同济大学 TONGJI UNIVERSITY

生物信息学系 DEPARTMENT OF BIOINFORMATICS

Section 9

Python: Building Program Pipelines Writing Good Programs Bioinformatics Applications

Yong Zhang, Ph. D

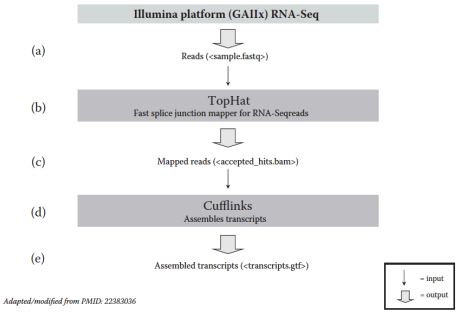
School of Life Science and Technology
Tongji University

Apr 28, 2019

yzhang@tongji.edu.cn

Story: building an NGS pipeline

- An example pipeline for RNA-seq



Adapted/modified from PMID: 22383036

Example Python session

```
import os

tophat_output_dir = '/home/RNA-seq/tophat'
tophat_output_file = 'accepted_hits.bam'
bowtie_index_dir = '/home/RNA-seq/index'
cufflinks_output_dir = '/home/RNA-seq/cufflinks'
cufflinks_output_file = 'transcripts.gtf'
illumina_output_file = 'sample.fastq'

tophat_command = 'tophat -o %s %s %s' %\
    (tophat_output_dir, bowtie_index_dir,\
    illumina_output_file)
os.system(tophat_command)

cufflinks_command = 'cufflinks -o %s %s%s%s' %\
    (cufflinks_output_dir, tophat_output_dir, os.sep,\
    tophat_output_file)
os.system(cufflinks_command)
```

os.system()

```
import os

tophat_output_dir = '/home/RNA-seq/tophat'
tophat_output_file = 'accepted_hits.bam'
bowtie_index_dir = '/home/RNA-seq/index'
cufflinks_output_dir = '/home/RNA-seq/cufflinks'
cufflinks_output_file = 'transcripts.gtf'
illumina_output_file = 'sample.fastq'

tophat_command = 'tophat -o %s %s %s' %\
    (tophat_output_dir, bowtie_index_dir,\
    illumina_output_file)
os.system(tophat_command)

cufflinks_command = 'cufflinks -o %s %s%s%s' %\
    (cufflinks_output_dir, tophat_output_dir, os.sep,\
    tophat_output_file)
os.system(cufflinks_command)
```

os.system()

- When copy and paste the *os.system()* string argument to a prompt in a UNIX shell, you will get the same result as running *os.system()* from a program with that argument.
- The *os* module provides a Python interface to the operating system, and *the os.system()* method makes the operating system execute the command line specified in the method argument.
- The argument of *os.system()* is a single string consisting of the UNIX shell command needed to run the program.
- For the sake of clarity, the parts of these strings are concatenated from the variables previously defined.

What is a pipeline?

- Pipelines are scripts used to connect programs to each other.
- A program that runs another program is also called a wrapper.

Writing a program wrapper

- The *os.path.exists()* method returns *True* if its argument is an existing path on your computer.

```
tophat_dir = '/home/RNA-seq/tophat'
index_dir = '/home/RNA-seq/index'
if os.path.exists(tophat_dir) and os.path.exists(index_dir):
    os.system('tophat -o ' + tophat_dir + ' ' + index_dir + \
      ' sample.fastq')
else:
    print 'You have to create tophat and/or index\
        directories before running your wrapper'
```

Writing a program wrapper

- You can assign all path variables in a separate module, for example, *pathvariables.py*:

```
tophat_dir = '/home/RNA-seq/tophat'
index_dir = '/home/RNA-seq/index'
cufflinks_dir = '/home/RNA-seq/cufflinks'
```

- Here is the previous example:

```
from pathvariables import tophat_dir, index_dir
if os.path.exists(tophat_dir) and os.path.exists(index_dir):
    os.system('tophat -o ' + tophat_dir + ' ' + index_dir +\
        sample.fastq')
else:
    print 'You have to create tophat and/or index\
        directories before running your wrapper'
```

Using command-line parameters

- Instead of changing the program or editing a file, you can use command-line parameters.

```
python ngs_pipeline.py dataset_one.fastq
python ngs_pipeline.py dataset_two.fastq
```

- You can try the following Python code (create a file *arguments.py*):

```
import sys
print sys.argv
```

- Try calling this program with different arguments:

```
python arguments.py Hello
python arguments.py 1 2 3
```

- If you need to manage many different command-line options, consider using the *optparse* module.

Module attributes

- Some attributes of Python modules are module specific, whereas three in particular refer to modules in general: *__file__*, *__doc__*, and *__name__*.
  - The *__file__* attribute returns the path of the module.
  - The *__doc__* attribute returns the module documentation, if present.
  - The *__name__* attribute returns the name of the imported file without the *.py* suffix if the module is imported and the string '*__main__*' if the module is executed.

Testing modules: *if __name__ == '__main__'*

- If you insert the following condition in your programs, <*statements*> will be executed only if the module is run from the command line and not imported by means of an *import* statement.

```
if __name__ == '__main__':
        <statements>
```

Reading files from directories

- The function *os.listdir(directory)* reads all files from a given directory into a list. This is particularly useful in a pipeline if you want to automatically run a program on all the files in a given folder:

```
import os
for filename in os.listdir('data/'):
        os.system('<my_program>%s'%(filename))
```

Changing directories

- Some third-party programs require to be started from a particular directory. You would go to that directory using the *os.chdir()* function:

```
os.chdir('../data/')
```

- You can determine the current directory with the *os.getcwd()* function:

```
print os.getcwd()
```

File / directories manipulations

- If a file needs to be deleted, you can do this by using *os.remove(filename)*.

```
if os.path.exists('log.txt'):
       os.remove('log.txt')
```

- Functions to make and remove directories are *os.mkdir()* and *os.rmdir()*, respectively.

Examples

- Running T-COFFEE.
  - The following code is a wrapper for the multiple sequence alignment T-COFFEE algorithm.

```
import sys
import os
sys.path.append('pathmodules/')
from tcoffeevariables import tcoffeeout
cmd = 't_coffee -in="file.fasta" -run_name="' + \
    tcoffeeout + 'tcoffe.aln" -output=clustalw')
os.system(cmd)
```

  - Variables are assigned in the *tcoffeevariable.py* module, which is stored in the *pathmodules* directory (which is a subdirectory of the wrapper directory).
  - *-in* is the option for the input file, *-run_name* is the option to assign a name to the output file, and *-output* is the option to set the output format (*clustalw*, in this case).

Examples

- Write a wrapper that uses *bl2seq* to align two sequences, the Uniprot ACs of which are command-line arguments.
  - The UNIX command to run the wrapper with two Uniprot ACs as arguments is

    python Bl2seqWrapper.py F1B2B3 E2CXB4

Examples

```
import sys
import os
import urllib2

sys.path.append('pathmodules/')
from blastvariables import *

def run_blastp(seq1, seq2):
    os.system("blastp -query " + input_seq + seq1 +
              ".fasta -subject " + input_seq + seq2 +
              ".fasta -out " + seq1 + "-" + seq2 + ".aln")

def get_seq(seq1, seq2):
    for seq in (seq1, seq2):
        url = 'http://www.uniprot.org/uniprot/' + seq + '.fasta'
        handler = urllib2.urlopen(url)
        fasta = handler.read()
        out = open(input_seq + seq + '.fasta', 'w')
        out.write(fasta)
        out.close()
```

Examples

```
if __name__ == '__main__':
    try:
        seq1 = sys.argv[1]
        seq2 = sys.argv[2]
    except:
        print 'usage: BlastpWrapper.py seq1-UniprotAC \
            seq2-UniprotAC'
        raise SystemExit
    else:
        if os.path.exists(input_seq + seq1 + '.fasta') \
and os.path.exists(input_seq + seq2 + '.fasta'):
            run_blastp(seq1, seq2)
        else:
            get_seq(seq1, seq2)
            run_blastp(seq1, seq2)
```

Example programming project

Dear coworker,

At the conference last week, I had a great idea I'll tell you about later. I made some BLAST queries with a couple of sequences of Phenylalanine-tRNA-synthetases (PheRS) from different organisms. The results need some cleanup, however.

I saved the sequences in a set of files in FASTA format in the same directory. Each file contains many sequences in the following format:

> gi|sequence name|species name
AMINQACIDSEQUENCE

Because the sequence diversity of the aminoacyl-tRNA synthetase family is very high, there are many sequences from other aaRS proteins among the results. We therefore need to filter out sequences that do not have the right tRNA specificity (marked by Phenylalanine, Phe etc. in the sequence name). Ah, and there might be duplicate hits in the files that need to be removed, too.

Example programming project

I think the results fit well into your project. Could you please find all the sequences for Phe and put them into a single FASTA file? We could then pass it to an alignment program. Your new programming skills might come in handy for that.

Best regards,
Your supervisor

P.S.: If this works, we will probably do the same thing for the other 19 amino acids as well.

The dogma of programming

- First, make it work.
- Second, make it nice.
  - Breaking up a programming task into smaller steps.
  - Formatting source code.
  - Using comments.
  - Keeping track of changes electronically.
  - Sharing your program with fellow scientists.
  - Improving a program iteratively.
- Third, and only if it is really necessary, make it fast.
  - You need a deeper knowledge about algorithmics.

Dividing a project into smaller tasks

- From the supervisor's email, the following program purpose can be extracted:
  - Clean up FASTA files with aaRS protein sequences so that they can be used to build a sequence alignment.
- What is the input?
  - The program reads a directory with many FASTA files containing protein sequences. The sequence entries in the files have the format:

    > gi|sequence name|species name
    AMINQACIDSEQUENCE

Dividing a project into smaller tasks

- What is the output?
  - The program writes a single FASTA file with aaRS sequences.
    - The format of the FASTA file should be the same as it is in the input.
    - The order of the sequences in the output does not matter.
    - The format described for the FASTA output must be readable by the program used for creating a sequence alignment.

Dividing a project into smaller tasks

- What should happen between input and output?
  - Remove all sequences that do not have Phenylalanine or Phe in its name.
  - Remove all duplicate sequence entries (identical sequences).
    - What exactly does "duplicate entries" mean? Does it mean that the sequence is identical or that the description is identical? Or both?
- Writing down things the program should do is called *collecting requirements*.
  - Having clearly written requirements helps you to communicate about the program with the people who need it (including yourself) and with people you ask for advice and to track what you have already completed.

Split a program into functions and classes

```
def read_fasta_files(directory):
    '''
    Reads a directory with many FASTA files containing
    protein sequences.
    '''
    pass

def filter_phe(sequences):
    '''
    Removes all sequences that do not have
    Phenylalanine or Phe in their name.
    '''
    pass

def remove_duplicates(sequences):
    '''
    Removes all sequence records, having an identical
    sequence.
    '''
    pass
```

Split a program into functions and classes

```
def write_fasta(sequences, filename):
    '''
    Writes a single FASTA file with aaRS sequences.
    '''
    pass

if __name__ == '__main__':
    INPUT_DIR = 'aars/'
    OUTPUT_FILE = 'phe_filtered.fasta'
    seq = read_fasta_files(INPUT_DIR)
    filter_phe(seq)
    remove_duplicates(seq)
    write_fasta(seq, OUTPUT_FILE)
```

Writing well-formatted code

- You should add a space after commas and arithmetic operators.
- Variables in functions should have lowercase names.
- Constants in modules should have uppercase names.
- After each function there should be two empty lines.
- Each function should have a documentation string as a comment.

Docstrings & comments

- Documentation strings or *docstrings* are important components of formatting conventions.
- Each function and class should have a triple-quoted comment in the line right after the function or class definition.

```
class AARSFilter(object):
    '''
    Reads a set of FASTA files and removes duplicate
    sequence entries.
    '''
    pass
```

- Using comments (text preceded by a # that is ignored by the interpreter) to describe a line or a block of code makes your programs much more readable.

Using a repository to control program versions

- Repositories record changes you apply to your program over time together with short remarks.
- At any point you can return to an older version stored in the repository or switch back and forth between versions.
- There are many programs that allow you to control versions of program code:
  - Mercurial (http://mercurial.selenic.com/wiki/Mercurial)
  - GIT (http://git-scm.com)
  - Bazaar (http://bazaar.canonical.com/en/)
  - SVN (http://subversion.apache.org)

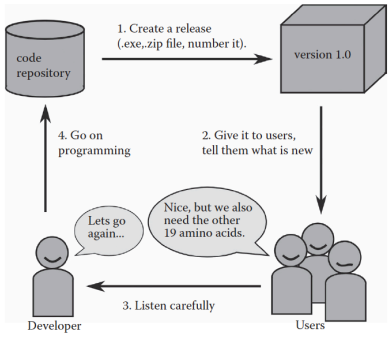
Releasing your program to other people

- To distribute a program, you create a stable version called a *release.*
- You should set a version number for the release.
- You should write a short README.TXT file, covering the following points:
  - What is the name and the version of the program?
  - What is your name and how can you be contacted?
  - What is the program good for?
  - How can the program be used? A good idea is to include a simple command-line example.
  - Under what conditions can the program be distributed?
    - "All rights reserved", "Distributed under the conditions of the Python license", etc.

Releasing your program to other people

- More sophisticated approaches to creating releases include the following:
  - Use the Python *distutils* library to autogenerate an install script.
  - Use the *py2exe* program to create executable files for Windows that don't require your users to install Python.
  - Manage releases on a sourceforge (http://sourceforge.net) or github (https://github.com) account.

The cycle of software development



The cycle of software development

- **Plan**. You first need to have a rough idea about what your program should do.
- **Program**. Next you need to implement the plan. Create the functions, classes, and modules that realize your initial idea.
- **Prove**. Expose your program to a real-world situation.

Creating your own module

- A better strategy for bigger programs is to split the code into several files and then import them.
- To clean up your program with modules, you should:
  - Collect all functions for file parsing in one module.
  - Collect all functions for creating output in one module.
  - Create a module for constants like filenames and parameters.
  - Assign the paths of your data files to variable names in a separate module, say *my_paths*, and save it in a given directory, say *my_modules/*

```
import sys
sys.path.append('/Users/kate/my_modules/')
import my_paths
```

Creating your own package

- A package in Python is a folder with Python modules. You can group together modules into packages by storing them in the same place.
- Python has a list of default directories where it looks for modules and packages. These include the current directory and the *site-packages/* folder.
  - You can see and modify the complete list of directories for modules and packages using *sys.path*.
  - you can add Python directories to the *PYTHONPATH* environment variable.

Creating your own package

- To make a package importable, you also need to add a file *__init__.py*, which may be empty.
- You can use *import* to use single modules or the entire package.
  - You have a directory *neuroimaging/* with three Python files: *neuron_count.py*, *shrink_images.py*, *__init__.py*.
  - All of the following import statements work:

```
import neuroimaging
from neuroimaging import neuron_count
from neuroimaging.shrink_images import *
```

  - The *__init__.py* file is imported automatically by all three commands.

Example 1 (FASTA format parser)

```
def read_fasta(path):
    """
    Read FASTA file into a list of lists.
    Each inner list contains a name and a sequence.
    """
    out = []
    last_record = -1
    for line in open(path).xreadlines():
        if line.startswith(">"):
            contig_name = line.split()[0]
            contig_seq = ""
            out.append([contig_name, contig_seq])
        else:
            out[last_record][1] += line.strip()
    return out

fasta_out = read_fasta('yeast_gene.fa')
```

Example 2 (convert DNA to RNA)

```
import re
# import regular expression module

# setting the DNA string
myDNA = 'ACGTTGCAACGTTGCAACGTTGCA'

# assigning a new regex and compiling it to find all Ts
regexp = re.compile('T')

# create a new string tha will receive the regex result with Us replacing Ts
myRNA = regexp.sub('U', myDNA)

print myRNA
```

Example 2 (convert DNA to RNA)

```
#! /usr/bin/python

myDNA = 'ACGTTGCAACGTTGCAACGTTGCA'
myRNA = myDNA.replace('T', 'U')

print myRNA
```

Example 3 (finding motifs in DNA sequences)

```
import re
# we use the RegEx module

#still keep the file fixed
dnafile = "AY162388.seq"

#opening the file, reading the sequence and storing in a list
seqlist = open(dnafile, 'r').readlines()

#let's join the the lines in a temporary string
temp = ''.join(seqlist)

#assigning our sequence, with no carriage returns to our final variable/object
sequence = temp.replace('\n', '')

#we start to deal with user input first we use a boolean variable to check for valid
    input
inputfromuser = True
```

Example 3 (finding motifs in DNA sequences)

```
#while loop: while there is an motif larger than 0 the loop continues
while inputfromuser:
    #raw_input received the user input as string
    inmotif = raw_input('Enter motif to search: ')
    #now we check for the size of the input
    if len(inmotif) >= 1:
        #we compile a regex with the input given
        motif = re.compile('%s' % inmotif)
        #looking to see if the entered motif is in the sequence
        if re.search(motif, sequence):
            print 'Yep, I found it'
        else:
            print 'Sorry, try another one'
    else:
        print 'Done, thanks for using motif_search'
        inputfromuser = False
```

Example 4 (generates a random DNA sequence)

```
#!/usr/bin/python

import sys, random

length = int(sys.argv[1])

dnaseq = ['A', 'T', 'C', 'G']

result = ''
for i in range(length):
    result += random.choice(dnaseq)

print result
```

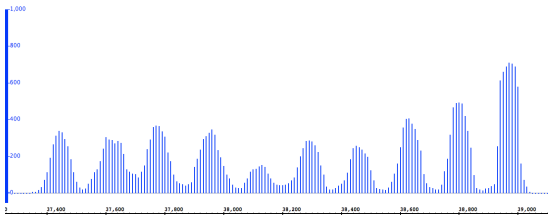
Example 5 (simulate DNA sequence sets)

```
#!/usr/bin/python
import random, sys

def simulate_sequence(length):
    dna = ['A', 'C', 'G', 'T']
    sequence = ''
    for i in range(length):
        sequence += random.choice(dna)
    return sequence

setsize = int(sys.argv[1])
minlength = int(sys.argv[2])
maxlength = int(sys.argv[3])

sequenceset = []
for i in range(setsize):
    rlength = random.randint(minlength, maxlength)
    sequenceset.append(simulate_sequence(rlength))

for sequence in sequenceset:
    print sequence
```

Example 5 (simulate DNA sequence sets)

```
if len(sys.argv) <= 3:
    print "python example_5.py setsize minlenth maxlength"
    sys.exit()
```

Example 6 (nucleosome positioning profile)



Example 6 (Wiggle format)

```
track type=wiggle_0 name="variableStep"
variableStep chrom=chr19 span=150
59304701 10.0
59304901 12.5
59305401 15.0
59305601 17.5
59305901 20.0
track type=wiggle_0 name="fixedStep"
fixedStep chrom=chr19 start=59307401 step=300 span=200
1000
900
800
700
600
500
400
300
```

http://genome.ucsc.edu/goldenPath/help/wiggle.html

Example 6 (BED format to WIG format)

```
#!/usr/bin/python

chrlength = {}
chrlength['Yeast_SGD'] = {'chr1' : 230208,  'chr2' : 813178,  'chr3' : 316617,  'chr4' : 1531918,
                          'chr5' : 576869,  'chr6' : 270148,  'chr7' : 1090947, 'chr8' : 562643,
                          'chr9' : 439885,  'chr10' : 745745,  'chr11' : 666454,  'chr12' : 1078175,
                          'chr13' : 924429,  'chr14' : 784333,  'chr15' : 1091289, 'chr16' : 948062,
                          'chrM' : 85779}
```

Example 6 (BED format to WIG format)

```
def tag2hash(bedfile, spename = 'Yeast_SGD', extension = 73, shift = 36):
    values = {}
    for chrname in chrlength[spename].keys():
        values[chrname] = [0] * chrlength[spename][chrname]

    for line in open(bedfile).xreadlines():
        line = line.strip().split()
        if line[0] not in chrlength[spename].keys():
            continue

        if line[5] == '+':          # Tag in plus strand
            b = int(line[1]) + shift
            e = b + extension
            for k in xrange(max(b, 1), min(e, chrlength[spename][line[0]] + 1)):
                values[line[0]][k - 1] += 1

        elif line[5] == '-':        # Tag in minus strand
            e = int(line[2]) - shift
            b = e - extension
            for k in xrange(max(b, 1), min(e, chrlength[spename][line[0]] + 1)):
                values[line[0]][k - 1] += 1
        else:
            continue
    return values
```

Example 6 (BED format to WIG format)

```
def tag_to_wig(hash_values, wigfile, step = 1, spename = 'Yeast_SGD'):
    fileout = open(wigfile, 'w')
    namelist = chrlength[spename].keys()
    namelist.sort()
    for name in namelist:
        print >>fileout, "track type=wiggle_0 name=" + name
        print >>fileout, "fixedStep  chrom=" + name + "  start=1  step=" + str(step)
        for k in xrange(0, len(hash_values[name]), step):
            print >>fileout, hash_values[name][k]
    fileout.close()

if __name__ == '__main__':
    values = tag2hash('YPGal.bed')
    tag_to_wig(values, 'YPGal.wig', step = 10)
```

Summary

- Managing Your Biological Data with Python
  - Chapter 14. Building Program Pipelines
  - Chapter 15. Writing Good Programs
  - Python codes in https://bitbucket.org/krother/python-for-biologists/src/