Python Lab: Making Command Line Scripts

Umer Zeeshan Ijaz

Problem statement:
We have processed a given test.fasta file by running it against NCBI's locally-installed **nt** database using –outfmt 6 switch in blastn, which returns the matches in tabular form as 'qseqid sseqid pident length mismatch gapopen qstart qend sstart send evalue bitscore'
where,

qseqid means Query Seq-id
sseqid means Subject Seq-id
pident means Percentage of identical matches
length means Alignment length
mismatch means Number of mismatches
gapopen means Number of gap openings
qstart means Start of alignment in query
qend means End of alignment in query
sstart means Start of alignment in subject
send means End of alignment in subject
evalue means Expect value
bitscore means Bit score

The blastn command used was as follows:

```
blastn -db nt -query test.fasta -out test.out -outfmt "6" -num_threads 8 -max_target_seqs 1
```

As a bioinformatician, your task is to get the title and taxonomy ID of the reference hits by writing a python program that takes as an input the blast output file generated from the above command, and use NCBI's eutils to search the relevant record in NCBI's online databases using URL requests. We are **NOT** going to use **Biopython** in our script!

What is expected from YOU: Type all the python commands given in this handout and go through the links mentioned in the text.

Solution:

Python provides a **getopt** module that helps you parse command-line options and arguments.

```
$ python test.py arg1 arg2 arg3
```

The Python **sys** module provides access to any command-line arguments via the **sys.argv**. This serves two purpose:

- sys.argv is the list of command-line arguments.
- len(sys.argv) is the number of command-line arguments.

Here sys.argv[0] is the program ie. script name.
Consider the following script test.py:

```
#!/usr/bin/python
import sys
print 'Number of arguments:', len(sys.argv), 'arguments.'
```

```
print 'Argument List:', str(sys.argv)
```

Now run above script as follows:

```
$ python test.py arg1 arg2 arg3
```

This will produce following result:

```
Number of arguments: 4 arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

NOTE: As mentioned above, first argument is always script name and it is also being counted in number of arguments.

Parsing Command-Line Arguments:

Python provided a **getopt** module that helps you parse command-line options and arguments. This module provides two functions and an exception to enable command line argument parsing. This tutorial would discuss about one method and one exception, which are sufficient for your programming requirements.

getopt.getopt method:

This method parses command line options and parameter list. Following is simple syntax for this method:

```
getopt.getopt(args, options[, long_options])
```

Here is the detail of the parameters:

- **args**: This is the argument list to be parsed.
- **options**: This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
- **long_options**: This is optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.

This method returns value consisting of two elements: the first is a list of **(option, value)** pairs. The second is the list of program arguments left after the option list was stripped.
Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

exception getopt.GetoptError:

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none.The argument to the exception is a string indicating the cause of the error. The attributes **msg** and **opt** give the error message and related option

Consider we want to pass two file names through command line and we also want to give an option to check the usage of the script. Usage of the script is as follows:

```
usage: test.py -i <inputfile> -o <outputfile>
```

Here is the following script to test.py:

```
#!/usr/bin/python
import sys, getopt
def main(argv):
        inputfile = ''
        outputfile = ''
        try:
```

```
                opts, args =getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
        except getopt.GetoptError:
                print 'test.py -i <inputfile> -o <outputfile>'
                sys.exit(2)
        for opt, arg in opts:
                if opt == '-h':
                        print 'test.py -i <inputfile> -o <outputfile>'
                        sys.exit()
                elif opt in ("-i", "--ifile"):
                        inputfile = arg
                elif opt in ("-o", "--ofile"):
                        outputfile = arg
        print 'Input file is "', inputfile
        print 'Output file is "', outputfile

if __name__ == "__main__":
        main(sys.argv[1:])
```

Now, run above script as follows:

```
$ test.py -h
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i BMP -o
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i inputfile
Input file is " inputfile
Output file is "
```

Now type the following address in your web browser and see if you can get an XML file with the following contents:
http://eutils.ncbi.nlm.nih.gov/entrez/eutils/esummary.fcgi?db=nuccore&id=319498500

```
- <eSummaryResult>
  - <DocSum>
      <Id>319498500</Id>
      <Item Name="Caption" Type="String">HQ791361</Item>
    - <Item Name="Title" Type="String">
        Uncultured organism clone ELU0124-T310-S-NI_000142 small subunit ribosomal RNA gene, partial sequence
      </Item>
      <Item Name="Extra" Type="String">gi|319498500|gb|HQ791361.1|[319498500]</Item>
      <Item Name="Gi" Type="Integer">319498500</Item>
      <Item Name="CreateDate" Type="String">2011/12/31</Item>
      <Item Name="UpdateDate" Type="String">2012/07/30</Item>
      <Item Name="Flags" Type="Integer">0</Item>
      <Item Name="TaxId" Type="Integer">155900</Item>
      <Item Name="Length" Type="Integer">1439</Item>
      <Item Name="Status" Type="String">live</Item>
      <Item Name="ReplacedBy" Type="String"/>
      <Item Name="Comment" Type="String"> </Item>
    </DocSum>
  </eSummaryResult>
```

(See http://www.ncbi.nlm.nih.gov/books/NBK25500/ for detailed description about NCBI's eutils)

To parse this type of data one can use python's **xml.dom.minidom** package
Say we have an XML file **items.xml** with the following contents:

```
<data>
        <items>
                <item name="item1"></item>
                <item name="item2"></item>
                <item name="item3"></item>
                <item name="item4"></item>
        </items>
</data>
```

In python we can write the following lines of code:

```
#!/usr/bin/python
from xml.dom import minidom
xmldoc = minidom.parse('items.xml')
itemlist = xmldoc.getElementsByTagName('item')
print len(itemlist)
print itemlist[0].attributes['name'].value
for s in itemlist :
        print s.attributes['name'].value
```

which will give the following output

```
4
item1
item1
item2
item3
item4
```

We are now going to change the test.py script we created before to accept only one argument –b as a tab-delimited blast output file:

```
#!/usr/bin/python
import sys, getopt
def usage():
        print 'Usage:'
        print '\tpython test.py -b <blast_file>'
def main(argv):
        blastfile = ''
        try:
                opts, args =getopt.getopt(argv,"hb:",["blast_file="])
        except getopt.GetoptError:
                usage()
                sys.exit(2)
        for opt, arg in opts:
                if opt == '-h':
```

```
                        usage()
                        sys.exit()
                elif opt in ("-b", "--blast_file"):
                        blastfile = arg
        print 'Input file is "', blastfile

if __name__ == "__main__":
        main(sys.argv[1:])
```

Next, we will try to incorporate basic regular expression based searching in our python code. In order to do that, we will first learn about some of the functions in **re** package

The *match* Function:

This function attempts to match RE *pattern* to *string* with optional *flags*. Here is the syntax for this function:

```
re.match(pattern, string, flags=0)
```

Here is the description of the parameters:

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern at the beginning of string.
flags	You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.match* function returns a **match** object on success, **None** on failure. We would use *group(num)* or *groups()* function of **match** object to get matched expression.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num)
groups()	This method returns all matching subgroups in a tuple (empty if there weren't any)

```
#!/usr/bin/python
import re
line = "Cats are smarter than dogs"
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)
if matchObj:
        print "matchObj.group() : ", matchObj.group()
        print "matchObj.group(1) : ", matchObj.group(1)
        print "matchObj.group(2) : ", matchObj.group(2)
else:
        print "No match!!"
```

When the above code is executed, it produces following result:

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

The *search* Function:

This function searches for first occurrence of RE *pattern* within *string* with optional *flags*. Here is the syntax for this function:

```
re.search(pattern, string, flags=0)
```

Here is the description of the parameters:

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern anywhere in the string.
flags	You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.search* function returns a **match** object on success, **None** on failure. We would use *group(num)* or *groups()* function of **match** object to get matched expression.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num)
groups()	This method returns all matching subgroups in a tuple (empty if there weren't any)

```
#!/usr/bin/python
import re
line = "Cats are smarter than dogs";
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)
if matchObj:
        print "matchObj.group() : ", matchObj.group()
        print "matchObj.group(1) : ", matchObj.group(1)
        print "matchObj.group(2) : ", matchObj.group(2)
else:
        print "No match!!"
```

When the above code is executed, it produces following result:

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

Matching vs Searching:

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search** checks for a match anywhere in the string (this is what Perl does by default).

```
#!/usr/bin/python
import re
line = "Cats are smarter than dogs";
matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
        print "match --> matchObj.group() : ", matchObj.group()
else:
        print "No match!!"
matchObj = re.search( r'dogs', line, re.M|re.I)
if matchObj:
        print "search --> matchObj.group() : ", matchObj.group()
else:    print "No match!!"
```

When the above code is executed, it produces the following result:

```
No match!!
search --> matchObj.group() :  dogs
```

Search and Replace:

Some of the most important **re** methods that use regular expressions is **sub**.

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method would return modified string. Following is the example:

```
#!/usr/bin/python
import re
phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

When the above code is executed, it produces the following result:

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

Regular-expression Modifiers - Option Flags:

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these:

Modifier	Description
re.I	Performs case-insensitive matching.
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.X	Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

Regular-expression patterns:

Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python:

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
(?imx)	Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?-imx)	Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?: re)	Groups regular expressions without remembering matched text.
(?imx: re)	Temporarily toggles on i, m, or x options within parentheses.
(?-imx: re)	Temporarily toggles off i, m, or x options within parentheses.
(?#...)	Comment.
(?= re)	Specifies position using a pattern. Doesn't have a range.
(?! re)	Specifies position using pattern negation. Doesn't have a range.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word characters.
\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\1...\9	Matches nth grouped subexpression.
\10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Regular-expression Examples

Literal characters:

Example	Description
python	Match "python".

Character classes:

Example	Description
[Pp]ython	Match "Python" or "python"
rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit

Special Character Classes:

Example	Description
.	Match any character except newline
\d	Match a digit: [0-9]
\D	Match a nondigit: [^0-9]
\s	Match a whitespace character: [\t\r\n\f]
\S	Match nonwhitespace: [^ \t\r\n\f]
\w	Match a single word character: [A-Za-z0-9_]
\W	Match a nonword character: [^A-Za-z0-9_]

Repetition Cases:

Example	Description
ruby?	Match "rub" or "ruby": the y is optional
ruby*	Match "rub" plus 0 or more ys
ruby+	Match "rub" plus 1 or more ys
\d{3}	Match exactly 3 digits
\d{3,}	Match 3 or more digits
\d{3,5}	Match 3, 4, or 5 digits

Nongreedy repetition (matches the smallest number of repetitions) :

Example	Description
<.*>	Greedy repetition: matches "<python>perl>"

<.*?>	Nongreedy: matches "<python>" in "<python>perl>"

Grouping with parentheses:

Example	Description
\D\d+	No group: + repeats \d
(\D\d)+	Grouped: + repeats \D\d pair
([Pp]ython(,)?)+	Match "Python", "Python, python, python", etc.

Backreferences (matches a previously matched group again):

Example	Description
([Pp])ython&\1ails	Match python&pails or Python&Pails
(['"])[^\1]*\1	Single or double-quoted string. \1 matches whatever the 1st group matched . \2 matches whatever the 2nd group matched, etc.

Alternatives:

Example	Description
python|perl	Match "python" or "perl"
rub(y|le))	Match "ruby" or "ruble"
Python(!+|\?)	"Python" followed by one or more ! or one ?

Anchors (needs to specify match position):

Example	Description
^Python	Match "Python" at the start of a string or internal line
Python$	Match "Python" at the end of a string or line
\APython	Match "Python" at the start of a string
Python\Z	Match "Python" at the end of a string
\bPython\b	Match "Python" at a word boundary
\brub\B	\B is nonword boundary: match "rub" in "rube" and "ruby" but not alone
Python(?=!)	Match "Python", if followed by an exclamation point
Python(?!!)	Match "Python", if not followed by an exclamation point

Special syntax with parentheses:

Example	Description
R(?#comment)	Matches "R". All the rest is a comment
R(?i)uby	Case-insensitive while matching "uby"
R(?i:uby)	Same as above
rub(?:y|le))	Group only without creating \1 backreference

Final program:
Now we are going to incorporate all that we have learned so far and make a function **process_record**, which accepts as an input a single record of the tab-delimited file. From the reference hit, it first extracts the GI accession number, and then uses NCBI's eutils to get the summary record for this GI accession number from the NCBI's **nucleotide** database. It then extracts the **Title** and **TaxId** from the returned XML file and appends them to the end of the record. Furthermore, we do one more optimization to our program: -b field either accepts the tab-delimited file or a single record from blast output file. This way, we can make use of **GNU Parallel** (http://www.gnu.org/software/parallel/) which is a shell tool for running jobs in parallel on multicore computers, i.e. we will process each blast output record on a separate core (with NCBI eutils, it is not recommended to generate frequent URL searches and it may be possible that our data requests may time out if many users start executing this script in parallel. For this purpose, NCBI eutils also allow batch processing, say 100 GI accession numbers at once. Nevertheless, for demonstration purposes on how to utilize GNU Parallel with python programs, we will execute this code).

```
#!/usr/bin/python
import sys, getopt
import re
from xml.dom import minidom
import urllib

def usage():
        print 'Usage:'
        print '\tpython test.py -b <blast_file>'

def process_record(record):
        params={
                'db':'nucleotide',
        }
        output_flag=1
        title=''
        taxid=''
        gid=int(record.split("|")[1])
        params['id']=gid
        url='http://eutils.ncbi.nlm.nih.gov/entrez/eutils/esummary.fcgi?'+ urllib.urlencode(params)
        data=urllib.urlopen(url).read()

        #parse XML output
        xmldoc=minidom.parseString(data);
        for ind in range(0,xmldoc.getElementsByTagName('Item').length-1):
                if xmldoc.getElementsByTagName('Item')[ind].attributes.item(1).value=='Title':
                        title=xmldoc.getElementsByTagName('Item')[ind].childNodes[0].nodeValue
                if xmldoc.getElementsByTagName('Item')[ind].attributes.item(1).value=='TaxId':
                        taxid=xmldoc.getElementsByTagName('Item')[ind].childNodes[0].nodeValue
        if output_flag==1:
                print record.rstrip('\n')+"\t"+re.sub('\s+','_',title)+"\t"+taxid
        else:
                print record.rstrip('\n')+"\t"+re.sub('\s+','_',title).split("_")[0]+"_"+re.sub('\s+','_',title).split("_")[1]+"\t"+taxid


def main(argv):
        blastfile = ''
        try:
                opts, args =getopt.getopt(argv,"hb:",["blast_file="])
        except getopt.GetoptError:
                usage()
                sys.exit(2)
        for opt, arg in opts:
                if opt == '-h':
                        usage()
                        sys.exit()
                elif opt in ("-b", "--blast_file"):
```

```
                        blastfile = arg

        # Check if it is a file or a record
        if re.findall(r'gi\|(\w.*?)\|',blastfile):
                process_record(blastfile)
        else:
                ins=open(blastfile,"r")
                for line in ins:
                        process_record(line)
                ins.close()

if __name__ == "__main__":
        main(sys.argv[1:])
```

When we execute the above program, we get the following output:

```
$ python test.py -b test.out
C1_Desulfovibrionaceae      gi|380865194|gb|JQ244906.1|  100.00    254       0         0         1         254       300       553       3e-129     470
        Uncultured_Desulfovibrio_sp._clone_NDSV59_16S_ribosomal_RNA_gene,_partial_sequence    167968
C2_Thermoanaerobacterales   gi|45056518|gb|JX988421.1|  100.00    254       0         0         1         254       499       752       3e-129     470
        Caldicellulosiruptor_sp._DIB107C_16S_ribosomal_RNA_gene,_partial_sequence    1298323
C3_Verrucomicrobiaceae      gi|444304012|ref|NR_074436.1|          100.00    254       0         0         1         254       492       745       3e-129     470 Akkermansia_muciniphila_ATCC_BAA-835_strain_ATCC_BAA-
835_16S_ribosomal_RNA,_complete_sequence        349741
C4_Bacteroidaceae  gi|444304091|ref|NR_074515.1|          100.00    254       0         0         1         254       535       788       3e-129     470
        Bacteroides_vulgatus_ATCC_8482_strain_ATCC_8482_16S_ribosomal_RNA,_complete_sequence  435590
C5_Nostocaceae     gi|402768260|gb|JQ700517.1|  100.00    254       0         0         1         254       354       607       3e-129     470      Trichormus_variabilis_BAC_9894_16S_ribosomal_RNA_gene,_partial_sequence
        1229007
C6_Hydrogenothermaceae      gi|444304133|ref|NR_074557.1|          100.00    254       0         0         1         254       494       747       3e-129     470
        Sulfurihydrogenibium_sp._YO3AOP1_strain_YO3AOP1_16S_ribosomal_RNA,_complete_sequence  436114
C7_Rhodobacteraceaegi|471258250|gb|KC534156.1|  100.00    254       0         0         1         254       410       663       3e-129     470
        Sulfitobacter_pontiacus_strain_VSW034_16S_ribosomal_RNA_gene,_partial_sequence       60137
C8_Fusobacteriaceaegi|304557157|gb|HM347061.1|  100.00    253       0         0         1         253       466       718       1e-128     468
        Fusobacterium_nucleatum_subsp._polymorphum_strain_WAL_12230_16S_ribosomal_RNA_gene,_partial_sequence     76857
C9_Shewanellaceae  gi|459652782|gb|KC140316.1|  100.00    254       0         0         1         254       485       738       3e-129     470      Gamma_proteobacterium_BAL382_16S_ribosomal_RNA_gene,_partial_sequence
        1301889
C10_Aquificaceae   gi|444439645|ref|NR_074960.1|          100.00    254       0         0         1         254       530       783       3e-129     470
        Hydrogenobaculum_sp._Y04AAS1_strain_Y04AAS1_16S_ribosomal_RNA,_complete_sequence      380749
C11_Bacteroidaceae gi|444303855|ref|NR_074277.1|          100.00    254       0         0         1         254       537       790       3e-129     470      Bacteroides_thetaiotaomicron_VPI-5482_strain_VPI-
5482_16S_ribosomal_RNA,_complete_sequence       226186
C12_Thermoanaerobacteraceae  gi|444439745|ref|NR_075060.1|          100.00    254       0         0         1         254       584       837       3e-129     470 Thermoanaerobacter_brockii_subsp._finnii_Ako-1_strain_Ako-
1_16S_ribosomal_RNA,_complete_sequence509193
```

In order to run the program through GNU parallel, we will use the **cat** command and pipe it's output to parallel. **-k** switch is to keep the output file in the same order, **-- jobs** specifies the number of cores to use (0 implies all available cores), **{}** is the argument (blast record) received from the cat command (See GNU Parallel's tutorial: http://www.gnu.org/software/parallel/parallel_tutorial.html)

```
parallel -k --jobs 0 python test.py -b {}
```

Now we will use the **time** command to see the difference in execution times between parallel-mode vs sequential-mode and also use > /dev/null 2>&1 to discard all the output since we are only interested in script timings and not the output. As you can see, it is 4 seconds vs 1 minute 8 seconds ☺

```
$ time cat test.out | parallel -k --jobs 0 python test.py -b {} > /dev/null 2>&1
real    0m4.135s
user    0m16.928s
sys     2m16.663s

$ time python test.py -b test.out > /dev/null 2>&1
real  1m8.800s
user  0m1.402s
sys   0m0.243s
```