You collaboratively work w/ a friend on code, friend stays up later, than you, left a note, “Great progress! Put all changes in our bitbucket repo.” What command do you need to get friend’s update?
Gitpull

You made changes to code, didn’t do git operations. Friend is awake and wants to share changes. Which order of git?
Add,commit,push

You were part of open source project from beginning. New leader emerged that you don’t get along with. Leader convinces group to kick you out. Project still remains open source. Which will not be able to do?
Get modifications incorporated into project’s bit-bucket respitory

BSD license – class of extremely simple and liberal licenses for computer software, it doesn’t put restrictions on ability to distrubte more-restricted version. GPL would require that any modified version distrubted also has the GPL and you would not be able to distribute a version that customers can’t modify/redistribute.

GPL license – guarantees free software cannot become non-free, prevents others from distributing your code under anything other than GPL. They can’t charge for it because anyone else who got it could redistribute it for free

Code has been executed (“”” is delimiter for strings on more than 1 line):
S=”””<entry>
<id>tag:search.twitter.com,2005,1142881099</id><published>2009-01-23T20:04:53Z</published></entry>”””
print s.find(“>”)
6
print len(s.split(‘:’))
4
print len(s.split(‘>’)[2].split(‘:’))
2

Define function f that takes list of string, returns list containing 1st letter of every word with letter z. Make it pass test. Function must use list comprehension/map/filter.
Test.testEqual(f([‘Amazing’, ‘corny’, ‘zany’]), [‘A’, ‘z’])
Def f(L):
Return [item[0] for item in L if ‘z’ in item]

Define function g that takes list of strings, returns list of them, sorted in alphabetic order by last character. Make it pass test.
Test.testEqual (g([‘Amazing’, ‘corny’, ‘zanier’]), [‘Amazing’, ‘zanier’, ‘corny’])
Test.testEqual(g([‘good’, ‘good on ya’, ‘good on you’], [‘good on ya, ‘good’, ‘good on you’])
Def g(L):
Return sorted(L, key=lambda x: x[-1])

Testing functions
Return value test – return correct value
- test.testEqual(square(3), 9)
Side effect tests – modify contents of some mutable object, list/dictionary, tests if function makes right changes to mutable object (for multiple lines of code) – set mutable object to some value, run function, check if object has expected value

```
#find word with most Z
def sleepiest(L):
    max_word = ''
    max_ct = 0
    for word in L:
        for letter in word:
            new_max = 0
            if letter == 'z':
                new_max +=1
```

Fill in 2nd line using string interpolation to pass tests.
Test.testEqual(interp(5, “sir”), “That is 5 in a row, sir. Congratulations!”)
Test.testEqual(interp(6, “your higness”), “That is 6 in a row, your highness. Congratulations!”)
Def interp(x,y):
Mystr = “That is %d in a row, %s. Congratulations!” % (x, y)

Fill in parameter list for function h so tests pass
Test.testEqual(h(1,2), [1, 2, 4])
Test.testEqual(h(1, z=5), [1, 3, 5])
Def h(x, y=3, z=4):
Return [x, y, z]

You defined function enum; takes list as input and supposed to produce list of tuples that number original items: 1st is 1, 2nd is 2, etc. Code isn’t working right. Code generates output “Failed test 8: items in expected do not match.” Rewrite definition of enum so it passes test.
Test.testEqual(enum([“a”, “b”, “c”]), [(1, “a”), (2, “b”), (3, “c”)])
Wrong code:

```
def enum(L):
  res = []
  for item in L:
    n = 1 (put above for loop to get right)
    res.append((n, item))
    n=n+1
  return res
```

For following:

```
Def count_guesses(next_letter, guesses):
    """guesses is a list of guesses to be made, in order. Returns number of guesses that are made in order to guess next_letter, or None if not among guesses"""
    try:
        return guesses.index(next_letter)+1
    except:
        print "%s not among guesses" % next_letter
```

To write test cases for count_guesses, make:
Return value tests
Write test that checks that right thing happens when next_letter is among guesses:
Test.testEqual(count_guesses(“a”, [“b”, “c”, “e”, “a”, “f”]), 4)
Write test that checks right thing happens when next_letter isn’t among guesses:
Test.testEqual(count_guesses(“a”, []), None)

After sequence of capital letters in a text, makes more sense for Shannon guesser to guess another capital. You can take list of guesses and rearrange to put all capitals 1st, keeping same order among capitals that they had in original guess list. Define function caps_first. Pass test.
Test.testEqual(caps_first([“A”, “i”, “K”, “e”]), [“A”, “K”, “i”, “e”])

```
Def caps_first(L):
    """move all capital letters in L to front of list, maintaining the order"""
    L_caps = [c for c in L if c in caps]
    L_small = [c for c in L if c not in caps}
    Return L_caps + L_small
```

Rewrite find_redundancy() to make it a method of Post class

```
Class Post():
    """object representing one post"""
    def __init__(self, post_dict):
        self.message = post_dict['message']
def find_redundancy(self):
    min_guesses, actual_guesses = game(self.message)
    self.redundancy = actual_guesses/float(min_guesses)
```

```
dic = {"Nate" : 100, "Lefty" 90, "Simon" :0}
return sorted(dic.keys(), key = lambda x: dic[x], reverse = True)
#top name list by score
return sorted(dic, key = lambda x: dic[x], reverse = True)
```

Code below replaces line 7; makes list of instances instead of list of dictionaries; understand what it does
Posts=[post(p) for p in feed[‘data’] if ‘message’ in p]
Rewrite so find_redundancy is invoked as a method instead of function. Rewrite to sort instances instead of sorting dictionaries. Rewrite to print same things they printed before but taking into account that posts are instances instead of dictionaries now

```
For p in posts:
    Find_redundancy(p)
Surprising_posts=sorted(posts,key=lambda p: p['redundancy'], reverse=True)
For p in surprising_posts[:10]:
    Print "%.2f: "s" % (p['redundancy'], p['message'])
For p in surprising_posts[-10:]:
    Print "%,2f: %s" % (p['redundancy'], p['message'])
Surprising_posts = sorted(posts,key=lambda p:p.redundancy, reverse=True)
For p in surprising_posts[:10]:
    Print "%.2f: % s" % (p.redundancy, p.message)
For p in surprising_posts[-10:]:
    Print "%.2f: %s" % (p.redundancy, p.message)
```

What prints?
S=”<published>2009-01-23T20:04:53z</published>”
Print len(s.split(‘T’)[0].split(‘2’))
3

Following has been executed
L=[“First”, “Second”, “Third”]
What prints?

```
For x in L:
    Y=L[0]
    Print y
```

First
First
First

```
For x in L:
    Y=L[0]
Print y
```

First

```
For x in L:
  Y=x in L
  Print y
```

True
True
True

What prints?

```
L=[]
L.append('a')
L.append('b')
L.append('c')
L[1]=0
Print L
```

['a', 0, 'c']

Making a list that sorts by most hungry animal
Animals = sorted(animals, key =lambda x: x.hunger)
Generating a list of only the pets that are bored
Bored = [x for x in animlas if x.boredom > x,boredom_threshold
Generating a list of (boredom, hunger) tuples for each pet
Tups = [(“x.bordeom, x.hunger) for x in animals]
Write tamo to csv that gives each animal name, hunger, boredom

```
F = open("tamo.csv", "w")
f.write("Name, Hunger, Boredom\n")
for x in animals:
    f.write("{},{},{}\n".format('x.name, x.hunger, x.boredom')
f.close()
```

To test method of Pet class/ **return side effect**
write a test for the the teach method of the Pet class

```
p1=Pet()
pl.teach("yay")
test.testEqual(p1.sounds,["Mrrp", "Yay"])
```

define a sub class called Cyborg. Same as a regular pet but does not get hunfry until is bored

```
class Cyborg(Pet):
    def clock_tick(self):
        self.boredom +=1
        if self.boredom > self.boredom_threshold:
            self.hunger +=1
```

use list comp to return first let of words with z in

```
def f(L):
    return [x[0] for x in L if "z" in x]
```

```
f = open("results.csv", "w")
f.write("Song, Artist, Average Rank\n")
for x in sorted_list:
    f.write("{}, {}, {} {}\n".format(songs[x].name, songs[x].artist, songs[x].get_avg_ranking))
f.close()
```

What prints?

```
D=()
D[1] = 'a'
D[2] = 'b'
D['c'] = 3
D['c'] = d['c'] + 1
Print d['c']
```

4

Print ‘a’ in d.values()
True

Write code that accomplishes what last 3 lines do

```
T=(20, 30, 40)
X=t[0]
Y=t[1]
Z=t[2]
```

X, y, z = t

What prints?

```
Def g(x, y):
    Z = y + x
    Return y
Y = 10
Z = g(5,y)
Print z
```

10

What prints?

```
X=-1
Y=-2
Z=-3
Def h(x, y=2, z=3):
    Print x, y, z
H(1)
```

1, 2, 3

```
L=[{'a':1, 'b':2, 'd':11}, {'a': 4, 'b':5, 'e': 11}, {'a':7, 'b':8, 'f':11}]
Write code to print each value with key b
For d in L:
    Print d['b']
```

Write code that makes 1 dictionary with 1 key for each key

```
Dx = {}
For d in L:
    For k in d.keys():
        If k in dx:
            Dx[k] = dx[k] + 1
        Else:
            Dx[k] =1
```

Write code that asks user to input numbers until sum is 21+

```
Sum = 0
While sum < 21:
    X = int(raw_input("Enter a number"))
    Sum = sum + x
Print sum
```

Define function that takes list as input and returns list that has all duplicates removed keeping only first instance of each item.

```
Def deduplicate(lst):
    Acc_lust = []
    For x in lst:
        If x not in acc_lst:
            Acc_lst.append(x)
    Return acc_lst
```

Writing classes:

```
Class motorcycle():
    Def __init__ (self, color= 'red', mpg=40):
        Self.color = color
        Self.mpg = mpg
        Self.dist_from_origin = 0
    Def __str__ (self):
        S= "This motorcycle is %s and goes %d mpg" % (self.color, self.mpg)
        Return s
New_ride = motorcycle()
Green_bike = motorcycle(color='green', mpg=50)
Green_bike.move(20)
Print green_bike.dist_from_origin
```

Print new_ride → prints string method
Test_bike = motorcycle() → tests default
Test_2 = motorcycle(“blue”) → tests not default
Test.testEqual(test_bike.color, ‘red’) → what you are testing/what it is supposed to be
Test.testEqual(test_2.color, ‘blue’)

Return username based of follower count

```
Def pop_tweets(s):
    L = json.loads(s)
    Sorted_ds = sorted(L, key = lambda x:x['user']['follower_count'], rev = True)
    Return [d, ['user']['screen_name'] for d in
```

Which is reason to use version control system like github?
You would like to be able to see/revert to past version of any file in project ////You want to collaborate with others/work in parallel and merge changes together ///You want to distribute code in public repository that others can fork/comment on/// allow some to make changes but no all
Which behaviors are unfavorable by most people who participate in open source projects?
Making fork of existing project, improving code majorly, answering questions about project on StackOverFlow by pointing people to your repository with improved code//// inviting other project memebers to join your projec
What prints?
```
Def interp(L, i):
   Templ = "%s is at idx %d in a list with % d items"
   Vals = (L[i], i, len(L))
   Return templ % vals
Print interp(['you', 'are', 'a', 'genius'], 3)
```
Genius is at idx 3 in a list with 4 items
What prints?
```
X=-1
Y=-2
Z=-3
Def h(x, y=2, z=3):
   Print x, y, z
H(1, z=4)
```
1 2 4
What prints?
```
L1 = ['a', 'b', 'c']
L2 = [1, 2, 3]
Zipped = zip(L1, L2)
Print len(zipped)
Print zipped[1]
Print zipped[1] [0]
```
3
('b', 2)
b
Define function join_strings
```
Def join_strings(L, sep):
   New_str = ""
   For x in L:
     New_str = new_str + x + sep
  If sep== "":
    Return new_str
  Else:
    Return new_str[:-1]
```
Write code to sort L in reverse alphabetical order
```
L = ["Clear 40", "All 99", "Beautiful 20", "Delightful 80"]
```
Sort(L, reverse=True)
Use zip to take lists and turn into tuples with each 1st item as 1st item
```
L1=[1,2,34]
L2=[4,3,2,3]
L3=[0,5,0,5]
```
Tups = zip(L1, L2, L3)
Sort – method that arranges strings in alphabetical order, ints small-large
```
L1=[7, -2, 3]
L2=['Beta', 'Zeta', 'Alpha']
L1.sort()
```
Print L1 → [-2, 3, 7]
Print L1.sort() → prints None
Reverse – makes it go in reverse alphabetical or int large-small
```
L=[0, -2, 1, 10]
Print sorted(L, reverse=True) → [10, 1, 0, -2]
```
Sorts by alpha(keys) returns number

```
def sorted_by_keys(d):
        dict=sorted(d.keys())
       new_lst=[]
       for x in dict:
              new_lst.append(d[x])
return new_lst

d2={'alpha':10,'b':30,'d':20,'c':10}

print sorted_by_keys(d2)
```

Define function that takes dictionary as input and returns list of keys, sorted based on values with keys. Pass test.
```
D1 = {a': 10, 'b':30, 'c':20}
Test.testEqual(sorted_keys(d1), ['b', 'c', 'a'])
```
Def sorted_keys(d):
** Return sorted(d.keys(), key=lambda x:d[x], reverse=True)**
Define function that takes input a list of strings and returns output 3 longest strings, longest to shortest. Pass test.
```
Some_strings =['a', 'abcd', 'ab', 'abc', 'abcde']
Test.testEqual(longest_strings(some_strings), ['abcde', 'abcd', 'abc'])
```
Def longest_strings(L):
** Return sorted(L, key=len, reverse=True)[:3]**
What prints?
```
X=6
Print (lambda x: x-2)(5)
```
3
Something isn't working with word_counts. Takes string as input, produces a dictionary containing words in string as keys. Output is shown. Rewrite definition of word_counts to pass test.
```
--Failed test 21:
expected: {'a': 3, 'Panama': 1, 'plan': 1, 'canal': 1, 'man': 1}
got: {'a':2, 'panama':0, 'plan': 0, 'canal': 0, 'man':0}
def word_counts(s):
  d={}
  words = s.split()
  for w in words:
    try:
      d[w] = d[w]+1
    except:
      ~~d[w]=0~~
      d[w]=1
 return d
```
Write function that takes post as input and returns list of ID's of likers.
```
Def likers(post):
   Res=[]
   For d in post['likes']['data']:
       Res.append(d['id'])
   Return res
```
Write same with comp/map/filter.
```
Def likers(post):
   Return map(lambda d: d['id'], post['likes']['data'])
```
Or
```
Def likers(post):
   Return [d['id'] for d in post['likes']['data']]
```
Write code to create instances of Post class that have no likes/comments. 1st instance have phrase "I love python." 2nd instance has message "Gadzooks, I am a programmer now!"
```
P1=Post({'message': "I love python.})
P2=Post({'message': "Gadzooks, I am a programmer now!"})
```
Define method entropy from post class. Provides a poor estimate of actual entropy. Computes percentage of letters in post's message text are unusual.
```
Def entropy(self):
   Count=0
   For let in self.message:
      If let in ['q', 'x', 'j', 'z']:
          Count = count + 1
   Return count / float(len(self.message))
```
Write test case that checks ^^
```
P=Post({'message': 'qxajj'})
Test.testEqual(p.entropy(), 0.8)
```
Use zip/map/comp to make list of max value.
```
L1=[1,2,3,4]
L2=[4,3,2,3]
L3=[0,5,0,5]
```
Maxs=[max(x) for x in zip(L1, L2, L3)]
```
Def enthusiast_count(self):
    Ls = self.likers() #previous defined
    Cs = self.commenters() #prev define
   Return len([id for id in ls if id in cs]
#count of people who were in likes and comments
```

Use reduce
```
L1=[1,2,3,4]
L2=[4,3,2,3]
L3=[0,5,0,5]
Def sumSquares(L):
   Return reduce(lambda accum, y:accum + y*y, L, 0)
```
Use map and sum
```
L1=[1,2,3,4]
L2=[4,3,2,3]
L3=[0,5,0,5]
Def sumSquares(L):
   NewL=map((lambda x: x*x), L)
   Return sum(newL)
```
Write function that takes list of numbers and returns squares of numbers.
```
Def sumSquares(L):
   newL = []
   for x in L:
     newL.append(x*x)
   return sum(newL)
nums = [3,2,2,-1,1]
```
With map/filter
```
Def longlengths(strings):
   List2=filter((lambda x:len(x)>4, strings)
   Return map((lambda x: len(x)), list2)
```
Combine lengths with long words to make function that returns lengths of strings with at least 4 characters.
```
Def longlengths(strings):
   Return [lex(x) for x in strings if len(x)>4]
```
Use list comprehension
```
Def longwords(strings):
   Return [x for x in strings if len(x)>4]
```
Use filter
```
Def longwords(strings):
   Return filter((lambda x: len(x) > 4), strings)
```
Use manual accumulation
```
Def longwords(strings):
   Long_list =[]
   For x in strings:
      If len(x) > 4:
         Long_list.append(x)
   Return long_list
```
Write code to produce list of only positive things
```
Things = [3,5,-4,7]
```
Print filter((lambda x: x>0), things)
Write code that takes lengths as list of strings, returns list of numbers of strings in input
```
Def lengths(strings):
    Return [len(x) for x in strings]
```
Use map
```
Def lengths(strings):
   New_strings = map((lambda x: len(x)), strings)
   Return new_strings
```
Use manual accumulation
```
Def length(strings):
   Z=[]
   For x in strings:
      z.append(len(x))
   return z
```
Use map instead of manual accumulation
```
Things = [3,5,-4,7]
New = map((lambda x: x+1), things)
Print new
```
Map – takes 2 arguments, function and sequence, function is mapper that transforms items – automatically applied to each item in sequence so you don't need to initialize accumulation iteration
```
List = [2,5,9]
List 2 = map((lambda x: 2*x), List)
Print list 2 → [4,10,18]
```
What will print?
```
def rotate_char(c, postion =1, alphabet = "abcd…."):
     try:
       idx = alphabet.index(c)
       rotated_idx = (idx + position)%len(alphabet)
       return alphabet[rotated_idx]
   except:
       print "%s in not in %s" % (c, alphabet)
       return c
print rotate('I')  J
print rotate('Z')  A
print rotate('?')  ? in not in alpha / ?
print rotate('H', position = -1)  H
```
Define func encrypy, string as input and returns new string?
```
Def encrypt(a, position =1, alphabet = "abcd…."):
    L = map(lambda c: rotate_char(c.positons, alpha, s)
    Return join_string(c, "")
```

Reduce – takes list, produces combined value from all elements – 1st parameter is function that combines result-so-far with next element of list, 2nd is list to be aggregated, option 3rd is initial value for accumulator variable if not provided, first element of list is used as initial value
```
Nums=[3,4,6,-7,0,1]
Print reduce(lambda x,y:x+1, nums, 0)
counts
Print reduce(lamba x,y: x+y, nums) adds
Def greater(x,y):
   If x>y:
     Return x
   Else:
     Return y
Print reduce(greater,nums)
```
Zip – doing something with pairs of lists with all of 1st items of list, something with 2nd, etc.
```
L1=[3,4,5]
L2=[1,2,3]
L3=zip(L1, L2)
Print L3 → [(3,1), (4,2), (5,3)]
```
Zip makes multiple lists and turns them into list of tuples, once we have tuples we can iterate through and perform operations
```
L4=[]
For(x1, x2) in L3:
  L4.append(x1 + x2)
Print L4 → [4,6,8]
```
List comprehension:
```
L5=[(x1 + x2] for (x1, x2) in zip(L1, L2)]
```
Print L5 → [4,6,8]
string interpolation
```
def interp(x,y):
   mystr="This is fun %s, for %d days" % (x,y)
   return mystr
test.testEqual(interp("Alex", 5), "This is fun Alex, for 5 days")
```
Sorted – function, does not change original list, returns value of new list
```
L2=['Beta', 'Zeta', 'Alpha']
L3=sorted(L2)
```
Print L3 → ['Alpha', 'Beta', 'Zeta']
Print L2 → ['Beta, 'Zeta','Alpha']
*unchanged
```
airport_list=["LGA, "JFK", "DTW", "ISP"]
outfile=open("airport_tems.csv", "w")
outfile.write("airport_name, status_reason, current_temp, recent_update\n")
for airport in airport_list:
   last_var=safe_airport_data
   outfile.write("%s, %s, %s, %s\n")
outfile.close()
```
Fork – personal copy of another user's repository that lives on your account, allows you to freely make changes without affecting original, allows you to submit pull request to originals author to update with changes
Clone – copy of repository that lives on comp, can edit files, use git to keep track of changes
Commit – revision, change to a file, when you save with Git it creates a new ID
Pull request – proposed changes to repository submitted by user and accepted/rejected by collaborators
Issue – suggest improvements, can be created by anyone
Branch – parallel version of repository, contained within repository, does not affect primary/master branch, allows you to work freely without disrupting live version
Shannon game – guesses letter and then moves onto next letter when you get first letter right
Filter – goes through list and keeps only items that meet certain criteria, takes 2 arguments- function and sequence and don't need to initialize an accumulator/iterate with for loop
```
Nums=[5,6,7,8,9]
New_list=filter((lambda x: x%2==1), nums)
Print new_list
```
List comprehension - syntax<expression>for <item> in <sequence>
```
Nums = [2,5,9]
```