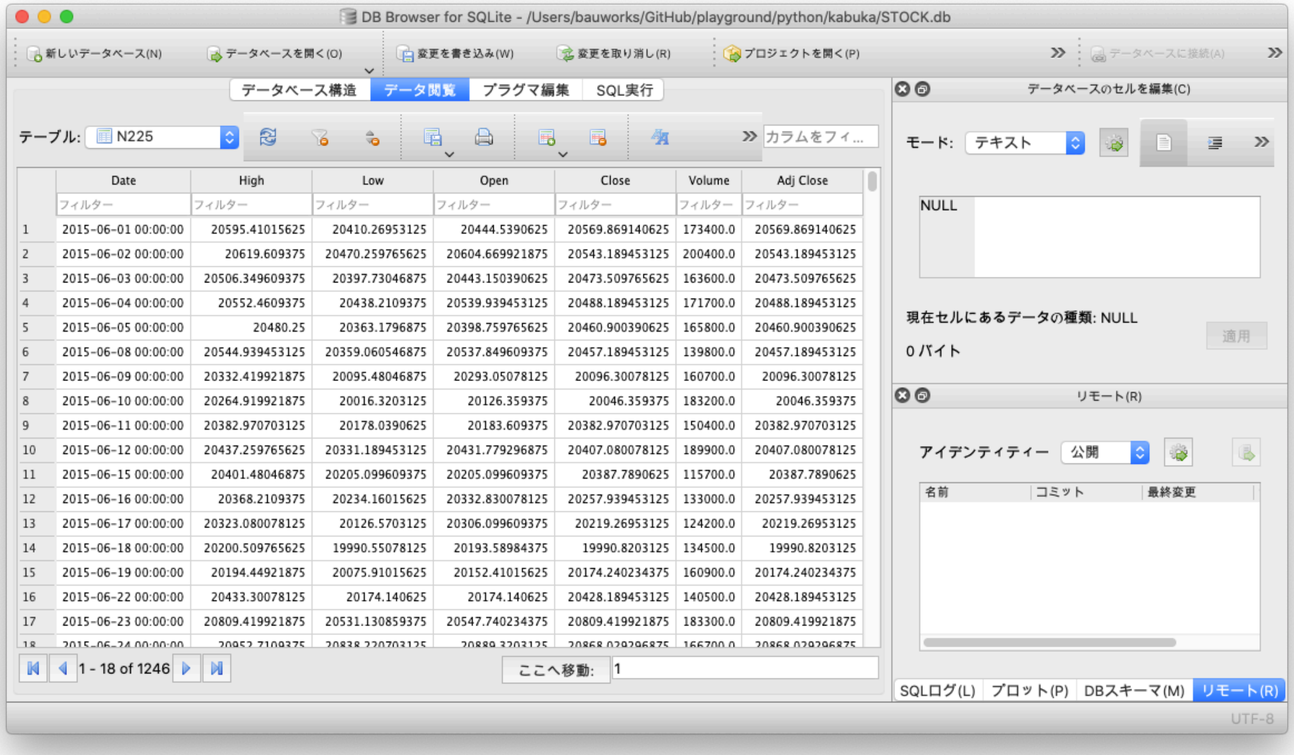
```
sql = '''
select * from N225
where Date between '2020-05-01 00:00:00'
  and  '2020-07-01 00:00:00'
'''
conn = sqlite3.connect(dbname)
df_vlm = pd.read_sql(sql, conn)
conn.close()
```

	Date	High	Low	Open	Close	Volume	Adj Close
0	2020-05-01 00:00:00	20000.250000	19551.730469	19991.970703	19619.349609	86600.0	19619.349609
1	2020-05-07 00:00:00	19720.869141	19448.929688	19468.519531	19674.769531	82900.0	19674.769531
2	2020-05-08 00:00:00	20179.089844	19894.580078	19972.089844	20179.089844	82200.0	20179.089844
3	2020-05-11 00:00:00	20534.880859	20285.039062	20333.730469	20390.660156	76200.0	20390.660156

SQLiteで日経平均DBを作ってみる
～ DB Browserでデータ確認 ～

「DB Browserインストール」

```
$ brew cask install db-browser-for-sqlite
```



フリートーク

～ 思いつくままに ～

次回のBSS

- 日程　：2020年7月8日(水)
- 司会者：有川
- テーマ：検討中