Advanced Topics in Python

Siavash Mirarab

What we will do

- Represent alignment using simple python constructs
- Make simple methods that do interesting stuff on the alignment
- Run external programs on the alignment
- Profile the program to find out about performance

We will learn about

- Object-oriented programming and Classes
- Python built-in function
  - all, any, map, reduce, sorted, zip, etc.
- Generators and lambda
- Calling external programs using Popen
- A bit about timeit
- A bit about Dendropy
- A bit about regex

FASTA Sequences/Alignments

```
>Human
AACGTGATACCTAACGACA-
>Chimp
AA-GTGCTA-CTAACGACAC
>Gorila
GACGTGAAA-CACTCGACAC
```

- Represent the alignment as a dictionary!
- Do stuff using the alignment

Class, Object, Instance, blah blah

- Object-oriented programming:
  - Put cohesive pieces of code together in units called Classes
  - A Class has a set of related data variables and a set of functions that operate on those variables
  - Multiple “instances” of each class can be constructed. These will each have their own data variables.
  - Goal: encapsulate data and functionality together
- Classes are like types (string, int, etc.) and objects are like variables you built from those types(3,6 of type int, “DNA” of type str)

An Alignment Class

```python
class Alignment(object):
    ''' This class represents an alignment '''

    def __init__(self):
        ''' Constructor '''
        pass
```

Let’s put the code above in a module called alignment.py

Using a dict to keep alignments

```python
class Alignment(object):
    ''' This class represents an alignment '''

    def __init__(self):
        ''' Constructor '''
        self.sequences = dict()
```

```python
from alignment import Alignment
a1 = Alignment()
a2 = Alignment()
a1.sequences["Human"]="ACGT"
a2.sequences["Human"]="TGCA"
print a1.sequences["Human"], a2.sequences["Human"]
ACGT TGCA
```

Adding Methods

```
class Alignment(object):
    ''' This class represents an alignment ’’’

    def __init__(self):
        ''' Constructor '''
        self.sequences = dict()

    def add_sequence(self, seq_name, seq):
        ''' Adds a sequence to the alignment.
        Overwrites old sequences with the same name'''
        self.sequences[seq_name] = seq

    def names(self):
        ''' return the list of names'''
        return self.sequences.keys()
    def sequence(self, k):
        ''' return the sequence for a given name'''
        return self.sequences[k]
```

How are methods used?

```
class Alignment(object):
    ''' This class represents an alignment '''


    …

    def add_sequence(self, seq_name, seq):
        ''' Adds a sequence to the alignment.
        Overwrites old sequences with the same name.'''
        self.sequences[seq_name] = seq
```

```
alignment = Alignment()
alignment.add_sequence("Human", "AACGTGATACCTA")
```

More useful functions

```python
class Alignment(object):
    …

    def read_fasta(self, filename):
        name = None
        seq = []
        for line in open(filename,'rU'):
            if line.startswith(">"):
                if name:
                    self.add_sequence(name, ''.join(seq))
                seq = []
                name = line[1:].strip()
            else:
                seq.append(line.strip())
        self.add_sequence(name, ''.join(seq))
```

```python
alignment.read_fasta("test.fasta")
```

Printing an alignment

```
class Alignment(object):
    ''' This class represents an alignment ’’
    …

 def write_fasta(self, dest):
        ''' Write alignment in fasta format to dest.
        sequences will be sorted. dest is a File Object.
        '''
        for name in sorted(self.sequences.keys()):
            dest.write('>%s\n%s\n' %
                       (name,self.sequences[name]))
```

```
import sys
alignment.write_fasta(sys.stdout)
with open("copy_of_test.fasta",'w') as f:
    alignment.write_fasta(f)
```

Checking types

```
class Alignment(object):
    ''' This class represents an alignment ’’
    …

 def write_fasta(self, dest):
        ''' Write alignment in fasta format to dest.
        sequences will be sorted. dest is a File Object. '''
        f = open(dest,'w') if isinstance(dest, str) else dest
        for name in sorted(self.sequences.keys()):
            f.write('>%s\n%s\n' %
                          (name,self.sequences[name]))
        if isinstance(dest, str):
            f.close()
```

```
alignment.write_fasta("another_copy.fasta")
```

Built-in functions: all, any

```
    def is_aligned(self):
        if len(self.sequences) == 0:
            raise ValueError('empty alignment; full or empty?')
        l = len(self.sequences.values()[0])
        return all( [ len(x) == l for
                             x in self.sequences.values() ] )

    def has_gaps(self, key):
        return any ( [x=="-" for x in self.sequences[key] ] )
```

```
print alignment.is_aligned()
print alignment.has_gaps("Human")
alignment.add_sequence("fragment","AAGTG")
print alignment.is_aligned()
print alignment.has_gaps("fragment")
```

Generators using () instead of []

```
    def is_aligned(self):
        if len(self.sequences) == 0:
            raise ValueError('empty alignment; full or empty?')
        l = len(self.sequences.values()[0])
        return all( ( len(x) == l for
                         x in self.sequences.values() ) )

    def has_gaps(self, key):
        return any ( (x=="-" for x in self.sequences[key] ) )
```

```
print alignment.is_aligned()
print alignment.has_gaps("Human")
alignment.add_sequence("fragment","AAGTG")
print alignment.is_aligned()
print alignment.has_gaps("fragment")
```

Built-in functions map, reduce & Generators

```
    def sequences_without_gaps(self):
        ''' Generates sequences with all gaps removed.
        '''
        for seq in self.sequences.values():
            yield seq.replace("-","")

    def max_length(self):
        ''' returns maximum sequence length'''
        return reduce(max,
                      map(len,self.sequences_without_gaps()))
```

```
print alignment.max_length()
```

Built-in functions filter and zip & lambda

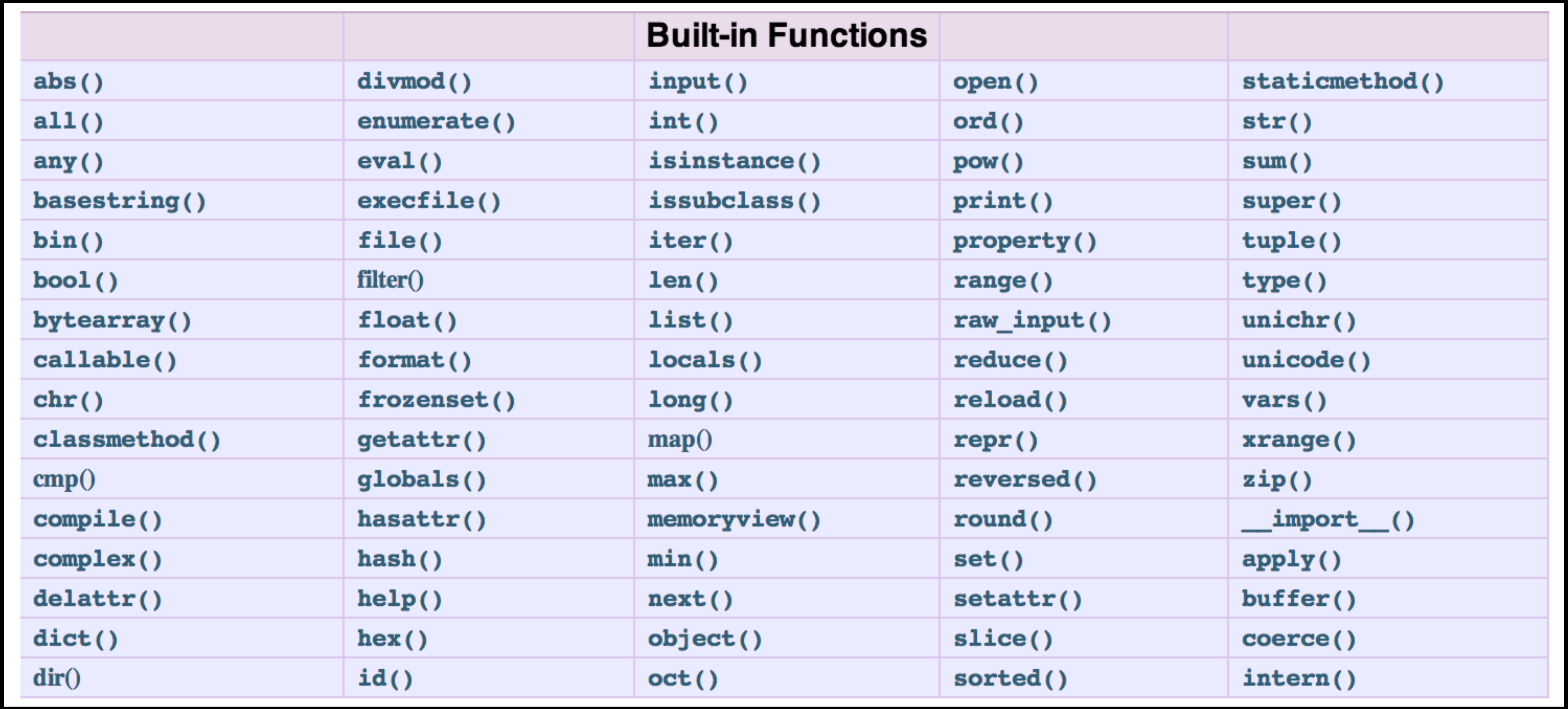
```
def hamming(self,k1,k2):
     s1=self.sequences[k1]
     s2=self.sequences[k2]
     l = float(max(len(s1),len(s2)))
     return sum(
             map( lambda x: 1 if x[0]==x[1] else 0,
                zip(s1,s2) ) ) / l

 def similar_sequences(self, name, thrs):
     return filter( lambda x:
                   self.hamming(name, x) >= thrs,
                   self.sequences.keys() )
```

```
print alignment.hamming("Human","Chimp”)  → 0.8
print alignment.hamming("Human","Gorila") → 0.65
print alignment.similar_sequences("Human", 0.7)
['Chimp', 'Human’]
```

Built-in functions

http://docs.python.org/2/library/functions.html



		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

Running external programs

- You can invoke other programs directly from python
- Many ways, some simple, some not so much
- Will look at the most “kosher” way: popen
- Scenario:
  - Read an alignment
  - Filter out short and distantly-related sequences
  - Run FastTree on it to get an alignment

Filtering function

```
from alignment import Alignment

def filter_alignment(alg, thrs, len_thrs):
    filtered = reduce(set.union,
                      (set(alg.similar_sequences(x, thrs)) - set([x])
                       for x in alg.names()))
    print "%d sequences after filtering by similarity" % len(filtered)

    filtered = filter(lambda x:
                      len(alg.sequence(x).replace("-", "")) > len_thrs,
                      filtered)
    print "%d sequences after filtering by length %d" % (len(filtered), len_thrs)

    filtered_alg = Alignment()
    for x in filtered:
        filtered_alg.add_sequence(x, alg.sequence(x))

    return filtered_alg
```

This is in a new module called filter_n_tree.py

subprocess.Popen

- Use this to call an external program (span new processes)
- Multiple options – simplest:
  - subprocess.call([command,arg1,arg2,…])
  - subprocess.check_call()
  - subprocess.check_output()
- Alternative: p=subprocess.Popen
  - p.communicate()
  - p.wait()

First attempt at running FastTree

```python
import subprocess

in_alg_file = sys.argv[1]
similarity_threshold = int(sys.argv[2])/100.0
length_threshold = int(sys.argv[3])

alg = Alignment()
alg.read_fasta(in_alg_file)
print "%d sequences were found in alignment %s" \
        %(len(alg.names()), in_alg_file)

filtered_alg = filter_alignment \
                (alg,similarity_threshold, length_threshold)

assert filtered_alg.is_aligned(), "input file is not aligned"

filtered_alg_file = "%s.filtered" % in_alg_file
filtered_alg.write_fasta(filtered_alg_file)

subprocess.check_call( ["FastTree",
                        "-out", "%s.tree” % filtered_alg_file,
                        "-nt", "-gtr",
                        filtered_alg_file])
```

Run …

```
33 sequences were found in alignment pythonidae.aln.fasta
23 sequences after filtering by similarity
19 sequences after filtering by length 1000
FastTree Version 2.1.7 SSE3
Alignment: pythonidae.aln.fasta.filtered
Nucleotide distances: Jukes-Cantor Joins: balanced Support: SH-like 1000
Search: Normal +NNI +SPR (2 rounds range 10) +ML-NNI opt-each=1
TopHits: 1.00*sqrtN close=default refresh=0.80
ML Model: Generalized Time-Reversible, CAT approximation with 20 rate categories
Non-unique name 'Antaresia' in the alignment
Traceback (most recent call last):
  File "/Users/smirarab/workspace/IBCC/src/PipeLine.py", line 52, in <module>
    filtered_alg_file])
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/subprocess.py", line 542, in check_call
    raise CalledProcessError(retcode, cmd)
subprocess.CalledProcessError: Command '['FastTree', '-out', 'pythonidae.aln.fasta.filtered.tree', '-nt', '-gtr', 'pythonidae.aln.fasta.filtered']' returned non-zero exit status 1
```

Run …

```
33 sequences were found in alg pythonidae.aln.fasta
23 sequences after filtering by similarity
19 sequences after filtering by length 1000
FastTree Version 2.1.7 SSE3
Alignment: pythonidae.aln.fasta.filtered
Nucleotide distances: Jukes-Cantor Joins: balanced Support: SH-like 1000
Search: Normal +NNI +SPR (2 rounds range 10) +ML-NNI opt-each=1
TopHits: 1.00*sqrtN close=default refresh=0.80
ML Model: Generalized Time-Reversible, CAT approximation with 20 rate categories
…
ML-NNI round 2: LogLk = -13870.760 NNIs 1 max delta 5.26 Time 1.09
    1.08 seconds: ML NNI round 3 of 8, 1 of 17 splits
ML-NNI round 3: LogLk = -13870.715 NNIs 0 max delta 0.00 Time 1.13
Turning off heuristics for final round of ML NNIs (converged)
ML-NNI round 4: LogLk = -13870.075 NNIs 0 max delta 0.00 Time 1.29 (final)
    1.28 seconds: ML Lengths 1 of 17 splits
Optimize all lengths: LogLk = -13870.074 Time 1.33
Total time: 1.56 seconds Unique: 19/19 Bad splits: 0/16
```

Safe names

```python
    def safe_name(self, name):
        return name.replace(" ", "_")

    def write_fasta(self, dest, safe_names=False):
        ''' Write alignment in fasta format to dest.
        sequences will be sorted.'''
        f = open(dest, 'w') if isinstance(dest, str) else dest
        for name in sorted(self.sequences.keys()):
            f.write('>%s\n%s\n' %
                      (self._safe_name(name) if safe_names else name,
                       self.sequences[name]))
        if isinstance(dest, str):
            f.close()

filtered_alg.write_fasta(filtered_alg_file, safe_names = True)
```

Practice

- Add a method to the Alignment class that returns an unaliged alignment object
- Improve filter_n_tree so that it:
  - Reads an input alignment
  - Filters unwanted sequences (you choose criteria)
  - Unaligns the alignment and outputs it
  - Aligns it using an alignment tool (maybe muscle)
  - Builds a tree on the alignment using FastTree

Don’t look at this … Answer

```
def degap(sequence):
    return sequence.replace("-","")

class Alignment(object):
…
    def unalign(self):
        ''' removes gaps from self. '''
        for (k,v) in self.sequences.items():
            self.sequences[k] = degap_seq(v)
```

Don’t look at this … Answer

```
def call_muscle(input_file_name,output_file_name):
    subprocess.check_call(["muscle", "-in", input_file_name,
            "-out", output_file_name])

… [In the main part]
 ''' 1- read input'''
    alg = Alignment()
    alg.read_fasta(in_alg_file)
    print "%d sequences were found in alg %s" %(len(alg.names()), in_alg_file)

    ''' 2- filter alignment and write to a file'''
    filtered_alg = filter_alignment(alg,similarity_threshold, length_threshold)
    assert filtered_alg.is_aligned(), "input file is not aligned"
    filtered_seq_file = "%s.filtered.unaligned" %in_alg_file
    filtered_alg.unalign()
    filtered_alg.write_fasta(filtered_seq_file, safe_names = True)

    ''' 3- call muscle on it to realign'''
    alignment_file_name = "%s.realigned" %filtered_seq_file
    call_muscle(filtered_seq_file, alignment_file_name)

    ''' 4- call fasattree on muscle alignment'''
    call_fasttree(alignment_file_name)
```

Now …

- Quick look at some other topics …

Do Generators help? timeit

```python
import alignment
from timeit import timeit
a=alignment.Alignment()
a.add_sequence("s1", 'AC-GT'*1000)

timeit(lambda: a.has_gaps('s1'), number=10000)
```

```python
 def has_gaps(self, key):
        return any ( (x=="-" for x in self.sequences[key]) )
```

```
0.01221609115600586
```

```python
 def has_gaps(self, key):
        return any ( [x=="-" for x in self.sequences[key]] )
```

```
2.80867600440979
```

regex example

```
def degap(sequence):
    return sequence.replace("-","")

import re
def degap_seq(sequence):
    return re.sub(r"[^a-zA-Z]","",sequence)


matches = [ name for name in alg.names()
            if re.match(".*GTGA[AT]A.*", alg.sequence(name)) ]
print matches

motives = re.findall("AA[CG]G[AT][CG]", alg.sequence('Human'))
print motives
```

Dendropy: reading and manipulating trees

- First need to install dendropy
  - http://pythonhosted.org/DendroPy/downloading.html
  - Useful to have setuptools

Dendropy example

```python
tree_str= subprocess.check_output( ["FastTree",
                                     "-nt","-gtr",
                                     filtered_alg_file] )
print tree_str

from dendropy import Tree
tree = Tree.get_from_string(tree_str, 'Newick')
to_rem = tree.get_edge_set(lambda edge:
                           False if edge.head_node.label is None
                           or float(edge.head_node.label) > 0.99
                           else True)
for edge in to_rem:
    edge.collapse()

print tree.as_newick_string()

tree.write(open("%s.tre.contracted"%filtered_alg_file,'w'), 'Newick')
```

Goal: read the fasttree and contract low support edges

Summary

- Classes are useful constructs in programming
  - The put data and functionality together, and allow you to create your own “types”
- Python has plenty of useful built-in functions (zip, all, any, etc.), language constructs (lambda, generators, etc.) and built-in libraries (re, timeit, sys, etc.)
  - Google your needs. Python likely already has it
- There are many 3rd party libraries; e.g. dendropy.
  - Usually easy to learn and use.