05/12/18 12:01:14 /Users/frankyoung/Documents/Python3/18 May/Nobody_gets_ready.py

```
""" formatting"""
from math import pi
import datetime
# print('{:07,.2f}'.format(pi))
lst = ['python', 'java']
dct = {'name': 'frank', 'age': 27}
dt = datetime.date.today()
str_formatting = 'the pi is {0:07,.2f}, my name is {name}, i am {age} years old,i am learing {1} and {2} ,today is {3:%B %Y %d}'.format(pi, *lst, dt, **dct)
# try:{name.upper()} ,Not Supported./AttributeError: 'str' object has no attribute 'upper()'
# "cant not use method call in ''.format(),only subscription (indexing by number or by unquoted (!) name), and attribute access is supported.but f'{name.upper()}' works.
# name = 'frank'
# print(f'{name.upper()}')
# print('{.upper()}'.format(name))
print(str_formatting)

import csv
import contextlib
# file doesnt exist,the point is DictReader.DictWriter is much easier to use,and you can modify the the new csv's fieldnames order if needed.
with contextlib.suppress(Exception):
    with open('Customer_Satisfaction.csv') as rf:
        reader = csv.DictReader(rf)
        print(reader.fieldnames)  # ['Year', 'Category', 'Satisfaction Rating']
        with open('Customer_Satisfaction_copy.csv', 'w') as wf:
            fieldnames = ['Category', 'Year']
            writer = csv.DictWriter(wf, fieldnames=fieldnames, delimiter='\t')
            writer.writeheader()
            for line in reader:
                del line['Satisfaction Rating']
                writer.writerow(line)

# Sort file into year_month folder
import os
```

```
33  import datetime
34  from contextlib import suppress
35  def year_month_folder(path):
36      os.chdir(path)
37      for file in os.listdir():
38          if os.path.isfile(file):
39
40              mtime = os.stat(file).st_mtime
41              mtime = datetime.date.fromtimestamp(mtime)
42              folder_name = f'{mtime:%y %B}'
43              with suppress(FileExistsError):
44                  os.mkdir(folder_name)
45              name_path = os.path.join(folder_name, file)
46              os.rename(file, name_path)
47
48
49
50  """object sorting"""
51  from operator import itemgetter, attrgetter, methodcaller
52
53  student_tuples = [('john', 'A', 15), ('jane', 'B', 12), ('dave',
    'B', 10)]
54  print(sorted(student_tuples, key=itemgetter(2)))
55  print(sorted(student_tuples, key=itemgetter(1, 2)))
56
57
58  class Student:
59      def __init__(self, name, grade, age):
60          self.name = name
61          self.grade = grade
62          self.age = age
63
64      def __repr__(self):
65          return repr((self.name, self.grade, self.age))
66
67
68  student_objects = [Student('john', 'A', 15), Student('jane',
    'B', 12), Student('dave', 'B', 10), ]
69
70  print(sorted(student_objects, key=attrgetter('age')))
71  print(sorted(student_objects, key=attrgetter('grade', 'age')))
72
73  messages = ['critical!!!', 'hurry!', 'standby', 'immediate!!']
74  print(sorted(messages, key=methodcaller('count', '!')))
75
```

```
 76
 77  """contextmanger"""
 78  import os
 79  import glob
 80  import contextlib
 81
 82
 83  @contextlib.contextmanager
 84  def find_py(path):
 85      ori_path = os.getcwd()
 86      try:
 87          os.chdir(path)
 88          result = glob.glob('*.py')
 89          yield result
 90      finally:
 91          os.chdir(ori_path)
 92
 93
 94  # with find_py('/Users/frankyoung/Documents/Python3/18 March')
     as f:
 95  #     for pyfile in f:
 96  #         print(pyfile)
 97  class Find_Py:
 98      def __init__(self, path):
 99          self.path = path
100          self.ori_path = os.getcwd()
101
102      def __enter__(self):
103          os.chdir(self.path)
104          result = glob.glob('*.py')
105          return result
106
107      def __exit__(self, exc_type, exc_val, traceback):
108          os.chdir(self.ori_path)
109
110
111  # with Find_Py('/Users/frankyoung/Documents/Python3/18 March')
     as f:
112  #     for pyf in f:
113  #         print(pyf)
114  # print(os.getcwd())
115
116
117  @contextlib.contextmanager
```

```
118 # Nick Coghlan
119 def suppress(*exceptions):
120     try:
121         yield
122     except exceptions:
123         pass
124
125
126 # decorator,scope,closure
127 # the LEGB rule , for accessing(say,print) and modifying(append)
    mutable var only.But not reasigning after referenced to a
    nonlocal(enclosing,Global) Var.UnboundLocalError: local variable
    referenced to  before assignment.
128 # <Thomas Ballinger - Finding closure with closures - PyCon
    2016>"It turns out that Python analyzes function source code,
    even compiles it, when a function is defined. During this
    process it determines the scope of each variable. This
    determines the process that will be used to find the value of
    each variables, but does not actually look up this value yet."
129 # "the scope of a var is determined when function is defined ,
    the value of a var is determined when the function is called".
130
131 # https://nedbatchelder.com/text/names.html
132 'so when the function returns, those names go away. But if the
    values they refer to are still referenced by other names, the
    values live on.--nedbatchelde'
133 # 'so when the function returns, those names go away.'i like to
    see it as when a function return ,the local var is not
    accessable from the global scope, i don't know if they disappear
    or not. maybe like ned said, 'if the values they refer to are
    still referenced by other names, the values live on(for thoese
    other names)' like what we have seen from a closure.
134
135 a = 1
136
137 def p():
138     print(a)
139
140 a = 2
141
142 def w():
143     print(a)
144
145 p()  # 2
```

```
146 w()  # 2
147
148 # http://docs.python-guide.org/en/latest/writing/gotchas/
149 # Late Binding Closures
150 funcs = [lambda x: x * i for i in range(3)]  # by the way , i
    here is a local var.so i doesnt ni=ot exist in the global
    scope.if try to print i , 'NameError: name 'i' is not defined'
151 for func in funcs:
152     print(func(2))
153 # 4,4,4
154 # you will get 2*2=4,2*2=4,2*2=4,because the three 'i's  are in
    the same scope,when the functions is called , i=2.so you get
    4,4,4.
155 # Solution:use keyword args.'Python's default arguments are
    evaluated once when the function is defined, not each time the
    function is called '
156 funcs = [lambda x, i=i: x * i for i in range(3)]
157 for func in funcs:
158     print(func(2))
159 # 0,2,4
160 # or you can use generator expression, without keyword
    args.because generator look up the value as it goes.
161 funcs = (lambda x: x * i for i in range(3))
162 for func in funcs:
163     print(func(2))
164 # 0,2,4
165 #
166 # function attribute, functions can have attributes.
167
168 # make a counter decorator using function attr.
169 from functools import wraps
170
171
172 def counter(my_func):
173     @wraps(my_func)
174     def inner(*args, **kwargs):
175         inner.count += 1
176         return my_func(*args, **kwargs)
177     inner.count = 0
178     return inner
179
180
181 @counter
182 def i_tell_you_what():
```

```
183     return 'i tell you what'
184
185
186 i_tell_you_what()
187 i_tell_you_what()
188 print(f'{i_tell_you_what.__name__} run {i_tell_you_what.count}
    times')
189
190 # make a cache with a default return dictionary as an arg
191
192
193 def cache_with_default(dct=None):
194     if dct is None:
195         dct = {}
196
197     def cache(my_func):
198         @wraps(my_func)
199         def wrapper(*args):
200             if args in dct:
201                 return dct[args]
202             result = my_func(*args)
203             dct[args] = result
204             return result
205         return wrapper
206     return cache
207
208
209 @cache_with_default({(1,): 100})  # be careful, pass 1 as a
    tuple (1,) or it wont work.becaues the args will be (1,)
210 def times_two(x):
211     return x + x
212
213
214 print(times_two(1))  # get 100 inside of 2 , because it was
    looked up in the dct
215
216
217 # Brett Slatkin - How to Be More Effective with Functions -
    PyCon 2015 - YouTube
218 # keword_only_args:forced to be clear.
219 # dont pass infinity generator into *args, like
    itertools.count().it will try to tuple(count()),and that will
    crash.
220 # To avoid this possibility entirely, you should use keyword-
    only arguments when you want to extend functions that accept
```

```
    *args
221 # http://www.informit.com/articles/article.aspx?p=2314818
222 # *this is a kwargs_only function, so :'anything after a "*" or
    "*args"  is FORCED to be clear (keyword)'
223 def kwargs_only(*args, a=1):
224     print(a)
225     print(args)
226
227
228 kwargs_only(2, 2, 2, 2, a=4)
229
230 def bobby(*, propane=True, charcoal=False):
231     if propane:
232         print('there you go')
233     else:
234         print('the hell you say')
235
236
237 try:
238     bobby(True, False)
239 except Exception as e:
240     print(e)  # bobby() takes 0 positional arguments but 2 were
    given
241 bobby(propane=True, charcoal=False)  # there you go
242 #same idea , if you can ,put return value into a namedtuple ,to
    be clear.--> 'Raymond Hettinger's Transforming Code into
    Beautiful, Idiomatic Python'
243
244
245
246 # what is generator?
247 # iter(foo) is iter(foo)
248 # base on the talk-->
249 # if iter(foo) is iter(foo):
250 #     now,then =itertools.tee(foo,2)
251 # customize iteration : "Brett Slatkin - How to Be More
    Effective with Functions - PyCon 2015 - YouTube" + "Loop like a
    native_ while, for, iterators, generators" ---->by using class
    __iter__ method:
252 # compare this info_get function and Info_Gen Class:
253 # difference is ever time you call the 'for' , __iter__method on
    class, it return a new iterator over a container.so it is like
    itertools.tee
254 def info_gen(path):
```

```
255     with open(path) as f:
256         reader = csv.DictReader(f)
257         for line in reader:
258             del line['Year']
259             yield line
260 
261 
262 class Info_Gen:
263     def __init__(self, path):
264         self.path = path
265         print(self.path)
266 
267     def __iter__(self):
268         return info_gen(self.path)  # important must be returned ! to a genator func ,i believe it is the scope reason, if not returned, values are not catched.
269 
270 
271 # http://nvie.com/posts/iterators-vs-generators/    : iter(iterable)-->iteration
272 # how to detect a generator
273 # how does generator function(yield) run?
274 lst = [1, 2, 3]
275 a = iter(lst)
276 b = iter(lst)
277 print(a is b)  # False
278 print(a == b)  # false
279 print([a] == [b])  # false
280 print(list(a) == list(b))  # true
281 c = iter(a)
282 print(a is c)  # True
283 print(a == c)  # True
284 print([a] == [c])  # false
285 
286 lst = [1, 2, 3]
287 a = iter(lst)
288 c = iter(a)
289 print(list(a) == list(c))  # False  ([1,2,3] ==[])
290 # print(list(a))
291 # print(list(c))
292 # so if iter(foo) is iter(foo), foo is a generator; if iter(foo) is not iter(foo), foo is a container. 'iter over a iterator returns itself.'
293 # that is 'is' how about '=', how about [a],[b],[c] and list(a),list(d),list(c),see above.(this how i see it)basiclly,
```

```
    python doesn't look inside a iterator see what value it
    carry(and it shouldn't),so if 2 iterator object with different
    address, it is not equal(you can see as not 'is' ,so not '=').
    same thing with [],but list() is different. list() will really
    loop up the value.
294
295
296 # how yield works
297
298 import contextlib
299
300
301 def HYW():
302     print('hello')
303     yield
304     print('world')
305
306
307 a = HYW()  # Nothing happend, ! hello was not printed.
308 next(a)  # ---> now hello was printed. so when you call next,
    generator will run till it hits a yield
309
310 with contextlib.suppress(StopIteration):
311     next(a)  # -----> world was printed, and then it hits the
    StopIteration
312
313
314 #  Ned Batchelder - Facts and Myths about Python names and
    values - PyCon 2015
315
316
317
318
319 a = [1, 2, 3]
320 b = a
321 a += [4, 5]  # what happened here unline is "a.extend([4,5]) and
    a =a "
322 print(b)  # -->[1, 2, 3, 4, 5]
323
324 a = [1, 2, 3]
325 b = a
326 a = a + [4, 5]
327 print(b)  # -->[1, 2, 3]
328
```

```
329 a = [1, 2, 3]
330 b = a
331 a.extend([4, 5])
332 print(b)  # -->[1, 2, 3, 4, 5]
333
334 a = [1, 2, 3]
335 b = a
336 a = a.extend([4, 5])
337 print(a)  # None
338 print(b)  # [1, 2, 3, 4, 5]
339
340 a = [1, 2, 3]  # try to make a =[10,20,30]
341 for x in a:
342     x = x * 10
343 print(a)  # [1, 2, 3] failed:) beacuse a[0]still is 1 . the
    right way ,a=[x*10 for x in a]
344
345 nums = [1, 2, 3]
346 print(nums.__iadd__([4, 5]))  # [1, 2, 3, 4, 5],inplace and
    return the new value
347 print(nums) #[1, 2, 3, 4, 5]
348 print(nums.extend([7, 8]))  #print None. inplace but no return
    value,so print None
349 print(nums) #[1, 2, 3, 4, 5, 7, 8]
350
351
352
353
354 nums = [1, 2, 3]
355
356 def modify():
357     print(nums)
358     nums.append(4)
359
360 modify()  # [1, 2, 3, 1]
361
362 def re_assign():
363     print(nums)
364     nums += [5]
365     # num=list.__iadd__(nums,[5]) this is modify first then
    re_assign,wont work for the assign part.
366

367 try:
368     re_assign()
```

```
except Exception as e:

    print(e)  # local variable 'nums' referenced before assignment# csv.DictWriter fieldnames doesn't have to in order as the original DictReader, but all fieldnames have to be there.to modify del reader['keys']before write to writer


# "Fact: Python passes function arguments by assigning to them."means when you call a function, you assign the parameter to the "value" of the arg.
# @nedbatchelder.com
# Let’s examine the most interesting of these alternate assignments: calling a function. When I define a function, I name its parameters:
# def my_func(x, y):
#     return x+y
# Here x and y are the parameters of the function my_func. When I call my_func, I provide actual values to be used as the arguments of the function. These values are assigned to the parameter names just as if an assignment statement had been used:
# When my_func is called, the name x has 8 assigned to it, and the name y has 9 assigned to it. That assignment works exactly the same as the simple assignment statements we’ve been talking about. The names x and y are local to the function, so when the function returns, those names go away. But if the values they refer to are still referenced by other names, the values live on

# https://nedbatchelder.com/text/names.html
def a_func(num):
    num = num + 2


num = 2
num = a_func(num)
print(num)  # - - > None , in that function , local num was asign to the value of global num ,which is 2,and  local var num assign to 4(2+2) , now we return the func. and global num assign to nothing :None. local num 4 was no accessable in the global scope.


# ITERATION
```

```
393 # a trick  zip(*[iter(s)]*n)
394 lst = range(10)
395 print(iter(lst) is iter(lst))  # False
396 print(list(zip(*[iter(lst)] * 3)))  # [(0, 1, 2), (3, 4, 5), (6,
    7, 8)]
397 # https://stackoverflow.com/questions/2233204/how-does-
    zipitersn-work-in-python
398 # zip(*lst) is funny
399 # [a]*n=[a,a,a,a,a....,a],same object a. so in this case
    [iter(lst)]*3 is != [iter(lst),iter(lst),iter(lst)],becasue
    three iter(lst) are 3 different objects.if you have
    to:a=iter(lst),then [iter(lst)]*3 =[a,a,a],By the way, range is
    not iterator.iter(lst) is Not iter(lst),but map is a
    iterator.see below:
400 # https://stackoverflow.com/questions/16425166/accumulate-items-
    in-a-list-of-tuples
401 # try to make lst = [(0, 0), (2, 3), (4, 3), (5, 1)] into
    new_lst = [(0, 0), (2, 3), (6, 6), (11, 7)]
402 lst = [(0, 0), (2, 3), (4, 3), (5, 1)]
403
404 import itertools
405 new_lst = zip(*lst)  # zip_object contains ((0,2,4,5),(0,3,3,1))
406 new_lst = map(itertools.accumulate, new_lst)  # map_object
    contains ((0,2,6,11),(0,3,6,7))
407 # print(iter(new_lst) is iter(new_lst)) #True ,so map is a
    iterator
408 new_lst = list(zip(*new_lst))
409 print(new_lst)  # [(0, 0), (2, 3), (6, 6), (11, 7)]
410 # so all in one line:
    list(zip(*map(itertools.accumulate,zip(*lst))))
411
412 # itertools
413 # islice doesn't  consume the original iterator until next is
    called. most(all) itertools are like that.
414
415 # from itertools doc
416 from collections import deque
417
418
419 def consume(iterator, n=None):
420     "Advance the iterator n-steps ahead. If n is None, consume
    entirely."

421     if n is None:
422         deque(iterable, maxlen=0)
```

```
423     else:
424         # itertools.islice(iterator,n,n) # THAT IS A NONO!!! islice doesn't  consume the original iterator  until Next is called!!!!!
425         next(itertools.islice(iterator, n, n), None)  # YES
426
427
428 def tail(n, iterable):
429     "Return an iterator over the last n items"
430     # tail(3, 'ABCDEFG') --> E F G
431     return iter(collections.deque(iterable, maxlen=n))
432
433
434 # cycle+compress, wanted a serial condition ,say one False and 20 True, forever
435 iterable = range(45)
436 result = itertools.compress(iterable, itertools.cycle(range(21)))  # 1-20,22-41,43,44
437
438 # itertools.repeat take container, not iterator. won't work.use repeat(tuple(iterator)).while cycle takes iterators.
439
440
441 # @accumulate usage: turn [1,2,3] in to int 123. or reduce
442 lst = [1, 2, 3]
443 result = itertools.accumulate(lst, lambda a, b: 10 * a + b)
444 print(list(result))  # [1, 12, 123]
445 # or use reduce , it is actually better
446 from functools import reduce
447 result = reduce(lambda a, b: 10 * a + b, lst)
448 print(result)  # 123
449
450 # takewhile,dropwhile,iter(callable func, sentinel(break) value);they works for <,>,=; to read a file by 32 characters -->iter(partial(f.read,32),'') see 'Transforming Code into Beautiful, Idiomatic Python'
451 # get all the fib nums  < 40,000
452 # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657]
453
454
455 def fib():
456     a, b = 0, 1
457     while True:
```

```
458         yield a
459         a, b = b, b + a
460
461
462 result = itertools.takewhile(lambda x: x < 40000, fib())  #
    right,or you can use generator expression+break func()
463
464
465 def breakfunc():
466     'for generator seeing StopIteration will automaticlly break
    loop'
467     raise StopIteration
468
469
470 result_2 = (x if x < 40000 else breakfunc() for x in fib())
471 print(list(result) == list(result_2))  # this is right ,too.
472
473
474 # groupby+defaultdict
475 # groupby :Itertools.groupby: 2 things need to point out, they
    are 1“the iterable needs to already be sorted on the same key
    function”. 2 “the source is shared, when the groupby() object is
    advanced, the previous group is no longer visible.” _doc
476 # it returns a tuple (key,A:iterator of the items that match the
    key)since this iterator shares the data of the groupby return
    value.when we iter over the return tuple, we need to capture the
    returned  A value right away.
477 # the standard  way will be to use “for key ,items in
    return_value: print key , list(items)”. so the problem I had
    before is I used the list()
478 """Must get the value right away!
479 for key , items in groupby:
480     use for loop store the items value into container.like :list
    or dictionay.most common way is list(items), you can use more
    complex as well.
481     see """
482 # https://stackoverflow.com/questions/3749512/python-group-by
483 input = [('11013331', 'KAT'), ('9085267', 'NOT'), ('5238761',
    'ETH'), ('5349618', 'ETH'), ('11788544', 'NOT'), ('962142',
    'ETH'), ('7795297', 'ETH'), ('7341464', 'ETH'), ('9843236',
    'KAT'), ('5594916', 'ETH'), ('1550003', 'ETH')]
484 # result = [
485 #             {
486 #               type:'KAT',
487 #               items: ['11013331', '9843236']
```

```
488 #             },
489 #             {
490 #               type:'NOT',
491 #               items: ['9085267', '11788544']
492 #             },
493 #             {
494 #               type:'ETH',
495 #               items: ['5238761', '962142', '7795297',
    '7341464', '5594916', '1550003']
496 #             }
497 #           ]
498 from operator import itemgetter
499 input = sorted(input, key=itemgetter(1))
500 result = itertools.groupby(input, key=itemgetter(1))
501 # for key, items in result:
502 #     print(f'{key}--->{list(items)}')
503
504 # TH--->[('5238761', 'ETH'), ('5349618', 'ETH'), ('962142',
    'ETH'), ('7795297', 'ETH'), ('7341464', 'ETH'), ('5594916',
    'ETH'), ('1550003', 'ETH')]
505 # KAT--->[('11013331', 'KAT'), ('9843236', 'KAT')]
506 # NOT--->[('9085267', 'NOT'), ('11788544', 'NOT')]
507 result = [{'type': key, 'items': [x for x, y in items]} for key,
    items in result]
508 import json
509 result = json.dumps(result, indent=2)
510 print(result) #yes
511
512 # now same thing again, with defaultdict
513 input = [('11013331', 'KAT'), ('9085267', 'NOT'), ('5238761',
    'ETH'), ('5349618', 'ETH'), ('11788544', 'NOT'), ('962142',
    'ETH'), ('7795297', 'ETH'), ('7341464', 'ETH'), ('9843236',
    'KAT'), ('5594916', 'ETH'), ('1550003', 'ETH')]
514 from collections import defaultdict
515 result=defaultdict(list)
516 for num,key in input:
517     result[key].append(num)
518 # print(result)
519 # defaultdict(<class 'list'>, {'KAT': ['11013331', '9843236'],
    'NOT': ['9085267', '11788544'], 'ETH': ['5238761', '5349618',
    '962142', '7795297', '7341464', '5594916', '1550003']})

520 result=[{'type': key, 'items':items} for key,items in
    result.items()]
521 result = json.dumps(result, indent=2)
```

```
522 print(result)#works too.
523
524 # another funny thing about groupby
525 # from [1,1,1,1,1,3,3,4,2,2,1,1,1,3,3] get[{1: 5}, {3: 2}, {4:
    1}, {2: 2}, {1: 3}, {3: 2}]
526 lst=[1,1,1,1,1,3,3,4,2,2,1,1,1,3,3] #without sorting
527 result=itertools.groupby(lst)
528 # for key,items in result:
529 #     print(key,'--->',list(items))
530 # 1 ---> [1, 1, 1, 1, 1]
531 # 3 ---> [3, 3]
532 # 4 ---> [4]
533 # 2 ---> [2, 2]
534 # 1 ---> [1, 1, 1]
535 # 3 ---> [3, 3]
536 result=[{key:len(list(items))}for key ,items in result]
537 #if you only use {key:len(list(items))} ,you will get your
    result updated.you will get {1: 3, 3: 2, 4: 1, 2: 2}
538
539 print(result) #[{1: 5}, {3: 2}, {4: 1}, {2: 2}, {1: 3}, {3: 2}]
540
541 from collections import Counter
542 lst=[1,1,1,1,1,3,3,4,2,2,1,1,1,3,3]
543 result=Counter(lst)
544 print(result) #Counter({1: 8, 3: 4, 2: 2, 4: 1})
545 print(result.most_common(2))#[(1, 8), (3, 4)]
546
547 #the Great Raymond Hettinger's  Section
548 #Transforming Code into Beautiful, Idiomatic Python + Python
    Class Toolkit
549 # iter(callable_func,sentinel_value)
550 # blocks=[]
551 # for block in iter(functools.partial(f.read,32),''):
552 #     blocks.appem=nd(block)
553 #
554 # for loop ,else:no break
555 dct= {'matthew': 'blue', 'rachel': 'green', 'raymond': 'red'}
556 # 'for key in dct' vs 'for key in list(dct)' --->when you are
    mutating the dictionary.
557 try :
558     for k in dct:
559         if k.startswith('r'):
560             del dct[k]
561 except Exception as e:
```

```
562     print(e) #dictionary changed size during iteration
563 for key in list(dct):
564     if key.startswith('r'):
565         del dct[key]
566 print(dct) # {'matthew': 'blue'} --->works
567 
568 dct= {'matthew': 'blue', 'rachel': 'green', 'raymond': 'red'}
569 while dct:
570     key,value = dct.popitem()
571     print(key,'---->',value)
572 # raymond ----> red
573 # rachel ----> green
574 # matthew ----> blue
575 
576 # defaultdict for counting (collections.Counter),
    grouping(itertools.groupby
577 #count
578 colors = ['red', 'green', 'red', 'blue', 'green', 'red']
579 from collections import defaultdict
580 result=defaultdict(int)
581 for color in colors:
582     result[color]+=1
583 print(result) #defaultdict(<class 'int'>, {'red': 3, 'green': 2,
    'blue': 1})
584 # use Counter
585 from collections import Counter
586 result=Counter(colors)
587 print(result)#Counter({'red': 3, 'green': 2, 'blue': 1})
588 # use nothing(get)
589 result={}
590 for color in colors :
591     result[color]=result.get(color,0)+1
592 print(result) #{'red': 3, 'green': 2, 'blue': 1}
593 #group
594 names = ['raymond', 'rachel', 'matthew', 'roger','betty',
    'melissa', 'judith', 'charlie']
595 result_1=defaultdict(list)
596 for name in names:
597     key=len(name)
598     result_1[key].append(name)
599 print(result_1)
600 #defaultdict(<class 'list'>, {7: ['raymond', 'matthew',
    'melissa', 'charlie'], 6: ['rachel', 'judith'], 5: ['roger',
    'betty']})
```

```
601 result_2={}
602 for name in names:
603     key=len(name)
604     result_2[key]=result_2.get(key,[])+[name]#be careful, don't
    use append, becauese it returns nothing,result_2[key] will be
    None
605 print(result_2)
606 # {7: ['raymond', 'matthew', 'melissa', 'charlie'], 6:
    ['rachel', 'judith'], 5: ['roger', 'betty']}
607 import itertools
608 result_3=sorted(names,key=len)
609 result_3=itertools.groupby(result_3,key=len)
610 # for key,names in result_3:
611 #     print(key,'--->',list(names))
612 # 5 ---> ['roger', 'betty']
613 # 6 ---> ['rachel', 'judith']
614 # 7 ---> ['raymond', 'matthew', 'melissa', 'charlie']
615 result_3={key:list(names) for key,names in result_3}
616 print(result_3)
617 # {5: ['roger', 'betty'], 6: ['rachel', 'judith'], 7:
    ['raymond', 'matthew', 'melissa', 'charlie']}
618 "Linking dictionaries" 'ChainMap'
619 defaults = {'color': 'red', 'user': 'guest'}
620 envir={'user':'frank','login':'Unknown'}
621 command={'login':True}
622 from collections import ChainMap
623 result=ChainMap(command,envir,defaults) #high to low
624 print(result['color'])#red
625 print(result['login'])#True
626 print(result['user'])#frank
627
628
629 # use namedtuple as return tuple for clarity
630 from collections import namedtuple
631 def
    twitter_search(name,*,retweets=True,numtweets=0,popluar=False):
632     twsearch=namedtuple('twsearch',
    ['name','retweets','numtweets','popluar'])
633     result=twsearch(name,retweets,numtweets,popluar)
634     return result
635 obama=twitter_search('obama',retweets=False,numtweets=10,popluar
    =True)
636 print(obama)#twsearch(name='obama', retweets=False,
    numtweets=10, popluar=True)
```

```
637 #unpacking seqiences,high level of thinking.
638
639 from functools import wraps
640 # famous cache decorator
641 def cache(my_func):
642     saved={}
643     @wraps(my_func)
644     def wrapper(*args):
645         if args in saved:
646             print('returned from saved')
647             return saved[args]
648         result=my_func(*args)
649         saved[args]=result
650         print('return from func(*args)')
651         return result
652     return wrapper
653
654 @cache
655 def printer(a):
656     print(a.upper())
657 printer('a')
658 # A
659 # return from func(*args)
660 printer('a')
661 # returned from saved
662
663 # this is realy a bad example,because second time 'A' was not
    printed.i guess it wont work for this kind of func.
664 @cache
665 def rt(a):
666     return a.upper()
667 print(rt('a'))
668 # return from func(*args)
669 # A
670 print(rt('a'))
671 # returned from saved
672 # A
673 # works good this time:)

674
675
676
677 #the ignored (suppress) contextmanager
678 @contextlib.contextmanager
679 def suppress(*exceptions):
```

```
680     try:
681         yield
682     except exceptions:
683         pass
684 
685 # Codingbat
686 
687 # http://codingbat.com/prob/p118406
688 # We want to make a row of bricks that is goal inches long. We
    have a number of small bricks (1 inch each) and big bricks (5
    inches each). Return True if it is possible to make the goal by
    choosing from the given bricks.
689 def make_bricks(small, big, goal):
690   return small+5*big>=goal and (goal-small)//5<=big and
    goal%5<=small
691 
692 # http://codingbat.com/prob/p167025
693 # Return the sum of the numbers in the array, returning 0 for an
    empty array. Except the number 13 is very unlucky, so it does
    not count and numbers that come immediately after a 13 also do
    not count.
694 def sum13(nums):
695     nums=nums+[0]#important
696     result=[num for index,num in enumerate(nums) if not num==13
    and nums[index-1]!=13]
697     return sum(result)
698 
699 # http://codingbat.com/prob/p186048
700 # Return the number of times that the string "code" appears
    anywhere in the given string, except we'll accept any letter for
    the 'd', so "cope" and "cooe" count.
701 
702 def count_code(str):
703   str=str+' ' #important !'eaacow'
704   result=[x for index,x in enumerate(str) if x=='e' and
    str[index-2]=='o' and str[index-3]=='c' ]
705   return len(result)
706 
707 
708 # Return True if the given string contains an appearance of
    "xyz" where the xyz is not directly preceeded by a period (.).
    So "xxyz" counts but "x.xyz" does not.
709 # xyz_there('abcxyz') → True
710 # xyz_there('abc.xyz') → False
```

```
711 # xyz_there('xyz.abc') → True
712
713 def xyz_there(str):
714   str=str.replace('.xyz','wwww')# important can not do
    replace('.xyz','').
715   return 'xyz' in str
716
717 # any(iterable) all(iterable)
718 print(all(()) is True) #so true
719 print(any(())) #False
720
721 s='abcd'#--->['a', 'b', 'c', 'd']
722 try:
723     print(s.split(''))
724 except Exception as e:
725     print(e)#empty separator
726 # str.split('') wont work, just use list(str)
727 print('a'*3) #'aaa'
728 lst=['a',1,'b']
729 try:
730     print(''.join(lst))
731 except Exception as e:
732     print(e) #sequence item 1: expected str instance, NoneType
    found
733 # join is a str method, only works for string. not int or
    NoneType
734
735 print('a'[:4])#--->No Error ,gets 'a'
736 # print('a'[-4])
737 print('a'[:-4]) #''empty str
738 print('a'[-1:]) #'a'
739
740 '~~~~~~~'
741 "Python's Class Development Toolkit _ Raymond Hettinger"
742 #"Python is consenting as an adult language. We don't leave the
    locks on the door." _ Raymond Hettinger
743
744 """
745 Circles, Inc.
746 """
747
748
749 class Circle:  # python 3 is automatically a new style class.
    2.7 needs to inherit (object)
```

```
750     from math import pi
751     """An advvanced circle analytics toolkit"""
752     # don/t skip the elevator pitch ,your doc string.
753     # what is inside a class is effectlly a module ,it is like
    the code run in its own module.
754
755     print('i am defining a class')  # it will print only by
    defining it.
756     # raymond also talked about you can open file or for loop
    with in the class.
757
758     version = '0.1'  # class variable for shared data,while
    instance var for unique data. use str, or tuple
759     print('dont use bi_floats , try:0.1+0.7,you will get ', 0.1
    + 0.7)  # 0.7999999999999999
760
761     def __init__(self, radius):
762         # "__init__ " is not a constructor. is calling the class
    construct a instance.__init__ is 'poplulate' instance variable.
763         # one thing is for sure, user is gonna make lots of
    instance, i mean a lot .
764         print('i am running __init__')
765         self.radius = radius
766
767     def area(self):
768         return self.radius**2 * pi
769     # so far we are good to go, more method ? until user ask for
    it! before that,YAGNI:) Lean startup.
770
771
772 # First customer: Academia
773 # from random import random, seed
774 # seed(8675309)
775 # print 'Using Circuituous(tm) version', Circle.version
776 # n = 10
777 # circles = [Circle(random()) for i in xrange(n)]
778 # print 'The average area of', n, 'random circles'
779 # avg = sum([c.area() for c in circles]) / n
780 # print 'is %.1f' % avg
781 # print
782
783     def perimeter(self):
784         # new customer wants a perimeter method.
785         return self.radius * 2 * pi
```

```
786
787 # Second customer: Rubber sheet company
788 # cuts = [0.1, 0.7, 0.8]
789 # circles = [Circle(r) for r in cuts]
790 # for c in circles:
791 #     print 'A circlet with with a radius of', c.radius
792 #     print 'has a perimeter of', c.perimeter()
793 #     print 'and a cold area of', c.area()
794 #     c.radius *= 1.1
795 #     print 'and a warm area of', c.area()
796 #     print
797
798
799 # this customer changed the attribute "c.radius *= 1.1"
800 "if it is a variable, it is gonna change, sooner or later"  #
    R.H
801
802 # If you expose an attribute, expect users to all kinds of
    interesting things with it.
803
804
805 # 3rd customer Tire
806 class Tire(Circle):
807     'Tires are circles with a corrected perimeter'
808     # again
809     "if it is a variable, it is gonna change, sooner or later"
    # R.H
810
811
812 def perimeter(self):
813     'Circumference corrected for the rubber'
814     return Circle.perimeter(self) * 1.25
815
816
817 # t = Tire(22)
818 # print 'A tire of radius', t.radius
819 # print 'has an inner area of', t.area()
820 # print 'and an odometer corrected perimeter of',
821 # print t.perimeter()
822 # print
823
824
825 # Next customer: Na;onal graphics company
826 # bbd = 25.1
827 # c = Circle(bbd_to_radius(bbd)
```

```
828 # print 'A circle with a bbd of 25.1'
829 # print 'has a radius of', c.radius
830 # print 'an an area of', c.area()
831 # print
832
833 # c = Circle(bbd_to_radius(bbd)) -------> this is Baaaad!
834 'USE Alternative Constructor'
835 print(dict.fromkeys(['name', 'age', 'language']))
836 #{'name': None, 'age': None, 'language': None}
837
838 # /lets go back and add the alternative constructor
839
840 import math
841
842
843 class Circle:
844
845     'An advanced circle analytic toolkit'
846     version = '0.3'
847
848     def __init__(self, radius):
849         self.radius = radius
850
851     def area(self):
852         return math.pi * self.radius ** 2.0
853
854     def perimeter(self):
855         return 2.0 * math.pi * self.radius
856
857     @classmethod
858     # classmethod make sure you use cls , for the subclass usage
859     def from_bbd(cls, bbd):
860         radius = bbd / 2.0 / math.sqrt(2.0)
861         # return Circle(radius) NONO!
862         # classmethod make sure you use cls , for the subclass
    usage
863         return cls(radius)
864
865
866 c = Circle.from_bbd(25.1)
867 # print 'A circle with a bbd of 25.1'
868 # print 'has a radius of', c.radius
869 # print 'an an area of', c.area()
870 # print
```

```
871
872 # New customer request: add a func
873 # use staticmethod ,a giveaway is your func does not need 'self'
    or 'cls'. you use staticmethod for the findability of your func.
874
875
876 class Circle(object):
877     'An advanced circle analytic toolkit'
878     version = '0.4'
879
880     def __init__(self, radius):
881         self.radius = radius
882
883     @staticmethod
884     # attach functions to classes to increase the findability of
    your func.
885     # a giveaway is your func does not need 'self' or 'cls'.
886     def angle_to_grade(angle):
887         'Convert angle in degree to a percentage grade'
888         return math.tan(math.radians(angle)) * 100.0
889
890
891 # Government request: ISO-11110: "you need to use perimeter to
    calc the area" ,like this:
892
893 # class Circle(object):
894 #     'An advanced circle analytic toolkit'
895 #     version = '0.5b'
896 #     def __init__(self, radius):
897 #             self.radius = radius
898 #     def area(self):
899 #         p = self.perimeter()
900 #         r = p / math.pi / 2.0 return math.pi * r ** 2.0
901 #     def perimeter(self):
902 #         return 2.0 * math.pi * self.radius
903
904
905 # that wasnot too bad,really?
906 # the Tire subclass update the perimeter, now you broke their
    code.
907
908 # class Tire(Circle):
909 #     'Tires are circles with an odometer corrected perimeter'
910 # def perimeter(self):
```

```
911 # 'Circumference corrected for the rubber' return
    Circle.perimeter(self) * 1.25
912
913
914 'so what to do?'  # normally 'self' means you or your
    children.in this case. self.perimeter(). means if tire class has
    this method.it will not look up to the mother class.So you want
    to make 'self' means you Only ------>local reference.
915 # the idea is to  use classname+methodname.
916 # __perimeter---> Name mangling into--->
    '_(class.__name__)__perimeter'
917
918
919 class Circle:
920     def __init__(self, radius):
921         self.radius = radius
922
923     def perimeter(self):
924         return self.radius * 2 * math.pi
925
926     # make local refernce perimeter
927     __perimeter = perimeter
928
929     # see Ned Batchelder - Facts and Myths about Python names
    and values - PyCon 2015
930     # a=3
931     # b=a
932     # a=4
933     # print(b)---->3
934
935     def area(self):
936         p = self.__perimeter()
937         r = p / (2 * math.pi)
938         return math.pi * r**2
939
940
941 # Government request: ISO-22220
942 # •  You're not allowed to store the radius
943 # •  You must store the diameter instead!
944
945 # we get to keep the api the same. still i accept radius in
    __init__, but diameter will be stored instead.
946
947
```

```
948 # it breaks our entire class!
949 #" I just wish everytime i use dot for look up,            it
    will magicly trans into a get method call ()"
950 #" I just wish everytime I set a radius(even in __init__) ,it
    will magicly trasn in to s set radius call,--store the
    diameter."
951 # yes, this is the @property .But dont do it just for it.dot
    look up and '=' assign is much easier."if you find yourself
    design a setter and getter,you probably doing it wrong"
952 # property is for "after the fact , that you dont need to change
    any existing code.and add on the property"
953
954
955 # User request: Many circles
956 # n = 10000000
957 # seed(8675309)
958 # print 'Using Circuituous(tm) version', Circle.version
959 # circles = [Circle(random()) for i in xrange(n)]
960 # print 'The average area of', n, 'random circles'
961 # avg = sum([c.area() for c in circles]) / n
962 # print 'is %.1f' % avg
963 # print
964 # I sense a major memory problem.
965 # Circle instances are over 300 bytes each!
966
967 'Flyweight design paUern: Slots'
968 # save this for the last.you cant add new attr ,you cant access
    the dictinary no more.no vars() or .__dict__.
969 # "from the user view, there are no changes at all"_R.H
970 # dont worry ,subclass does not inherit the slots
971
972
973 class Circle(object):
974
975     'An advanced circle analytic toolkit'
976 # flyweight design pattern suppresses
977 # the instance dictionary
978     __slots__ = ['diameter']
979     version = '0.7'
980
981     def __init__(self, radius):
982
983         self.radius = radius
984
985     @property  # convert dotted access to method calls
```

```
    def radius(self):
        return self.diameter / 2.0

    @radius.setter
    def radius(self, radius):
        self.diameter = radius * 2.0


"""Summary: Toolset for New - Style Classes
1.  Inherit from object().
2.  Instance variables for informa
on unique to an instance.
3.  Class variables for data shared among all instances.
4.  Regular methods need “self” to operate on instance data.
5.  Thread local calls use the double underscore. Gives subclasses the freedom to override methods without breaking other methods.
6.  Class methods implement alterna
ve constructors. They need “cls” so they can create subclass instances as well.
7.  Sta
c methods aUach func
ons to classes. They don’t need either “self” or “cls”. Sta
c methods improve discoverability and require context to be specified.
8.  A property() lets geUer and seUer methods be invoked automa
cally by aUribute access. This allows Python classes to freely expose their instance variables.
9.  The “__slots__” variable implements the Flyweight Design PaUern by suppressing instance dic
onaries."""


# from until_mar_29.py
# classmethod always use cls for subclass
#__repr__ usage
#__from_string__usage, if string contain classname then: classname,*info=info_string.split(' ')


class Employee:

    def __init__(self, first, last):
        self.first = first
```

```
1023            self.last = last
1024
1025        @classmethod
1026        def from_str(cls, info_string):
1027            return cls(*info_string.split(' '))
1028
1029        def __repr__(self):
1030            return f'{self.__class__.__name__}
    {tuple(vars(self).values())}'
1031
1032
1033    emp_1 = Employee.from_str('frank young')
1034    print(emp_1)
1035
1036
1037    # cache decorator with default dict
1038    # beware  use "()" in @cache_with_default() even when  no
    default args are passed in,  that make you go to the depper
    level into the wrapper func
1039    from functools import wraps
1040
1041
1042    def cache_with_default(saved=None):
1043        # cannot use mutable value for keyword args,use None for
    different my_func passed in, if only 1 my_func is gonna passed
    in, you dont need decorator, just pass the saved={},into cache
    level.see 'Transforming Code into Beautiful, Idiomatic Python'
1044        if saved == None:
1045            saved = {}
1046
1047        def cache(my_func):
1048            @wraps(my_func)
1049            def wrapper(*args):
1050                if args in saved:
1051                    print('return from saved dict')
1052                    print(saved)
1053                    return saved[args]
1054                result = my_func(*args)
1055                saved[args] = result
1056                print('return function called')
1057                print(saved)
1058                return result
1059            return wrapper
1060        return cache
```

```
1061
1062
1063  @cache_with_default(saved={(1,): 123})
1064  def my_func(a):
1065      return a**2
1066
1067
1068  print(my_func(1))
1069  # return from saved dict
1070  # {(1,): 123}
1071  # 123
1072  print(my_func(2))
1073  # return function called
1074  # {(1,): 123, (2,): 4}
1075  # 4
1076  # [Finished in 0.1s]
1077
1078
1079  # global can be accssed (print,modify(mutable) ), but cant not
      be used to re-assign
1080
1081
```