SciComp with Py

I/O

Vladimir Kulyukin
Department of Computer Science
Utah State University

Outline

- Command Line Args
- Files
- Pipes

Command Line Args

SYS.ARGV

- If you need to process command line args, you need to **import sys**
- All command line args are stored in **sys.argv** array
- The 0^{th} element in **sys.argv** is the name of the Py program that takes **sys.argv** as input

Py 2 Example

Py 2

```
#!/usr/bin/python

import sys

for argn, arg in zip(xrange(len(sys.argv)), sys.argv):
    print argn, ' --> ', arg
```

Output

```
>>> python sys_argv.py  python  perl  10
0  -->  sys_argv.py
1  -->  python
2  -->  perl
3  -->  10
```

Py source in py2_sys_argv.py

Py 3 Example

Py 3

```
#!/usr/bin/python3

import sys

for argn, arg in zip(range(len(sys.argv)), sys.argv):
    print(argn, ' --> ', arg)
```

Output

```
>>> python3 sys_argv.py  python  perl  10
0  -->  sys_argv.py
1  -->  python
2  -->  perl
3  -->  10
```

Py source in py3_sys_argv.py

Problem

Write two functions for computing Fibonacci numbers recursively and iteratively (fibonacci.py). Import them into a program (py2_argparse_fib.py) that accepts two command line arguments (-m and -n). The -m argument specifies the method of computaiton (iter or rec). The -n argument specifies the n-th Fibonacci number to compute.

Sample calls

```
$ python py2_argparse_fib.py -m iter -n 35


$ python py2_argparse_fib.py -m rec -n 35
```

Source Code Walkthrough

Let's take a look at Py source in py2_fibonacci.py, py3_fibonacci.py, py2_argparse_fib.py, py3_argparse_fib.py.

Files

Processing Files

Common file processing paradigm:

- Open a file (read mode, write mode, append mode, read binary mode, write binary mode, etc)
- Apply some function to each chunk (byte, line, etc.) read from file
- Save results somewhere

Problem

Write a program that takes a path to a file of integers where each integer is written on a single line, reads the file line by line, and applies a function (e.g. x+10, fibiter) to each integer.

Py source in py2_process_lines.py, py3_process_lines.py

Problem

Input File (numbers.txt)

```
1
2
3
4
5
6
7
8
9
10
```

Output

```
>>>python process_lines.py numbers.txt
11
12
13
14
15
16
17
18
19
20
1
1
2
3
5
8
13
21
34
55
```

+10

Fibonacci

Source Code Walkthrough

Let's take a look at Py source in py2_process_lines.py and py3_process_lines.py

Problem

Write a program that takes a tab-separated file of student records and uses regular expressions to extract and output their A-numbers, names, and emails.

Input File (students.txt)

```
John    Balbiro    A1001    john.balbiro@usu.edu
Alice   Nelson     A0011    alice.nelson@workflow.net
Jacob   Roberts    A1100    j.s.roberts@gmail.com
```

Output

```
$ python py2_process_student_file.py students.txt
A1001 john.balbiro@usu.edu
A0011 alice.nelson@workflow.net
A1100 j.s.roberts@gmail.com
```

Py source in py2_process_student_file.py, py3_process_student_file.py

Solution: Regular Expressions

```
# ?: is a non-capturing version of a group - it will match com or
# net or org or edu but you will not be able to retrieve it via
# back reference.
a_num_and_email   = r'.*(A\d+)\s*([\w\.-]+@[\w\.-]+\.(?:com|net|org|edu)).*'
first_name_last_name_a_num = r'(\w+)\s*(\w+)\s*(A\d+).*'
```

Solution: Making Pattern Extractors

```
def make_info_extractor(pat):
    def info_extractor(line):
        match = re.match(pat, line)
        if match is not None:
            return [match.group(i) for i in xrange(1, len(match.groups())+1)]
    return info_extractor
```

Solution: Applying a Function to Each Line w/ List Comprehension

```
def process_lines(file_path, fun):
    with open(file_path, 'r') as infile:
        return [fun(line) for line in infile]
```

This is how you apply a function to each line in the file

Source Code Walkthrough

Let's take a look at Py source in py2_process_student_file.py and py3_process_student_file.py

Problem

Write Py programs for sorting files lexicographically and numerically in ascending order and descending order. We will assume that each line in a file contains exactly one item of data, i.e., a number or a string.

Py source in ascii_sort.py, num_asc_sort.py, num_des_sort.py

Sample Files for Sorting

numbers.txt

```
1
2
3
4
5
6
7
8
9
10
```

words.txt

```
python
perl
c
poetry
music
abracadabra
mathematics
alchemy
```

ASCII File Sorting

```
# 1. get the 1st command line argument
file_path = sys.argv[1]
# 2. open the file for reading
infile = open(file_path, 'r')
# 3. read lines, sort, and print
for n in sorted([line for line in infile.readlines()]):
    print n,
```

Py source in ascii_sort.py

Ascending Numeric File Sorting

```
# 1. get 1st command line arg
file_path = sys.argv[1]
# 2. open file_path for reading
infile = open(file_path, 'r')
# 3. read the lines, sort, print
for n in sorted([int(line) for line in infile.readlines()]):
    print n
```

Py source in num_asc_sort.py

Descending Numeric File Sorting

```
# 1. get 1st command line argument
file_path = sys.argv[1]
# 2. open file_path for reading
infile = open(file_path, 'r')
# 3. read in the lines, sort, print
for n in sorted([int(line) for line in infile.readlines()],
                reverse=True):
    print n
```

Py source in num_des_sort.py

Pipes & Pipelines

Pipes

- A Pipe is a stream b/w two processes
- A pipe can be uni-directional or bi-directional
- Pipes originated on Unix and are now available on many other operating systems
- Piping paradigm: write many small useful programs and then use pipes to channel the output of one program as input into another program

Pipelines

- If there are two programs $\mathbf{P_1}$ and $\mathbf{P_2}$, you can use piping to channel the output of $\mathbf{P_1}$ into $\mathbf{P_2}$ as follows:

  > P_1 | P_2

- Pipelines can be arbitrarily long:

  > P_1 | P_2 | ... | P_{n-1} | P_n

Problem

Write Py programs for filtering odd/even numbers, thresholding number files, sorting number files in ascending/descending order, finding min/max, and computing total sums of input numbers. We will construct different pipelines from these programs.

Filtering Odds

```
#!/usr/bin/python

## print odd numbers from file in sys.argv[1]
import sys
file_path = sys.argv[1]
with open(file_path, 'r') as infile:
    for n in [int(x) for x in infile.readlines() if int(x) % 2 != 0]:
        print n
```

Py source in filter_odds.py

Filtering Evens

```
#!/usr/bin/python

## print even numbers in file in sys.argv[1]
import sys
file_path = sys.argv[1]
with open(file_path, 'r') as infile:
    for n in [int(x) for x in infile.readlines() if int(x) % 2 == 0]:
        print n
```

Py source in filter_evens.py

Testing FILTER_EVENS/ODDS

```
$ more numbers2.txt
1
3
4
10
100
12
9
11
25
78
490
56
5
8
7
17
31
90
45
```

```
$ python filter_odds.py numbers2.txt
1
3
9
11
25
5
7
17
31
45
```

```
$ python filter_evens.py numbers2.txt
4
10
100
12
78
490
56
8
90
```

Filtering Odds & Evens from STDIN

```
#!/usr/bin/python
## print odd integers from STDIN
## to test:
## > more numbers.txt | python filter_stdin_odds.py
import sys
for n in [int(x) for x in sys.stdin.readlines() if int(x) % 2 != 0]:
    print n
```

Py source in filter_stdin_odds.py

```
#!/usr/bin/python
## print even integers from STDIN
## to test:
## > more numbers.txt | python filter_stdin_evens.py
import sys
for n in [int(x) for x in sys.stdin.readlines() if int(x) % 2 == 0]:
    print n
```

Py source in filter_stdin_evens.py

Testing FILTER_STDIN_EVENS/ODDS

```
$ more numbers2.txt | python filter_stdin_evens.py
4
10
100
12
78
490
56
8
90
```

```
$ more numbers2.txt | python filter_stdin_odds.py
1
3
9
11
25
5
7
17
31
45
```

Thresholding Files on <=

```
#!/usr/bin/python
## print numbers from file in sys.argv[1] that are <= int(sys.argv[2])
## to run: >python lte_thresh.py numbers.txt 10
## @author: vladimir kulyukin
import sys
file_path, thresh = sys.argv[1], int(sys.argv[2])
with open(file_path, 'r') as infile:
    for n in [int(line) for line in infile.readlines()
                if int(line) <= thresh]:
        print n
```

Py source in lte_thresh.py

Testing LTE_TRESH.PY

```
$ python lte_thresh.py numbers.txt 5
1
2
3
4
5
```

```
$ python lte_thresh.py numbers2.txt 20
1
3
4
10
12
9
11
5
8
7
17
```

Thresholding Files on >=

```
#!/usr/bin/python
## print numbers from file in sys.argv[1] that are >= int(sys.argv[2])
## @author: vladimir kulyukin
import sys
file_path, thresh = sys.argv[1], int(sys.argv[2])
with open(file_path, 'r') as infile:
    for n in [int(line) for line in infile.readlines()
                if int(line) >= thresh]:
        print n
```

Py source in gte_thresh.py

Testing GTE_TRESH.PY

```
$ python gte_thresh.py numbers.txt 5
5
6
7
8
9
10
```

```
$ python gte_thresh.py numbers2.txt 20
100
25
78
490
56
31
90
45
```

Scripts for Thresholding STDIN on <=

```
#!/usr/bin/python
## print integers from STDIN that are <= sys.argv[1]
## to run: more numbers.txt | python lte_input_thresh.py 10
import sys
thresh = int(sys.argv[1])
for n in [int(x) for x in sys.stdin.readlines() if int(x) <= thresh]:
    print n
```

Py source in lte_stdin_thresh.py

Testing LTE_STDIN_TRESH.PY

```
$ cat numbers.txt | python lte_stdin_thresh.py  5
1
2
3
4
5
```

```
$ cat numbers2.txt | python lte_stdin_thresh.py  20
1
3
4
10
12
9
11
5
8
7
17
```

Scripts for Thresholding STDIN on >=

```
#!/usr/bin/python
## print integers from STDIN that are >= sys.argv[1]
import sys
thresh = int(sys.argv[1])
for n in [int(x) for x in sys.stdin.readlines() if int(x) >= thresh]:
    print n
```

Py source in gte_stdin_thresh.py

Testing GTE_STDIN_TRESH.PY

```
$ cat numbers.txt | python gte_stdin_thresh.py numbers.txt 5
5
6
7
8
9
10
```

```
$ cat numbers2.txt | python gte_stdin_thresh.py numbers2.txt 20
100
25
78
490
56
31
90
47
```