Python Programming 2
Regular Expressions, Arrays, Dictionaries, Debugging

Biol4230 Thurs, Feb 11, 2016
Bill Pearson wrp@virginia.edu 4-2818 Jordan 6-057

• String matching and regular expressions:
   import re
   if (re.match('^>', fasta_line)): # match beginning of string
       re_acc_parts = re.compile(r'gi\|\d+\|\w+\|\w+\|\w+\|') # extract parts of a match
       if (re_acc_parts.search(ncbi_acc)) :
           gi,db,acc,id = re_acc_parts.groups()
   file_prefix = re.sub('.aa','',file_name) # substitute

• Working with arrays (lists)
• Dictionaries (dicts[]) and zip()
• python debugging – what is your program doing?
• References and dereferencing – multi-dimensional arrays and dicts

To learn more:
• Practical Computing: Part III – ch. 7 – 10, merging files: ch. 11
• regular expressions:
  – https://docs.python.org/2/howto/regex.html#regex-howto
• Learn Python the Hard Way: learnpythonthehardway.org/book/
• Think Python (collab) www.greenteapress.com/thinkpython/thinkpython.pdf
• Exercises due noon Monday, Feb. 15 (save in biol4230/hwk4)
  1. Download 10: 200 aa and 10: 800 aa random protein sequences from
     http://www.bioinformatics.org/sms2/random_protein.html
  2. For each of 5 scoring matrices from either ssearch (e.g. BL50, BP62, VT160, VT80, VT40) or
     blastp (BL45, BL62, BL80, PAM70, PAM30), calculate the
     average alignment length and percent identity from the best match (one
     average), the second best match (a second average), and the 5th best match (a
     third average) in a search the 10 shorter sequences against SwissProt. The
     result should be 5 sets of 3 average alignment lengths, and 5 sets of three
     average percent identities.
  3. Repeat the analysis, reporting the average alignment length and percent identity
     of the top 10 hits from one of the 10 shorter query searches.
  4. Are the averages you get in step 2 the same as those in step 3? Why might they
     be different?
  5. Repeat step 2 for the longer query sequences, reporting only the averages for
     the best hit.
Regular expressions

>gi|121694|sp|P20432.3|GSTT1_DROME Glutathione S-transferase 1-1

used for string matching, substitution, pattern extraction

- import re
- `r'\^>gi\|\d+\|\w+\|\w+\|'` matches `>gi|121694|sp|P20432.3|GSTT1_DROME ...
- if (re.match(r'\^>gi\', line)): # match  
  re.match(r'\^>gi\|\d+\|\w+\|\w+\|', line) # extract gi with ()  
  gi = re.match.group(1); (  
    (gi, db, acc)  
    = re.match(r'\^>gi\|\d+\|\w+\|\w+\|', line).groups()  
  )  
- re.sub(r'.+ \(\d+\) aa$', '', file) # delete "aa" at end  
- re.sub(r'^>(.*)$', r'\^>\1/', line) # substitution  
- re.sub(r'^>','>>', line, 1)  # same thing (simpler),  
  # substitution is global, use ,1 for once
- `'^'` – beginning of line; `'$'` – end of line

Regular expressions (cont.)

>gi|121694|sp|P20432.3|GSTT1_DROME Glutathione S-transferase 1-1

- 'plaintext'
  - 'one|two' # alternation
    - '(one|two)|three' # grouping with  
      # parenthesis(capture)
  - r'\^>gi\|\d+\)' # 'beginning of line  
    # use r'\|\d+' whenever '\'
  - r'.+ (\d+) aa$' # $ end of line
- 'a*bc' # bc, abc, aabc, ... # repetitions
  - 'a?bc' # abc, bc
  - 'a+bc' # abc, aabc, ...
Regular Expressions, III

Matching classes:
- r'\[^>\]gi\|[0-9]\|[a-z]\|[0-9a-z]\|[0-9]\]|'
  * [a-z] -> class
  * [0-9] -> class
  * [a-z] -> negated class
- r'\[^>\]gi\|\d\|[a-z]\|[w]\]|'
  * \d -> number
  * [a-z] -> string
  * [w] -> word
  * \s -> space
  * \S -> not a space

Capturing matches:
- r'\[^>\]gi\|\d\|[a-z]\|[w]\|.\d\]|'
  .group(1) .group(2) .group(3)
- (gi,db,db_acc) =
  re.match(r'\[^>\]gi\|\d\|[a-z]\|[w]\|',line).groups()

Regular expressions – modifiers

If your regular expression needs a '\' (e.g. '\\', '\d', '\w', '\|'), be sure to prefix with 'r' - r'\[^>\]'r\|\d\|\w\|'

import re
r'\[^>\]gi\|\d\|[a-z]\{2,3\}\|\w\|' #range

rel=re.compile('That',re.I) # re.IGNORECASE
if rel.search("this or that"): re2=re.compile('\[^>\]gi ...',re.M) # treat as multiple lines re3=re.compile('\n',re.S) # treat as single long line with internal 'n's re3.sub('',string) # remove 'n in multiline entry
String expressions (with regular expressions)

if re.match(r'^>gi\',line):
    while (not re.match(r'^>gi\',line)):
        Substitution:
            new_line = re.sub(r'\',':',old_line)
        Pattern extraction:
            (gi,db,db_acc) = re.match(r'^>gi\|([a-z])\|([\w+])',line).groups()
        re.split('\s+', line)  # like sseqid.split()

Working with arrays (lists) I –

• Create array:
    array=[]
    array_str="cat dog piranha"; array = array_str.split(" ")
    array1=range(1,10)
    [1, 2, 3, 4, 5, 6, 7, 8, 9]  # no 10!!!, 9 elements
    array2=range(0,10)
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  # still no 10, but 10 elements
    array3=range(1,20,2)  # second number is max+1
    [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

• Extract/set individual element:
    value=array[1]; value=array[i]
    array[0]=98.6; array[i]=101.4

• Extract/set list of elements (array slice)
    (first, second, third) = array[0:3]  # [start:end-1]

• Python array elements do not have a constant type; array[0] can be a "string" while array[1] is a number.
Working with arrays (lists) II–

months_str = 'Jan Feb Mar Apr ... Dec'
months = split(' ', months_str)
months[0] == 'Jan'; months[3] == 'Apr';

• Add to array (array gets longer, at end or start)
  – add to end of array
    array.append(value)  # array[-1]==value
  – add to beginning, less common, less efficient
    array.insert(0,value)  # array[0] == value
  – (inserts can go anywhere)

• Remove from array (array gets shorter/smaller)
  first_element = array.pop(0)
  last_element = array.pop();

• Parts of an array (slices, beginning, middle, end)
  second_third_array = array[1:3] = array[start:end+1]

Working with arrays (lists) III–

• Array assignments are *aliases*, NOT copies:

  >>> array2
  [1, 'second', 5, 7, 9, 11, 13, 15, 17, 19]
  >>> array2_notcopy = array2
  >>> array2_notcopy.pop()
  19
  >>> array2
  [1, 'second', 5, 7, 9, 11, 13, 15, 17]
  >>> array2_notcopy.pop(0)
  1
  >>> array2_notcopy
  ['second', 5, 7, 9, 11, 13, 15, 17]
  >>> array2
  array2
  ['second', 5, 7, 9, 11, 13, 15, 17]

• To create a genuine copy, "list comprehensions"

  array2_copy = [ x for x in array2 ]
Working with arrays (lists) IV–

- Two functions: array.sort() and sorted(array)
  ```python
  num_array = [2.48, 1.72, 2.15, 1.55]
  num_array.sort() # .sort() sorts in place
  [1.55, 1.72, 2.15, 2.48]
  num_array.sort(reverse=True)
  [2.48, 2.15, 1.72, 1.55]
  ```
  ```python
  str_array = ['Bat', 'Aardvark', 'Dog', 'Cat']
  str_array.sort() # or sorted(str_array)
  ['Aardvark', 'Bat', 'Cat', 'Dog']
  ```

- Build new array: list comprehension
  ```python
  new_array = [ x*x for x in num_array ]
  ```

- Build a subset of an array: list comprehension
  ```python
  no_a_animal = [ x for x in str_array if not re.search('[aA]',x) ]
  no_a_animal == ['Dog']
  ```

---

Python dictionaries (dicts) –
Arrays with names, not positions

```python
months = ['Jan', 'Feb', 'Mar', 'Apr', ...] # list
months[0] == 'Jan'; months[3]=='Apr'
month_days = [31, 28, 31, 30, ...] # month_days[1] == 28
month_day_dict={'Jan':31,'Feb':28,'Mar':31,'Apr':30,...}
# alternatively:
month_day_dict=dict(zip(months, month_days))
month_day_dict[Feb]==28; month_day_dict.get('Feb')==28
month_day_dict[XYZ]==error; month_day_dict.get('XYZ')==None
```

```python
data_dict = {}
data_dict[key] = value;
for key in data_dict.keys():
    print key, data_dict[key] # note keys are not ordered
```

Practical Computing, Ch 9, pp. 151-158
python dicts (cont.)

- dict keys can be checked with 'in' or '.get()'
  
  ```python
  'Meb' in month_day_dict == False
  month_day_dict.get('Meb') == None
  ```

- "in" is convenient for checking for duplicates, e.g.
  ```python
  if ('P09488' in acc_dict): #do something
  else: acc_dict['P09488']= evalue # now it is defined
  ```

- Unlike an array=[], a dict={} is unordered:
  ```python
  for month in months:  # prints months in order;
  for month in month_dict.keys():
      # could be Dec, Mar, Sep. etc.
  If you need the elements of a dict in order, either keep a separate array
  (months), or make a 2-D dict with an index (see next)
  ```

---

Array parts / Dict parts

```
<table>
<thead>
<tr>
<th>qseqid</th>
<th>sseqid</th>
<th>pident</th>
<th>len</th>
<th>mis</th>
<th>gp</th>
<th>qs</th>
<th>ss</th>
<th>se</th>
<th>evalue</th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp</td>
<td>GSTM1_HUMAN</td>
<td>sp</td>
<td>GSTM1_HUMAN</td>
<td>100.00</td>
<td>218</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>218</td>
<td>218</td>
</tr>
<tr>
<td>sp</td>
<td>GSTM1_HUMAN</td>
<td>sp</td>
<td>GSTM4_HUMAN</td>
<td>86.70</td>
<td>218</td>
<td>29</td>
<td>0</td>
<td>1</td>
<td>218</td>
<td>218</td>
</tr>
</tbody>
</table>
```

Python loves arrays (lists). Most python programs NEVER refer to
individual data elements with an index (no array[i]).

How to easily isolate the information desired (sseqid; evalue)?
How do we refer to the data?

```python
data = line.split(' \t ')
```

1) **Array slice:**

- data[0], data[1], data[3], ...
- or isolate the ones you need: (array slice, just pick what you want)

```python
hit_data = [data[0:4], data[10]]
hit_data = [data[0:4], data[-2]]
```
Array parts / Dict parts

<table>
<thead>
<tr>
<th>seqid</th>
<th>sseqid</th>
<th>pident</th>
<th>len</th>
<th>mis</th>
<th>gp</th>
<th>qs</th>
<th>ss</th>
<th>se</th>
<th>evalue</th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp</td>
<td>GSTM4_HUMAN</td>
<td>100.00</td>
<td>218</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>218</td>
<td>1</td>
<td>218</td>
<td>7e-127</td>
</tr>
<tr>
<td>sp</td>
<td>GSTM1_HUMAN</td>
<td>86.70</td>
<td>218</td>
<td>29</td>
<td>0</td>
<td>1</td>
<td>218</td>
<td>1</td>
<td>218</td>
<td>3e-112</td>
</tr>
</tbody>
</table>

```
data = line.split('\t')
hit_data = [data[1], data[10]];
The problem with arrays is that you need to remember where the
data is. Is data[10] the evalue, or the bitscore?
```

2) dict:

```
hit_dict =
  dict(zip(['qseqid','sseqid', ... 'evalue', 'bits'],data))
or
field_name_str = 'qseqid sseqid ... evalue bits'
field_names = field_name_str.split(' '
hit_dict = dict(zip(field_names,data))
hit_dict = dict(zip(field_names,line.split('\t'))) 
print '"t'.join([hit_dict[sseqid], str(hit_dict[evalue])])
```

python debugging

1. Fix syntax errors (undeclared variables, missing ':' or '()')
```
python script_name.pl
```

2. If the program does not work (or prints nonsense), or if you just want to watch it work, add:
```
import pdb; pdb.set_trace() # then
script_name.pl # immediately stops for debugging
- 'n' : next (over functions)
- 's' : step (into functions)
- 'b' : break # 'disable #' to remove break #
- 'c' : continue
- 'q' : quit
- 'h' : help
```

1. The debugger is a python interpreter, so you can try anything you like.
```
(Pdb) print re.split('s+','this is a short string')
['thi', ' i', ' a', 'hort', 'tring']
```
#!/seqprg/bin/python

import pdb; pdb.set_trace()  # load the debugger

month_str = 'Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec'
months = month_str.split(' ')
month_days = [31, 28, 31, 30, 31, 30, 31, 31, 31, 30, 31]

month_dict = {}

for i in range(len(months)):
    month_dict[months[i]] = month_days[i]

for month in months:  # line 14
    print month

for month in months:  # line 17
    print month, month_dict[month]

month_dict2 = dict(zip(months, month_days))

for month in months:
    print month, month_dict2[month]

given this code, we would like to see a breakdown of the following:
- line 14
- line 17

To do this, we will use the Python debugger, Pdb, which allows us to step through the code line by line.

First, we load the debugger:

```
import pdb; pdb.set_trace()  # load the debugger
```

Then, we define the months and their corresponding number of days:

```
month_str = 'Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec'
months = month_str.split(' ')
month_days = [31, 28, 31, 30, 31, 30, 31, 31, 31, 30, 31]
```

We then create a dictionary `month_dict` where each month is a key and the corresponding number of days is the value:

```
month_dict = {}

for i in range(len(months)):
    month_dict[months[i]] = month_days[i]
```

Next, we print the months:

```
for month in months:  # line 14
    print month
```

Then, we print each month along with its corresponding number of days:

```
for month in months:  # line 17
    print month, month_dict[month]
```

Finally, we create a new dictionary `month_dict2` using the `zip` function:

```
month_dict2 = dict(zip(months, month_days))
```

And then print each month along with its corresponding number of days using the new dictionary:

```
for month in months:
    print month, month_dict2[month]
```

This is the breakdown of the code using the Python debugger, Pdb.
Arrays of arrays (and dicts of dicts)

Python variables are references (already)

- python arrays and dicts are always one-dimensional, but data is usually (at least) two-dimensional.

- How do we build data structures that have multiple dimensions?

  ```python
  hit[1][percid]==86.70
  hit[1][evalue]==3e-112
  ```
Variable dereferencing

To build multi-dimensional (complex) data structures in python, simply put the simple object into the more complex structure (all variables are references in python, no need for reference type):

```python
tn = ['a', 'c', 'g', 't'];  # DNA
pur = ['a', 'g']; pry = ['c', 't']
nt = [pur + pry] == ['a', 'g', 'c', 't']

nt2 = [pur, pry] == [['a', 'g'], ['c', 't']]
# lists do not "flatten"

hit_dict = dict(zip(field_names, line.split('	')))
hit_list.append(hit_dict)
print hit_list
```

```
Variable dereferencing

/seqprg/bin/python
import fileinput
#import pdb; pdb.set_trace()

field_str = 'qseqid sseqid pident length mismatch ... evalue bitscore'
fields = field_str.split(' ')

hits = []  # list of best hits

for line in fileinput.input():
    line = line.strip('
')
    data_dict = dict(zip(fields, line.split('	')))
    hits.append(data_dict)  # hit[n] = {data}

for hit in hits:
    print hit['sseqid'], hit['evalue']
```
Variable dereferencing

```python
franklin: 20 $ python read_hits.py hit_list.data
> /net/t102/users/wrp/biol4230/scripts/read_hits.py()<module>()
-> field_str = 'qseqid sseqid pident length mismatch ... evalue bitscore'
(Pdb) a
-> for line in fileinput.input():
->   line = line.strip('
')
->   data = dict(zip(fields, line.split('	')))
(Pdb) print hits[0]
*** IndexError: list index out of range  # have not appended anything, list empty
-> hits.append(data)  # hits[0] == {}
(Pdb) a
(Pdb) print hits[0]
(..., 'bitscore': '452', 'evalue': '7e-127', ..., 'pident': '100.00', 'length': '218',
  'sseqid': 'sp|GSTM1_HUMAN', 'qseqid': 'sp|GSTM1_HUMAN', ...)
(Pdb) print hits[0]['sseqid']
sp|GSTM1_HUMAN
(Pdb) print hits[0]['sseqid'], hits[0]['evalue']
sp|GSTM1_HUMAN 7e-127
... # after several loops
(Pdb) print hits[1]['sseqid'], hits[1]['evalue']
sp|GSTM1_HUMAN 3e-112
(Pdb) print hits[2]['sseqid'], hits[2]['evalue']
sp|GSTM1_MACFA 3e-110
```

keeping order with dicts[]

When keeping track of a list of hits (or a list of scoring matrices), one often needs two variables
1. a list of the data sets (matrix1, matrix2, matrix3)
2. a list of the results, indexed (keyed) on the dataset names
In the homework, you are asked to report summaries of alignment length and percent identity for multiple searches with multiple scoring matrices. You will need to keep track of the matrix specific data, and the query specific data. One way to do this is with a list of matrices:

```
mat_list=['mat1', 'mat2', 'mat3', etc.]
```
as well as

```
result_dict={mat1:array_of_hits, mat2:array_of_hits, etc.)
```
for the homework, you will need to read a set of files (with the matrix name part of the file name), extract the matrix name, add it to the list of matrix names, and then add the hits to a dict[] that uses the matrix name as the key.

simplify the process of keeping track of your search queries, search results, and matrix names by using a consistent naming scheme. For example, have

```
q200_0.aa, q200_2.aa, ... q200_9.aa, and results
q200_0.bl_blosum62, ... q200_9.bl_blosum62,
q200_0.bl_blosum45, etc.
```
Homework, due Monday, 15 Feb (biol4230/hwk4)


2. For each of 5 scoring matrices from either ssearch (e.g. BL50, BP62, VT160, VT80, VT40) or blastp (BL45, BL62, BL80, PAM70, PAM30), calculate the average alignment length and percent identity from the best match (one average), the second best match (a second average), and the 5th best match (a third average) in a search the 10 shorter sequences against SwissProt. The result should be 5 sets of 3 average alignment lengths, and 5 sets of three average percent identities. (To produce 5 hits from blast, you may need to increase the E()-value threshold.)

3. Repeat the analysis, reporting the average alignment length and percent identity of the top 10 hits from one of the 10 shorter query searches. (To produce 10 hits from blast, you should increase the E()-value threshold.)

4. Are the averages you get in step 2 the same as those in step 3? Why might they be different?

5. Repeat step 2 for the longer query sequences, reporting only the averages for the best hit.