05/13/18 03:45:00 /Users/frankyoung/Documents/Python3/18 May/Nobody_gets_ready.py

```
 1  # string formatting
 2  from math import pi
 3  import datetime
 4
 5  lst = ['python', 'java']
 6  dct = {'name': 'frank', 'age': 27}
 7  dt = datetime.date.today()
 8  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)
 9  print(str_formatting)#the pi is 0,003.14, my name is frank, i am
    27 years old,i am learing python and java ,today is May 2018 12
10
11  # tried {name.upper()} ,Not Supported./AttributeError: 'str'
    object has no attribute 'upper()'
12  # "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.
13  name = 'frank'
14  print(f'{name.upper()}') #FRANK
15  try:
16      print('{.upper()}'.format(name))
17  except AttributeError as e:
18      print(e) #'str' object has no attribute 'upper()'
19
20  # any(iterable) all(iterable)
21  print(all(()) is True) #so true
22  print(any(())) #False
23
24  lst=['a',1,'b']
25  try:
26      print(''.join(lst))
27  except Exception as e:
28      print(e) #sequence item 1: expected str instance, NoneType
    found
29  # join is a str method, only works for string. not int or
    NoneType
30
31  print('a'[:4])#--->No Error ,gets 'a'
32  print('a'[:-4]) #No Error,''empty str
33  try:
```

```
    print('a'[-4])
except Exception as e:
    print(e) #sequence item 1: expected str instance, int found


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 info order, but you will have to modify the fieldnames first. keyword args has no order.
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
import datetime
from contextlib import suppress
def year_month_folder(path):
    os.chdir(path)
    for file in os.listdir():
        if os.path.isfile(file):

            mtime = os.stat(file).st_mtime
            mtime = datetime.date.fromtimestamp(mtime)
            folder_name = f'{mtime:%y %B}'
            with suppress(FileExistsError):#or you can use os.path.exists() as a condition,but at <Raymond Hettinger's Transforming Code into Beautiful, Idiomatic Python>-43:28 ,he said it is not a good way,because it has a raise condition in it.I don't know why.
                os.mkdir(folder_name)

```

```
            name_path = os.path.join(folder_name, file)
            os.rename(file, name_path)



"""object sorting"""
from operator import itemgetter, attrgetter, methodcaller

student_tuples = [('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
print(sorted(student_tuples, key=itemgetter(2)))
# [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
print(sorted(student_tuples, key=itemgetter(1, 2)))
# [('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

class Student:
    def __init__(self, name, grade, age):
        self.name = name
        self.grade = grade
        self.age = age

    def __repr__(self):
        return repr((self.name, self.grade, self.age))


student_objects = [Student('john', 'A', 15), Student('jane', 'B', 12), Student('dave', 'B', 10), ]

print(sorted(student_objects, key=attrgetter('age')))
# [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
print(sorted(student_objects, key=attrgetter('grade', 'age')))
# [('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

messages = ['critical!!!', 'hurry!', 'standby', 'immediate!!']
print(sorted(messages, key=methodcaller('count', '!')))
# ['standby', 'hurry!', 'immediate!!', 'critical!!!']


"""contextmanger"""
import os
import glob
import contextlib


```

```
112 @contextlib.contextmanager
113 def find_py(path):
114     ori_path = os.getcwd()
115     try:
116         os.chdir(path)
117         result = glob.glob('*.py')
118         yield result
119     finally:
120         os.chdir(ori_path)
121
122
123 # with find_py('/Users/frankyoung/Documents/Python3/18 March')
    as f:
124 #     for pyfile in f:
125 #         print(pyfile)
126 # print(os.getcwd())
127
128 # or use a class
129 class Find_Py:
130     def __init__(self, path):
131         self.path = path
132         self.ori_path = os.getcwd()
133
134     def __enter__(self):
135         os.chdir(self.path)
136         result = glob.glob('*.py')
137         return result
138
139     def __exit__(self, exc_type, exc_val, traceback):
140         os.chdir(self.ori_path)
141
142
143 # with Find_Py('/Users/frankyoung/Documents/Python3/18 March')
    as f:
144 #     for pyf in f:
145 #         print(pyf)
146 # print(os.getcwd())
147
148
149 @contextlib.contextmanager
150 # by Nick Coghlan
151 def suppress(*exceptions):
152     try:
153         yield
```

```
154         except exceptions:
155             pass
156
157
158 # Decorator,Scope,Closure
159 # the LEGB rule , for accessing(say,print) and modifying(append)
    mutable var only.But not reasigning after referenced to a
    nonlocal(enclosing,Global) variable .
160 # UnboundLocalError: local variable referenced to  before
    assignment.
161
162 "the scope of a var is determined when function is defined , the
    value of a var is determined when the function is called"
163 # <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."
164
165
166 # https://nedbatchelder.com/text/names.html
167 # Ned Batchelder - Facts and Myths about Python names and values
    - PyCon 2015
168 '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'
169 # '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.
170
171 a = 1
172 def p():
173     print(a)
174
175 a = 2
176 def w():
177     print(a)
178
179 p()  # 2,because a`s scope is determined as a global var when
    the function was defined, when the func call , it look up the
    global a`s value by then, which is 2.
```

```
180 w()  # 2
181
182
183 # http://docs.python-guide.org/en/latest/writing/gotchas/
184 # Late Binding Closures
185 funcs = [lambda x: x * i for i in range(3)]  # by the way , i
    here is a local var.so i doesnt not exist in the global scope.if
    try to print i , 'NameError: name 'i' is not defined'
186 for func in funcs:
187     print(func(2))
188 # 4,4,4
189 # same :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.
190 # Solution #1:use keyword args.'Python's default arguments are
    evaluated once when the function is defined, not each time the
    function is called '
191 funcs = [lambda x, i=i: x * i for i in range(3)]
192 for func in funcs:
193     print(func(2))
194 # 0,2,4
195 # Solution #2 you can use generator expression, without keyword
    args.because generator look up the value as it goes.
196 funcs = (lambda x: x * i for i in range(3))
197 for func in funcs:
198     print(func(2))
199 # 0,2,4
200 #
201 'function attribute, functions can have attributes.'
202 # in python , functions can have attribute.
203
204 # make a counter decorator using function attr.
205 from functools import wraps
206
207
208 def counter(my_func):
209     @wraps(my_func)
210     def inner(*args, **kwargs):
211         inner.count += 1
212         return my_func(*args, **kwargs)
213     inner.count = 0
214     return inner
215
216
```

```
@counter
def i_tell_you_what():
    return 'i tell you what'


i_tell_you_what()
i_tell_you_what()
print(f'{i_tell_you_what.__name__} run {i_tell_you_what.count} times')
# i_tell_you_what run 2 times

# make a cache with a default return dictionary as an arg

def cache_with_default(dct=None):
    if dct is None:
        dct = {}

    def cache(my_func):
        @wraps(my_func)
        def wrapper(*args):
            if args in dct:
                return dct[args]
            result = my_func(*args)
            dct[args] = result
            return result
        return wrapper
    return cache

# beware: use "()" in @cache_with_default() even when  no default args are passed in,  that takes you go to the depper level, into the wrapper func

@cache_with_default({(1,): 100})  # be careful, pass 1 as a tuple (1,) or it wont work.becaues the args will be (1,)
def times_two(x):
    return x + x


print(times_two(1))  # get 100 inside of 2 , because it was looked up in the dct


# Brett Slatkin - How to Be More Effective with Functions - PyCon 2015 - YouTube
```

```
255 # http://www.informit.com/articles/article.aspx?p=2314818
256
257 # keword_only_args:forced to be clear.
258 # dont pass infinity generator into *args, like
    itertools.count().it will try to tuple(count()),and that will
    crash.
259 # To avoid this possibility entirely, you should use keyword-
    only arguments when you want to extend functions that accept
    *args
260 # *this is a kwargs_only function, so :'anything after a "*" or
    "*args"  is FORCED to be clear (keyword)'
261
262
263 def kwargs_only(*args, a=1): #this a is forced to be clear,
    because a is after a "*"
264     print(a)
265     print(args)
266
267
268 kwargs_only(2, 2, 2, 2, a=4)
269 # 4
270 # (2, 2, 2, 2)
271
272 def bobby(*, propane=True, charcoal=False):
273     # it nice to be clear, when your args are the same type
    data.
274     if propane:
275         print('I sell propane and propane accessories')
276     else:
277         print('the hell you say')
278
279
280 try:
281     bobby(True, False)
282 except Exception as e:
283     print(e)  # bobby() takes 0 positional arguments but 2 were
    given
284
285 bobby(propane=True, charcoal=False)  # I sell propane and
    propane accessories
286 #same idea , if you can ,put return value into a namedtuple
    inside of a tuple ,to be clear.--> 'Raymond Hettinger's
    Transforming Code into Beautiful, Idiomatic Python'
287 # use namedtuple as return tuple for clarity
288 from collections import namedtuple
```

```
289 def
    twitter_search(name,*,retweets=True,numtweets=0,popluar=False):
290     twsearch=namedtuple('twsearch',
    ['name','retweets','numtweets','popluar'])
291     result=twsearch(name,retweets,numtweets,popluar)
292     return result
293 obama=twitter_search('obama',retweets=False,numtweets=10,popluar
    =True)
294 print(obama)#twsearch(name='obama', retweets=False,
    numtweets=10, popluar=True)
295
296
297
298 # what is generator?
299 # iter(foo) is iter(foo)
300 # base on the talk-->Brett Slatkin - How to Be More Effective
    with Functions
301 # if iter(foo) is iter(foo):
302 #       now,then =itertools.tee(foo,2)
303 # 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:
304 # compare this info_get function and Info_Gen Class:
305 # difference is ever time you call the 'for' , __iter__method on
    class, it return a new iterator over a container.
306 # so far I prefer itertools.tee ,it is easier.
307 def info_gen(path):
308     with open(path) as f:
309         reader = csv.DictReader(f)
310         for line in reader:
311             del line['Year']
312             yield line
313
314
315 class Info_Gen:
316     def __init__(self, path):
317         self.path = path
318         print(self.path)
319
320     def __iter__(self):
321         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.
```

```
322
323
324 # http://nvie.com/posts/iterators-vs-generators/   -----> "iter(iterable)-->iteration"
325 # how to detect a generator
326
327 lst = [1, 2, 3]
328 a = iter(lst)
329 b = iter(lst)
330 print(a is b)  # False
331 print(a == b)  # false
332 print([a] == [b])  # false
333 print(list(a) == list(b))  # true
334 c = iter(a)
335 print(a is c)  # True
336 print(a == c)  # True
337 print([a] == [c])  # false
338
339 lst = [1, 2, 3]
340 a = iter(lst)
341 c = iter(a)
342 print(list(a) == list(c))  # False  ([1,2,3] ==[])
343 # print(list(a))
344 # print(list(c))
345 # 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.'
346 # 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.
347
348
349 # how does generator function(yield) run?
350
351 import contextlib
352
353
354 def HYW():
355     print('hello')
356     yield
```

```
357     print('world')
358
359
360 a = HYW()  # Nothing happend, ! hello was not printed.
361 next(a)  # ---> now hello was printed. so when you call next,
    generator will run till it hits a yield
362
363 with contextlib.suppress(StopIteration):
364     next(a)  # -----> world was printed, and then it hits the
    StopIteration
365
366
367
368
369
370 #  Ned Batchelder - Facts and Myths about Python names and
    values - PyCon 2015
371 # "reassign one of the name ,brother,doesnt reassign the other"
    ---Ned
372
373 a = [1, 2, 3]
374 b = a
375 a += [4, 5]  # what happened here unline is "a.extend([4,5]) and
    a =a "
376 print(b)  # -->[1, 2, 3, 4, 5]
377
378 a = [1, 2, 3]
379 b = a
380 a = a + [4, 5]
381 print(b)  # -->[1, 2, 3]
382
383 a = [1, 2, 3]
384 b = a
385 a.extend([4, 5])
386 print(b)  # -->[1, 2, 3, 4, 5]
387
388 a = [1, 2, 3]
389 b = a
390 a = a.extend([4, 5])
391 print(a)  # None
392 print(b)  # [1, 2, 3, 4, 5]
393
394 a = [1, 2, 3]  # try to make a =[10,20,30]
395 for x in a:
```

```
396         x = x * 10
397     print(a)  # [1, 2, 3] failed:) beacuse a[0]still is 1 . the
        right way ,a=[x*10 for x in a]
398
399     nums = [1, 2, 3]
400     print(nums.__iadd__([4, 5]))  # [1, 2, 3, 4, 5],inplace and
        return the new value
401     print(nums) #[1, 2, 3, 4, 5]
402     print(nums.extend([7, 8]))  #print None. inplace but no return
        value,so print None
403     print(nums) #[1, 2, 3, 4, 5, 7, 8]
404
405
406
407
408     nums = [1, 2, 3]
409
410     def modify():
411         print(nums)
412         nums.append(4)
413
414     modify()  # [1, 2, 3, 1]
415
416     def re_assign():
417         print(nums)
418         nums += [5]
419         # num=list.__iadd__(nums,[5]) this is modify first then
        re_assign,wont work for the assign part.
420
421     try:
422         re_assign()
423     except Exception as e:
424
425         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
426
427
428     # "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.

429     # @nedbatchelder.com
430     # Let’s examine the most interesting of these alternate
```

```
    assignments: calling a function. When I define a function, I
    name its parameters:
431 # def my_func(x, y):
432 #     return x+y
433 # 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:
434 # 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
435
436 # https://nedbatchelder.com/text/names.html
437 def a_func(num):
438     num = num + 2
439
440
441 num = 2
442 num = a_func(num)
443
444 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.
445
446
447 # ITERATION
448 # a trick  zip(*[iter(s)]*n)
449 lst = range(10)#[0,1,2,3,4,5,6,7,8,9]
450 print(iter(lst) is iter(lst))  # False
451 print(list(zip(*[iter(lst)] * 3)))  # [(0, 1, 2), (3, 4, 5), (6,
    7, 8)]
452 import itertools
453 print(list(itertools.zip_longest(*[iter(lst)]*3))) #[(0, 1, 2),
    (3, 4, 5), (6, 7, 8), (9, None, None)]
454 # https://stackoverflow.com/questions/2233204/how-does-
    zipitersn-work-in-python
455 # zip(*lst) is funny
456 # [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.so iter(lst) is Not iter(lst),
457 # but map is a iterator.see below:
458 # https://stackoverflow.com/questions/16425166/accumulate-items-
    in-a-list-of-tuples
459
460 # try to make lst = [(0, 0), (2, 3), (4, 3), (5, 1)] into
    new_lst = [(0, 0), (2, 3), (6, 6), (11, 7)]
461 lst = [(0, 0), (2, 3), (4, 3), (5, 1)]
462
463 import itertools
464 new_lst = zip(*lst)  # zip_object contains ((0,2,4,5),(0,3,3,1))
465 new_lst = map(itertools.accumulate, new_lst)  # map_object
    contains ((0,2,6,11),(0,3,6,7))
466 # print(iter(new_lst) is iter(new_lst)) #True ,so map is a
    iterator
467 new_lst = list(zip(*new_lst))
468 print(new_lst)  # [(0, 0), (2, 3), (6, 6), (11, 7)]
469 # so all in one line:
    list(zip(*map(itertools.accumulate,zip(*lst))))
470
471 # itertools
472 # islice doesn't  consume the original iterator until next is
    called. most(all) itertools are like that.
473
474 # from itertools doc
475 from collections import deque
476
477
478 def consume(iterator, n=None):
479     "Advance the iterator n-steps ahead. If n is None, consume
    entirely."
480     if n is None:
481         deque(iterable, maxlen=0)
482     else:
483         # itertools.islice(iterator,n,n) # THAT IS A NONO!!!
    islice doesn't  consume the original iterator  until Next is
    called!!!!!
484         next(itertools.islice(iterator, n, n), None)  # YES
485
486
487 def tail(n, iterable):
488     "Return an iterator over the last n items"
```

```
489     # tail(3, 'ABCDEFG') --> E F G
490     return iter(collections.deque(iterable, maxlen=n))
491
492
493 # cycle+compress, wanted a serial condition ,say one False and
    20 True, forever
494 iterable = range(45)
495 result = itertools.compress(iterable,
    itertools.cycle(range(21)))  # 1-20,22-41,43,44
496
497 # itertools.repeat take container, not iterator. won't work.use
    repeat(tuple(iterator)).while cycle takes iterators.
498
499
500 # @accumulate usage: turn [1,2,3] in to int 123. or reduce
501 lst = [1, 2, 3]
502 result = itertools.accumulate(lst, lambda a, b: 10 * a + b)
503 print(list(result))  # [1, 12, 123]
504 # or use reduce , it is actually better
505 from functools import reduce
506 result = reduce(lambda a, b: 10 * a + b, lst)
507 print(result)  # 123
508
509 # 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'
510 # get all the fib nums  < 40,000
511 # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
    987, 1597, 2584, 4181, 6765, 10946, 17711, 28657]
512
513
514 def fib():
515     a, b = 0, 1
516     #unpacking seqiences,high level of thinking._R.H
517     while True:
518         yield a
519         a, b = b, b + a
520
521

522 result = itertools.takewhile(lambda x: x < 40000, fib())  #
    right,or you can use generator expression+break func()
523
524 # or:
```

```
525
526 def breakfunc():
527     'for generator seeing StopIteration will automaticlly break
    loop'
528     raise StopIteration
529
530
531 result_2 = (x if x < 40000 else breakfunc() for x in fib())
532 print(list(result) == list(result_2))  # this also works
533
534
535 # groupby+defaultdict
536 # 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
537 # 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.
538 # 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()
539 """Must get the value right away!
540 for key , items in groupby:
541     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.
542     see """
543 # https://stackoverflow.com/questions/3749512/python-group-by
544 input = [('11013331', 'KAT'), ('9085267', 'NOT'), ('5238761',
    'ETH'), ('5349618', 'ETH'), ('11788544', 'NOT'), ('962142',
    'ETH'), ('7795297', 'ETH'), ('7341464', 'ETH'), ('9843236',
    'KAT'), ('5594916', 'ETH'), ('1550003', 'ETH')]
545 # result = [
546 #           {
547 #             type:'KAT',
548 #             items: ['11013331', '9843236']
549 #           },
550 #           {
551 #             type:'NOT',
552 #             items: ['9085267', '11788544']
553 #           },
554 #           {
```

```
555 #              type:'ETH',
556 #              items: ['5238761', '962142', '7795297',
    '7341464', '5594916', '1550003']
557 #            }
558 #          ]
559 from operator import itemgetter
560 input = sorted(input, key=itemgetter(1))
561 result = itertools.groupby(input, key=itemgetter(1))
562 # for key, items in result:
563 #     print(f'{key}--->{list(items)}')
564
565 # TH--->[('5238761', 'ETH'), ('5349618', 'ETH'), ('962142',
    'ETH'), ('7795297', 'ETH'), ('7341464', 'ETH'), ('5594916',
    'ETH'), ('1550003', 'ETH')]
566 # KAT--->[('11013331', 'KAT'), ('9843236', 'KAT')]
567 # NOT--->[('9085267', 'NOT'), ('11788544', 'NOT')]
568 result = [{'type': key, 'items': [x for x, y in items]} for key,
    items in result]
569 import json
570 result = json.dumps(result, indent=2)
571 print(result) #yes
572
573 # now same thing again, with defaultdict
574 input = [('11013331', 'KAT'), ('9085267', 'NOT'), ('5238761',
    'ETH'), ('5349618', 'ETH'), ('11788544', 'NOT'), ('962142',
    'ETH'), ('7795297', 'ETH'), ('7341464', 'ETH'), ('9843236',
    'KAT'), ('5594916', 'ETH'), ('1550003', 'ETH')]
575 from collections import defaultdict
576 result=defaultdict(list)
577 for num,key in input:
578     result[key].append(num)
579 # print(result)
580 # defaultdict(<class 'list'>, {'KAT': ['11013331', '9843236'],
    'NOT': ['9085267', '11788544'], 'ETH': ['5238761', '5349618',
    '962142', '7795297', '7341464', '5594916', '1550003']})
581 result=[{'type': key, 'items':items} for key,items in
    result.items()]
582 result = json.dumps(result, indent=2)
583 print(result)#works too.

584
585 # another funny thing about groupby
586 # 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}]
587 lst=[1,1,1,1,1,3,3,4,2,2,1,1,1,3,3] #without sorting
```

```
588 result=itertools.groupby(lst)
589 # for key,items in result:
590 #     print(key,'--->',list(items))
591 # 1 ---> [1, 1, 1, 1, 1]
592 # 3 ---> [3, 3]
593 # 4 ---> [4]
594 # 2 ---> [2, 2]
595 # 1 ---> [1, 1, 1]
596 # 3 ---> [3, 3]
597 result=[{key:len(list(items))}for key ,items in result]
598 #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}
599
600 print(result) #[{1: 5}, {3: 2}, {4: 1}, {2: 2}, {1: 3}, {3: 2}]
601
602 from collections import Counter
603 lst=[1,1,1,1,1,3,3,4,2,2,1,1,1,3,3]
604 result=Counter(lst)
605 print(result) #Counter({1: 8, 3: 4, 2: 2, 4: 1})
606 print(result.most_common(2))#[(1, 8), (3, 4)]
607
608 # some from Codingbat
609
610 # http://codingbat.com/prob/p118406
611 # 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.
612 def make_bricks(small, big, goal):
613   return small+5*big>=goal and (goal-small)//5<=big and
    goal%5<=small
614
615 # http://codingbat.com/prob/p167025
616 # 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.
617 def sum13(nums):
618     nums=nums+[0]#important
619     result=[num for index,num in enumerate(nums) if not num==13
    and nums[index-1]!=13]
620     return sum(result)
621
622 # http://codingbat.com/prob/p186048
```

```
623 # 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.
624
625 def count_code(str):
626   str=str+' ' #important !'eaacow'
627   result=[x for index,x in enumerate(str) if x=='e' and
    str[index-2]=='o' and str[index-3]=='c' ]
628   return len(result)
629
630
631 # 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.
632 # xyz_there('abcxyz') → True
633 # xyz_there('abc.xyz') → False
634 # xyz_there('xyz.abc') → True
635
636 # the logic of this problem is the highlight
637 def xyz_there(str):
638   str=str.replace('.xyz','wwww')# important can not do
    replace('.xyz','').
639   return 'xyz' in str
640
641
642
643 "the Great Raymond Hettinger's  Section"
644 #Transforming Code into Beautiful, Idiomatic Python + Python
    Class Toolkit
645 # iter(callable_func,sentinel_value)
646 # blocks=[]
647 # for block in iter(functools.partial(f.read,32),''):
648 #     blocks.appem=nd(block)
649 #
650 # for loop ,else:no break
651 dct= {'matthew': 'blue', 'rachel': 'green', 'raymond': 'red'}
652 # 'for key in dct' vs 'for key in list(dct)' --->when you are
    mutating the dictionary.
653
654 # "if you mutating something while you iter over it, you are
    living in the state of sin, and you deserve whatever happens to
    you"
655 # list is ever worse, make sure you don't do that,just make a
    new list.-->[x for index,x in enumerate(lst) if index%2==0]
```

```
656
657 try :
658     for k in dct:
659         if k.startswith('r'):
660             del dct[k]
661 except Exception as e:
662     print(e) #dictionary changed size during iteration
663
664
665 for key in list(dct):
666     if key.startswith('r'):
667         del dct[key]
668 print(dct) # {'matthew': 'blue'} --->works
669
670 dct= {'matthew': 'blue', 'rachel': 'green', 'raymond': 'red'}
671 while dct:
672     key,value = dct.popitem()
673     print(f'I just popped {key}---->{value}')
674 # I just popped raymond---->red
675 # I just popped rachel---->green
676 # I just popped matthew---->blue
677
678
679 # defaultdict for counting (collections.Counter),
    grouping(itertools.groupby
680
681
682
683 colors = ['red', 'green', 'red', 'blue', 'green', 'red']
684
685 # defaultdict
686 from collections import defaultdict
687 result=defaultdict(int)
688 for color in colors:
689     result[color]+=1
690 print(result) #defaultdict(<class 'int'>, {'red': 3, 'green': 2,
    'blue': 1})
691
692 # use Counter
693 from collections import Counter
694 result=Counter(colors)
695 print(result)#Counter({'red': 3, 'green': 2, 'blue': 1})
696
697 # use nothing(get)
698 result={}
```

```
699 for color in colors :
700     result[color]=result.get(color,0)+1
701 print(result) #{'red': 3, 'green': 2, 'blue': 1}
702
703 #group
704
705 #defaultdict
706 names = ['raymond', 'rachel', 'matthew', 'roger','betty',
    'melissa', 'judith', 'charlie']
707 result_1=defaultdict(list)
708 for name in names:
709     key=len(name)
710     result_1[key].append(name)
711 print(result_1)
712 #defaultdict(<class 'list'>, {7: ['raymond', 'matthew',
    'melissa', 'charlie'], 6: ['rachel', 'judith'], 5: ['roger',
    'betty']})
713
714 #use get
715 result_2={}
716 for name in names:
717     key=len(name)
718     result_2[key]=result_2.get(key,[])+[name]#be careful, don't
    use append, becauese it returns nothing,result_2[key] will be
    None
719 print(result_2)
720 # {7: ['raymond', 'matthew', 'melissa', 'charlie'], 6:
    ['rachel', 'judith'], 5: ['roger', 'betty']}
721
722
723 #groupby
724 import itertools
725 result_3=sorted(names,key=len)
726 result_3=itertools.groupby(result_3,key=len)
727 # for key,names in result_3:

728 #     print(key,'--->',list(names))
729 # 5 ---> ['roger', 'betty']
730 # 6 ---> ['rachel', 'judith']
731 # 7 ---> ['raymond', 'matthew', 'melissa', 'charlie']
732 result_3={key:list(names) for key,names in result_3}
733 print(result_3)
734 # {5: ['roger', 'betty'], 6: ['rachel', 'judith'], 7:
    ['raymond', 'matthew', 'melissa', 'charlie']}
735
```

```
"Linking dictionaries" 'ChainMap'
defaults = {'color': 'red', 'user': 'guest'}
envir={'user':'frank','login':'Unknown'}
command={'login':True}
from collections import ChainMap
result=ChainMap(command,envir,defaults) #high to low
print(result['color'])#red
print(result['login'])#True
print(result['user'])#frank



from functools import wraps
# famous cache decorator
def cache(my_func):
    saved={}
    @wraps(my_func)
    def wrapper(*args):
        if args in saved:
            print('returned from saved')
            return saved[args]
        result=my_func(*args)
        saved[args]=result
        print('return from func(*args)')
        return result
    return wrapper

@cache
def printer(a):
    print(a.upper())
printer('a')
# A
# return from func(*args)
printer('a')
# returned from saved

# this is realy a bad example,because second time 'A' was not printed.so it doesn't not work for all functions.

@cache
def rt(a):
    return a.upper()
print(rt('a'))
```

```
779 # return from func(*args)
780 # A
781 print(rt('a'))
782 # returned from saved
783 # A
784 # works good this time:)
785
786 # the setup,teardown in sqlite "with conn"
787 with contextlib.suppress(Exception):#since we have no database
    working now
788     with conn:
789         cur = conn.cursor()
790         cur.execute( ... )
791
792 # The patch contextmanager, this testing code is from
    CoreyMSchafer--> https://github.com/CoreyMSchafer
793 import requests
794
795 class Employee:
796
797     def __init__(self, first, last, pay):
798         self.first = first
799         self.last = last
800         self.pay = pay
801
802     def monthly_schedule(self, month):
803         response =
    requests.get(f'http://company.com/{self.last}/{month}')
804         if response.ok:
805             return response.text
806         else:
807             return 'Bad Response!'
808
809 # and another module :
810 import unittest
811 # from employee import Employee
812 from unittest.mock import patch
813 class TestEmployee(unittest.TestCase):
814     def test_monthly_schedule(self):
815         with patch('employee.requests.get') as mocked_get:
816             mocked_get.return_value.ok = True
817             mocked_get.return_value.text = 'Success'
818
819             schedule = self.emp_1.monthly_schedule('May')
820
```

```
    mocked_get.assert_called_with('http://company.com/Schafer/May')
821             self.assertEqual(schedule, 'Success')
822
823             mocked_get.return_value.ok = False
824
825             schedule = self.emp_2.monthly_schedule('June')
826
    mocked_get.assert_called_with('http://company.com/Smith/June')
827             self.assertEqual(schedule, 'Bad Response!')
828
829 # if __name__=='__main__':
830     # unittest.main()
831
832
833
834 "Python's Class Development Toolkit _ Raymond Hettinger"
835 #"Python is consenting as an adult language. We don't leave the
    locks on the door." _ Raymond Hettinger
836
837 """
838 Circles, Inc.
839 """
840
841
842 class Circle:  # python 3 is automatically a new style class.
    2.7 needs to inherit (object)
843     from math import pi
844     """An advvanced circle analytics toolkit"""
845     # don/t skip the elevator pitch ,your doc string.
846     # what is inside a class is effectlly a module ,it is like
    the code run in its own module.
847
848     print('i am defining a class')  # it will print only by
    defining it.
849     # raymond also talked about you can open file or for loop
    with in the class.
850
851     version = '0.1'  # class variable for shared data,while
    instance var for unique data. use str, or tuple
852     print('dont use bi_floats , try:0.1+0.7,you will get ', 0.1
    + 0.7)  # 0.7999999999999999
853
854     def __init__(self, radius):
855         # "__init__ " is not a constructor. is calling the class
```

```
    construct a instance.__init__ is 'poplulate' instance variable.
856         # one thing is for sure, user is gonna make lots of
    instance, i mean a lot .
857         print('i am running __init__')
858         self.radius = radius
859
860     def area(self):
861         return self.radius**2 * pi
862     # so far we are good to go, more method ? until user ask for
    it! before that,YAGNI:) Lean startup.
863
864
865 # First customer: Academia
866 # from random import random, seed
867 # seed(8675309)
868 # print 'Using Circuituous(tm) version', Circle.version
869 # n = 10
870 # circles = [Circle(random()) for i in xrange(n)]
871 # print 'The average area of', n, 'random circles'
872 # avg = sum([c.area() for c in circles]) / n
873 # print 'is %.1f' % avg
874 # print
875
876     def perimeter(self):
877         # new customer wants a perimeter method.
878         return self.radius * 2 * pi
879
880 # Second customer: Rubber sheet company
881 # cuts = [0.1, 0.7, 0.8]
882 # circles = [Circle(r) for r in cuts]
883 # for c in circles:
884 #     print 'A circlet with with a radius of', c.radius
885 #     print 'has a perimeter of', c.perimeter()
886 #     print 'and a cold area of', c.area()
887 #     c.radius *= 1.1
888 #     print 'and a warm area of', c.area()
889 #     print
890
891
892 # this customer changed the attribute "c.radius *= 1.1"
893 "if it is a variable, it is gonna change, sooner or later"  #
    R.H
894
895 # If you expose an attribute, expect users to all kinds of
```

```
    interesting things with it.
896
897
898 # 3rd customer Tire
899 class Tire(Circle):
900     'Tires are circles with a corrected perimeter'
901     # again
902     "if it is a variable, it is gonna change, sooner or later"
    # R.H
903
904
905 def perimeter(self):
906     'Circumference corrected for the rubber'
907     return Circle.perimeter(self) * 1.25
908
909
910 # t = Tire(22)
911 # print 'A tire of radius', t.radius
912 # print 'has an inner area of', t.area()
913 # print 'and an odometer corrected perimeter of',
914 # print t.perimeter()
915 # print
916
917
918 # Next customer: Na;onal graphics company
919 # bbd = 25.1
920 # c = Circle(bbd_to_radius(bbd)
921 # print 'A circle with a bbd of 25.1'
922 # print 'has a radius of', c.radius
923 # print 'an an area of', c.area()
924 # print
925
926 # c = Circle(bbd_to_radius(bbd)) -------> this is Baaaad!

927 'USE Alternative Constructor'
928 print(dict.fromkeys(['name', 'age', 'language']))
929 #{'name': None, 'age': None, 'language': None}
930
931 # /lets go back and add the alternative constructor
932
933 import math
934
935
936 class Circle:
937
```

```
938     'An advanced circle analytic toolkit'
939     version = '0.3'
940 
941     def __init__(self, radius):
942         self.radius = radius
943 
944     def area(self):
945         return math.pi * self.radius ** 2.0
946 
947     def perimeter(self):
948         return 2.0 * math.pi * self.radius
949 
950     @classmethod
951     # classmethod make sure you use cls , for the subclass usage
952     def from_bbd(cls, bbd):
953         radius = bbd / 2.0 / math.sqrt(2.0)
954         # return Circle(radius) NONO!
955         # classmethod make sure you use cls , for the subclass
    usage
956         return cls(radius)
957 
958 
959 c = Circle.from_bbd(25.1)
960 # print 'A circle with a bbd of 25.1'
961 # print 'has a radius of', c.radius
962 # print 'an an area of', c.area()
963 # print
964 
965 # New customer request: add a func
966 # use staticmethod ,a giveaway is your func does not need 'self'
    or 'cls'. you use staticmethod for the findability of your func.
967 
968 
969 class Circle(object):
970     'An advanced circle analytic toolkit'
971     version = '0.4'
972 
973     def __init__(self, radius):
974         self.radius = radius
975 
976     @staticmethod
977     # attach functions to classes to increase the findability of
    your func.
978     # a giveaway is your func does not need 'self' or 'cls'.
979     def angle_to_grade(angle):
```

```
980         'Convert angle in degree to a percentage grade'
981         return math.tan(math.radians(angle)) * 100.0
982
983
984 # Government request: ISO-11110: "you need to use perimeter to
    calc the area" ,like this:
985
986 # class Circle(object):
987 #     'An advanced circle analytic toolkit'
988 #     version = '0.5b'
989 #     def __init__(self, radius):
990 #             self.radius = radius
991 #     def area(self):
992 #         p = self.perimeter()
993 #         r = p / math.pi / 2.0 return math.pi * r ** 2.0
994 #     def perimeter(self):
995 #         return 2.0 * math.pi * self.radius
996
997
998 # that wasnot too bad,really?
999 # the Tire subclass update the perimeter, now you broke their
    code.
1000
1001 # class Tire(Circle):
1002 #     'Tires are circles with an odometer corrected perimeter'
1003 # def perimeter(self):
1004 # 'Circumference corrected for the rubber' return
    Circle.perimeter(self) * 1.25
1005
1006
1007 '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.
1008 # the idea is to  use classname+methodname.
1009 # __perimeter---> Name mangling into--->
    '_(class.__name__)__perimeter'
1010
1011
1012 class Circle:
1013     def __init__(self, radius):
1014         self.radius = radius
1015
1016     def perimeter(self):
```

```
1017            return self.radius * 2 * math.pi
1018
1019        # make local refernce perimeter
1020        __perimeter = perimeter
1021
1022        # see Ned Batchelder - Facts and Myths about Python names
      and values - PyCon 2015
1023        # a=3
1024        # b=a
1025        # a=4
1026        # print(b)---->3
1027
1028        def area(self):
1029            p = self.__perimeter()
1030            r = p / (2 * math.pi)
1031            return math.pi * r**2
1032
1033
1034    # Government request: ISO-22220
1035    # •  You're not allowed to store the radius
1036    # •  You must store the diameter instead!
1037
1038    # we get to keep the api the same. still i accept radius in
      __init__, but diameter will be stored instead.
1039
1040
1041    # it breaks our entire class!
1042    #" I just wish everytime i use dot for look up,            it
      will magiclly trans into a get method call ()"
1043    #" I just wish everytime I set a radius(even in __init__) ,it
      will magiclly trasn in to s set radius call,--store the
      diameter."
1044    # 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"
1045    # property is for "after the fact , that you dont need to change
      any existing code.and add on the property"
1046
1047
1048    # User request: Many circles
1049    # n = 10000000
1050    # seed(8675309)
1051    # print 'Using Circuituous(tm) version', Circle.version
1052    # circles = [Circle(random()) for i in xrange(n)]
```

```
1053  # print 'The average area of', n, 'random circles'
1054  # avg = sum([c.area() for c in circles]) / n
1055  # print 'is %.1f' % avg
1056  # print
1057  # I sense a major memory problem.
1058  # Circle instances are over 300 bytes each!
1059
1060  'Flyweight design paUern: Slots'
1061  # save this for the last.you cant add new attr ,you cant access
      the dictinary no more.no vars() or .__dict__.
1062  # "from the user view, there are no changes at all"_R.H
1063  # dont worry ,subclass does not inherit the slots
1064
1065
1066  class Circle(object):
1067
1068      'An advanced circle analytic toolkit'
1069  # flyweight design pattern suppresses
1070  # the instance dictionary
1071      __slots__ = ['diameter']
1072      version = '0.7'
1073
1074      def __init__(self, radius):
1075
1076          self.radius = radius
1077
1078      @property  # convert dotted access to method calls
1079      def radius(self):
1080          return self.diameter / 2.0
1081
1082      @radius.setter
1083      def radius(self, radius):
1084          self.diameter = radius * 2.0
1085
1086
1087  """Summary: Toolset for New - Style Classes
1088  1.  Inherit from object().
1089  2.  Instance variables for informa
1090  on unique to an instance.
1091  3.  Class variables for data shared among all instances.
1092  4.  Regular methods need “self” to operate on instance data.
1093  5.  Thread local calls use the double underscore. Gives
      subclasses the freedom to override methods without breaking
      other methods.
1094  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
        self.last = last

    @classmethod
    def from_str(cls, info_string):
        return cls(*info_string.split(' '))

    def __repr__(self):
        return f'{self.__class__.__name__} {tuple(vars(self).values())}'


emp_1 = Employee.from_str('frank young')
print(emp_1)



'THANKS TO Corey Schafer,Ned Batchelder,Brett Slatkin,and The Great Raymond Hettinger'
```

```
1132
1133
1134
1135  # Other modules on the side:pytz, re(Regular expression),
      logging, bs4.BeatifulSoup, sqlite3, basic terminal operations.
```