Python in style

```
 3 import sys, re, sh
 4 from collections import OrderedDict
 5 from plumbing.autopaths import AutoPaths
 6 from plumbing.cache import property_cached
 7 from plumbing.slurm import num_processors
 8 from plumbing.graphs import Graph
 9 from matplotlib import pyplot
10 class Checkm(object):
11     short_name='checkm'
12     long_name ='CheckM v0.9.7'
13     executable= 'checkm'
14     url= 'https://github.com/Ecogenomics/CheckM'
15     depend=['hmmer','prodigal',   'pplacer']
16     allp = """
17     /contigs.fasta
18     /stdout.txt
19     /stderr.txt
20     /output/
21     """
22     def __nonzero__ (self):
23         return bool(self.p.stdout)
24     def __init__(self, bin, dr):
25         self.b=b
26         self.dr=dr
27         self.p = AutoPaths(self.dr,self.allp)
28
29
```

```python
# Internal modules #
import sys, re
from collections import OrderedDict

# First party modules #
from plumbing.autopaths import AutoPaths
from plumbing.cache import property_cached
from plumbing.slurm import num_processors
from plumbing.graphs import Graph

# Third party modules #
import sh
from matplotlib import pyplot

###############################################################
class Checkm(object):
    """Use CheckM at to evaluate a bin of contigs.
    Expects version v0.9.7.
    """

    short_name = 'checkm'
    long_name  = 'CheckM v0.9.7'
    executable = 'checkm'
    url        = 'https://github.com/Ecogenomics/CheckM'
    dependencies = ['hmmer', 'prodigal', 'pplacer']

    all_paths = """
    /contigs.fasta
    /stdout.txt
    /stderr.txt
    /output/
    """

    def __nonzero__(self): return bool(self.p.stdout)

    def __init__(self, bin, result_dir):
        # Save attributes #
        self.bin = bin
        # Auto paths #
        self.result_dir = result_dir
        self.p = AutoPaths(self.result_dir, self.all_paths)

```

```python
# Internal modules #
import sys, re
from collections import OrderedDict

# First party modules #
from plumbing.autopaths import AutoPaths
from plumbing.cache import property_cached
from plumbing.slurm import num_processors
from plumbing.graphs import Graph

# Third party modules #
import sh
from matplotlib import pyplot

###############################################################
class Checkm(object):
    """Use CheckM at to evaluate a bin of contigs.
    Expects version v0.9.7.
    """

    short_name = 'checkm'
    long_name  = 'CheckM v0.9.7'
    executable = 'checkm'
    url        = 'https://github.com/Ecogenomics/CheckM'
    dependencies = ['hmmer', 'prodigal', 'pplacer']

    all_paths = """
    /contigs.fasta
    /stdout.txt
    /stderr.txt
    /output/
    """

    def __nonzero__(self): return bool(self.p.stdout)

    def __init__(self, bin, result_dir):
        # Save attributes #
        self.bin = bin
        # Auto paths #
        self.result_dir = result_dir
        self.p = AutoPaths(self.result_dir, self.all_paths)

```

```python
import sys, re, sh
from collections import OrderedDict
from plumbing.autopaths import AutoPaths
from plumbing.cache import property_cached
from plumbing.slurm import num_processors
from plumbing.graphs import Graph
from matplotlib import pyplot
class Checkm(object):
    short_name='checkm'
    long_name ='CheckM v0.9.7'
    executable= 'checkm'
    url= 'https://github.com/Ecogenomics/CheckM'
    depend=['hmmer','prodigal',    'pplacer']
    allp = """
    /contigs.fasta
    /stdout.txt
    /stderr.txt
    /output/
    """
    def __nonzero__ (self):
        return bool(self.p.stdout)
    def __init__(self, bin, dr):
        self.b=b
        self.dr=dr
        self.p = AutoPaths(self.dr,self.allp)


```

```python
#!/usr/bin/env python2
#-*- coding: utf-8 -*-

# PART A

def is_repetition(s1, s2):
    """Determines if a sequence s1 is the repetition of another sequence s2.

        :param s1: Sequence.
        :type s1: str
        :param s2: Motif.
        :type s2: str"""
    # Number of time to expect motif s2 in s1
    times = len(s1) / len(s2)
    if s1 == s2 * times:
        return True
    else:
        return False

# If True is returned by function is_repetition()
s1 = 'ACTACTACTACTACT'
s2 = 'ACT'
if is_repetition(s1, s2):
    print s1, 'is solely composed of motif', s2
else:
    print s1, 'is not solely composed of motif', s2

# PART B

def find_motif(s3):
    """Finds a repeated motif in a sequence.

        :param s3: Sequence.
        :type s3: str"""
    # Go through the string s3
    for i in range(1,len(s3)):
        # Number of time to expect the motif in s3
        times = len(s3)/len(s3[0:i])
        # Check if the string is a repetition of the motif
        if s3 == s3[0:i] * times:
            motif = s3[0:i]
            return motif
    return s3

s3 = 'ACTACTACTACTACT'
print find_motif(s3), 'is the shortest motif found in', s3
```

```python
"""
Solution to Problem 4
Written by Lucas Sinclair
<lucas.sinclair@me.com>

You can test the two functions from the shell like this::
$ python s4_veteran.py
"""

################################################################################
def is_repetition(s1, s2):
    """Determines if a sequence s1 is the repetition of another sequence s2.
       For instance, is_repetition('AAA', 'A') would return True.
       While, is_repetition('ABC', 'A') would return False."""
    return s1 == s2 * (len(s1) / len(s2))

def find_motif(seq):
    """Finds a repeated motif in the sequence *seq*.
    For instance, find_motifs('ABABABAB') would return 'AB'."""
    l = len(seq)
    divisors = [(i,l/i) for i in range(1,l/2) if l % i == 0]
    motifs = [seq[0:i] for i,n in divisors if seq==seq[0:i]*n]
    return motifs[0] if motifs else seq

################################################################################
assert is_repetition('ACTACTACTACTACT', 'ACT')
assert find_motif('ACTACTACTACTACT') == 'ACT'

```

Syntactic style

- Put a space after the = assignment
- Put a space after each , in a list
- Use a capital letter for a class definition
- Group imports by provenance
- Add docstrings to functions and classes
- Use descriptive variable names. Not "data".

Structural style

- Operations that reused and reoccur, should be placed in functions.

- Don't nest your else/if or for loops too much.

- Don't write spaghetti code.

- Use well-thought data structures and methods that act on these structures.

Why spend the time?

1. Readable. By others or by you in six months.

2. Easier to debug. Clear code is less prone to errors

3. What if you want to build on the code and add a feature?

Official Guidelines

PEP 8

Exceptions

```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

Exceptions

```
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str'
and 'int' objects
```

Stack trace

```
 1 def get_second(elems):
 2     return elems[1]
 3
 4 def get_fourth(elems):
 5     return elems[3]
 6
 7 def compute_ratio(elems):
 8     return float(get_second(elems)) / get_fourth(elems)
 9
10 print compute_ratio([2, 7, 8])
```

Stack trace

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-1-c3cafb7b8035> in <module>()
      8         return float(get_second(elems)) / get_fourth(elems)
      9
---> 10 print compute_ratio([2, 7, 8])

<ipython-input-1-c3cafb7b8035> in compute_ratio(elems)
      6
      7 def compute_ratio(elems):
----> 8         return float(get_second(elems)) / get_fourth(elems)
      9
     10 print compute_ratio([2, 7, 8])

<ipython-input-1-c3cafb7b8035> in get_fourth(elems)
      3
      4 def get_fourth(elems):
----> 5         return elems[3]
      6
      7 def compute_ratio(elems):

IndexError: list index out of range
```

Catching exceptions

```
1  try:
2      do_something()
3  except:
4      print "Caught it!"
5
6
7
8
9
10
11
12
13
```

Catching exceptions

```
1   (x,y) = (5,0)
2   try:
3       z = x/y
4   except ZeroDivisionError:
5       print "divide by zero"
6
7
8
9
10
11
12
13
```

Catching exceptions

```
1  try:
2      f = open('myfile.txt')
3      s = f.readline()
4      i = int(s.strip())
5  except IOError as (errno, strerror):
6      print "I/O error({0}): {1}".format(errno, strerror)
7  except ValueError:
8      print "Could not convert data to an integer."
9  except:
10     print "Unexpected error:", sys.exc_info()[0]
11     raise
12
13
14
15
16
```

Raise exceptions

```
1  def get_fourth(elems):
2      if len(elems) < 4:
3          raise Exception("Elems is too short")
4      return elems[3]
5
6
7  ##################
8
9
10 def get_fourth(elems):
11     if len(elems) < 4:
12         raise IndexError("Elems is too short")
13     return elems[3]
```

Warnings

```
1  import warnings
2
3  def get_fourth(elems):
4      if len(elems) < 10:
5          warnings.warn("You might want to check this")
6      return elems[3]
7
8
9
10
11 # /Users/sinclair/test.py: UserWarning: You might want
12 to check this
13
```

Parsing files

```
1  import os
2
3  # Set the path #
4  in_path = "distance.dat"
5
6  # Check that the path is valid #
7  if not os.path.exists(in_path):
8      raise Exception("No file at '%s'." % in_path)
9
10 # The file contains cities and distances tab separated #
11 parsed = (l.strip('\n').split('\t') for l in open(in_path))
12
13
14
15
16
```

Parsing files

```
handle = open('test.tsv')

features = []

for line in handle:
    if line.startswith("#"): continue
    start, end, score = line.split('\t')
    if end < start: raise Exception("Negative feature")
    if score < 0: raise Exception("Negative score")
    features.append((start, end, score))
```

Parsing files

```
1
2 import pandas
3 genes = pandas.read_csv('csvfile.txt',
4                         index_col=False,
5                         header=0)
6
7
8
9
10
11
12
13
14
15
16
```

15 minutes break