05/12/18 03:35:01 /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  s='abcd'#--->['a', 'b', 'c', 'd']
25  try:
26      print(s.split(''))
27  except Exception as e:
28      print(e)#empty separator
29  # str.split('') wont work, just use list(str)
30
31  print('a'*3) #'aaa'
32  lst=['a',1,'b']
33  try:
34      print(''.join(lst))
35  except Exception as e:
```

```
36      print(e) #sequence item 1: expected str instance, NoneType
    found
37  # join is a str method, only works for string. not int or
    NoneType
38
39  print('a'[:4])#--->No Error ,gets 'a'
40  try:
41      print('a'[-4])
42  except Exception as e:
43      print(e) #sequence item 1: expected str instance, int found
44  print('a'[:-4]) #''empty str
45  print('a'[-1:]) #'a'
46
47  import csv
48  import contextlib
49  # 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.
50  with contextlib.suppress(Exception):
51      with open('Customer_Satisfaction.csv') as rf:
52          reader = csv.DictReader(rf)
53          print(reader.fieldnames)  # ['Year', 'Category',
    'Satisfaction Rating']
54          with open('Customer_Satisfaction_copy.csv', 'w') as wf:
55              fieldnames = ['Category', 'Year']
56              writer = csv.DictWriter(wf, fieldnames=fieldnames,
    delimiter='\t')
57              writer.writeheader()
58              for line in reader:
59                  del line['Satisfaction Rating']
60                  writer.writerow(line)
61
62
63  # Sort file into year_month folder
64  import os
65  import datetime
66  from contextlib import suppress
67  def year_month_folder(path):
68      os.chdir(path)
69      for file in os.listdir():
70          if os.path.isfile(file):
71
72              mtime = os.stat(file).st_mtime
73              mtime = datetime.date.fromtimestamp(mtime)
```

```
74              folder_name = f'{mtime:%y %B}'
75              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.
76                  os.mkdir(folder_name)
77
78              name_path = os.path.join(folder_name, file)
79              os.rename(file, name_path)
80
81
82
83  """object sorting"""
84  from operator import itemgetter, attrgetter, methodcaller
85
86  student_tuples = [('john', 'A', 15), ('jane', 'B', 12), ('dave',
    'B', 10)]
87  print(sorted(student_tuples, key=itemgetter(2)))
88  # [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
89  print(sorted(student_tuples, key=itemgetter(1, 2)))
90  # [('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
91
92  class Student:
93      def __init__(self, name, grade, age):
94          self.name = name
95          self.grade = grade
96          self.age = age
97
98      def __repr__(self):
99          return repr((self.name, self.grade, self.age))
100
101
102 student_objects = [Student('john', 'A', 15), Student('jane',
    'B', 12), Student('dave', 'B', 10), ]
103
104 print(sorted(student_objects, key=attrgetter('age')))
105 # [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
106 print(sorted(student_objects, key=attrgetter('grade', 'age')))
107 # [('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
108
109 messages = ['critical!!!', 'hurry!', 'standby', 'immediate!!']
110 print(sorted(messages, key=methodcaller('count', '!')))
111 # ['standby', 'hurry!', 'immediate!!', 'critical!!!']
```

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

```
154 # print(os.getcwd())
155
156
157 @contextlib.contextmanager
158 # by Nick Coghlan
159 def suppress(*exceptions):
160     try:
161         yield
162     except exceptions:
163         pass
164
165
166 # Decorator,Scope,Closure
167 # the LEGB rule , for accessing(say,print) and modifying(append)
    mutable var only.But not reasigning after referenced to a
    nonlocal(enclosing,Global) variable .
168 # UnboundLocalError: local variable referenced to  before
    assignment.
169
170 "the scope of a var is determined when function is defined , the
    value of a var is determined when the function is called"
171 # <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."
172
173
174 # https://nedbatchelder.com/text/names.html
175 # Ned Batchelder - Facts and Myths about Python names and values
    - PyCon 2015
176 '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'
177 # '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.
178
179 a = 1
180 def p():
```

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

```
217     @wraps(my_func)
218     def inner(*args, **kwargs):
219         inner.count += 1
220         return my_func(*args, **kwargs)
221     inner.count = 0
222     return inner
223 
224 
225 @counter
226 def i_tell_you_what():
227     return 'i tell you what'
228 
229 
230 i_tell_you_what()
231 i_tell_you_what()
232 print(f'{i_tell_you_what.__name__} run {i_tell_you_what.count}
    times')
233 # i_tell_you_what run 2 times
234 
235 # make a cache with a default return dictionary as an arg
236 
237 def cache_with_default(dct=None):
238     if dct is None:
239         dct = {}
240 
241     def cache(my_func):
242         @wraps(my_func)
243         def wrapper(*args):
244             if args in dct:
245                 return dct[args]
246             result = my_func(*args)
247             dct[args] = result
248             return result
249         return wrapper
250     return cache
251 
252 # 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
253 
254 @cache_with_default({(1,): 100})  # be careful, pass 1 as a
    tuple (1,) or it wont work.becaues the args will be (1,)
255 def times_two(x):
256     return x + x
```

```
257
258
259 print(times_two(1))  # get 100 inside of 2 , because it was
    looked up in the dct
260
261
262 # Brett Slatkin - How to Be More Effective with Functions -
    PyCon 2015 - YouTube
263 # http://www.informit.com/articles/article.aspx?p=2314818
264
265 # keword_only_args:forced to be clear.
266 # dont pass infinity generator into *args, like
    itertools.count().it will try to tuple(count()),and that will
    crash.
267 # To avoid this possibility entirely, you should use keyword-
    only arguments when you want to extend functions that accept
    *args
268 # *this is a kwargs_only function, so :'anything after a "*" or
    "*args"  is FORCED to be clear (keyword)'
269
270
271 def kwargs_only(*args, a=1): #this a is forced to be clear,
    because a is after a "*"
272     print(a)
273     print(args)
274
275
276 kwargs_only(2, 2, 2, 2, a=4)
277 # 4
278 # (2, 2, 2, 2)
279
280 def bobby(*, propane=True, charcoal=False):
281     # it nice to be clear, when your args are the same type
    data.
282     if propane:
283         print('I sell propane and propane accessories')
284     else:
285         print('the hell you say')
286
287
288 try:
289     bobby(True, False)
290 except Exception as e:
291     print(e)  # bobby() takes 0 positional arguments but 2 were
```

```
    given
292
293 bobby(propane=True, charcoal=False)  # I sell propane and
    propane accessories
294 #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'
295 # use namedtuple as return tuple for clarity
296 from collections import namedtuple
297 def
    twitter_search(name,*,retweets=True,numtweets=0,popluar=False):
298     twsearch=namedtuple('twsearch',
    ['name','retweets','numtweets','popluar'])
299     result=twsearch(name,retweets,numtweets,popluar)
300     return result
301 obama=twitter_search('obama',retweets=False,numtweets=10,popluar
    =True)
302 print(obama)#twsearch(name='obama', retweets=False,
    numtweets=10, popluar=True)
303
304
305
306 # what is generator?
307 # iter(foo) is iter(foo)
308 # base on the talk-->Brett Slatkin - How to Be More Effective
    with Functions
309 # if iter(foo) is iter(foo):
310 #       now,then =itertools.tee(foo,2)
311 # 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:
312 # compare this info_get function and Info_Gen Class:
313 # difference is ever time you call the 'for' , __iter__method on
    class, it return a new iterator over a container.
314 # so far I prefer itertools.tee ,it is easier.
315 def info_gen(path):
316     with open(path) as f:
317         reader = csv.DictReader(f)
318         for line in reader:
319             del line['Year']
320             yield line
321
322
```

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

355 
```

```
356
357 # how does generator function(yield) run?
358
359 import contextlib
360
361
362 def HYW():
363     print('hello')
364     yield
365     print('world')
366
367
368 a = HYW()  # Nothing happend, ! hello was not printed.
369 next(a)  # ---> now hello was printed. so when you call next,
    generator will run till it hits a yield
370
371 with contextlib.suppress(StopIteration):
372     next(a)  # -----> world was printed, and then it hits the
    StopIteration
373
374
375
376
377
378 #  Ned Batchelder - Facts and Myths about Python names and
    values - PyCon 2015
379 # "reassign one of the name ,brother,doesnt reassign the other"
    ---Ned
380
381 a = [1, 2, 3]
382 b = a
383 a += [4, 5]  # what happened here unline is "a.extend([4,5]) and
    a =a "
384 print(b)  # -->[1, 2, 3, 4, 5]
385
386 a = [1, 2, 3]
387 b = a
388 a = a + [4, 5]
389 print(b)  # -->[1, 2, 3]
390
391 a = [1, 2, 3]
392 b = a
393 a.extend([4, 5])
394 print(b)  # -->[1, 2, 3, 4, 5]
```

```
395
396 a = [1, 2, 3]
397 b = a
398 a = a.extend([4, 5])
399 print(a)  # None
400 print(b)  # [1, 2, 3, 4, 5]
401
402 a = [1, 2, 3]  # try to make a =[10,20,30]
403 for x in a:
404     x = x * 10
405 print(a)  # [1, 2, 3] failed:) beacuse a[0]still is 1 . the
    right way ,a=[x*10 for x in a]
406
407 nums = [1, 2, 3]
408 print(nums.__iadd__([4, 5]))  # [1, 2, 3, 4, 5],inplace and
    return the new value
409 print(nums) #[1, 2, 3, 4, 5]
410 print(nums.extend([7, 8]))  #print None. inplace but no return
    value,so print None
411 print(nums) #[1, 2, 3, 4, 5, 7, 8]
412
413
414
415
416 nums = [1, 2, 3]
417
418 def modify():
419     print(nums)
420     nums.append(4)
421
422 modify()  # [1, 2, 3, 1]
423
424 def re_assign():
425     print(nums)
426     nums += [5]
427     # num=list.__iadd__(nums,[5]) this is modify first then
    re_assign,wont work for the assign part.
428
429 try:
430     re_assign()
431 except Exception as e:
432
433     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
434
435
436 # "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.
437 # @nedbatchelder.com
438 # Let's examine the most interesting of these alternate
    assignments: calling a function. When I define a function, I
    name its parameters:
439 # def my_func(x, y):
440 #     return x+y
441 # 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:
442 # 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
443
444 # https://nedbatchelder.com/text/names.html
445 def a_func(num):
446     num = num + 2
447
448
449 num = 2
450 num = a_func(num)
451
452 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.
453
454
455 # ITERATION
456 # a trick  zip(*[iter(s)]*n)
457 lst = range(10)#[0,1,2,3,4,5,6,7,8,9]
458 print(iter(lst) is iter(lst))  # False
```

```
459 print(list(zip(*[iter(lst)] * 3)))  # [(0, 1, 2), (3, 4, 5), (6,
    7, 8)]
460 import itertools
461 print(list(itertools.zip_longest(*[iter(lst)]*3))) #[(0, 1, 2),
    (3, 4, 5), (6, 7, 8), (9, None, None)]
462 # https://stackoverflow.com/questions/2233204/how-does-
    zipitersn-work-in-python
463 # zip(*lst) is funny
464 # [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),
465 # but map is a iterator.see below:
466 # https://stackoverflow.com/questions/16425166/accumulate-items-
    in-a-list-of-tuples
467
468 # try to make lst = [(0, 0), (2, 3), (4, 3), (5, 1)] into
    new_lst = [(0, 0), (2, 3), (6, 6), (11, 7)]
469 lst = [(0, 0), (2, 3), (4, 3), (5, 1)]
470
471 import itertools
472 new_lst = zip(*lst)  # zip_object contains ((0,2,4,5),(0,3,3,1))
473 new_lst = map(itertools.accumulate, new_lst)  # map_object
    contains ((0,2,6,11),(0,3,6,7))
474 # print(iter(new_lst) is iter(new_lst)) #True ,so map is a
    iterator
475 new_lst = list(zip(*new_lst))
476 print(new_lst)  # [(0, 0), (2, 3), (6, 6), (11, 7)]
477 # so all in one line:
    list(zip(*map(itertools.accumulate,zip(*lst))))
478
479 # itertools
480 # islice doesn't  consume the original iterator until next is
    called. most(all) itertools are like that.
481
482 # from itertools doc
483 from collections import deque
484
485
486 def consume(iterator, n=None):
487     "Advance the iterator n-steps ahead. If n is None, consume
    entirely."
488     if n is None:
489         deque(iterable, maxlen=0)
```

```
490     else:
491         # itertools.islice(iterator,n,n) # THAT IS A NONO!!! islice doesn't  consume the original iterator  until Next is called!!!!!
492         next(itertools.islice(iterator, n, n), None)  # YES
493
494
495 def tail(n, iterable):
496     "Return an iterator over the last n items"
497     # tail(3, 'ABCDEFG') --> E F G
498     return iter(collections.deque(iterable, maxlen=n))
499
500
501 # cycle+compress, wanted a serial condition ,say one False and 20 True, forever
502 iterable = range(45)
503 result = itertools.compress(iterable, itertools.cycle(range(21)))  # 1-20,22-41,43,44
504
505 # itertools.repeat take container, not iterator. won't work.use repeat(tuple(iterator)).while cycle takes iterators.
506
507
508 # @accumulate usage: turn [1,2,3] in to int 123. or reduce
509 lst = [1, 2, 3]
510 result = itertools.accumulate(lst, lambda a, b: 10 * a + b)
511 print(list(result))  # [1, 12, 123]
512 # or use reduce , it is actually better
513 from functools import reduce
514 result = reduce(lambda a, b: 10 * a + b, lst)
515 print(result)  # 123
516
517 # 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'
518 # get all the fib nums  < 40,000
519 # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657]
520
521
522 def fib():
523     a, b = 0, 1
524     #unpacking seqiences,high level of thinking._R.H
```

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

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

587 # print(result)
588 # defaultdict(<class 'list'>, {'KAT': ['11013331', '9843236'],
    'NOT': ['9085267', '11788544'], 'ETH': ['5238761', '5349618',
    '962142', '7795297', '7341464', '5594916', '1550003']})
589 result=[{'type': key, 'items':items} for key,items in
```

```
    result.items()]
590 result = json.dumps(result, indent=2)
591 print(result)#works too.
592
593 # another funny thing about groupby
594 # 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}]
595 lst=[1,1,1,1,1,3,3,4,2,2,1,1,1,3,3] #without sorting
596 result=itertools.groupby(lst)
597 # for key,items in result:
598 #     print(key,'--->',list(items))
599 # 1 ---> [1, 1, 1, 1, 1]
600 # 3 ---> [3, 3]
601 # 4 ---> [4]
602 # 2 ---> [2, 2]
603 # 1 ---> [1, 1, 1]
604 # 3 ---> [3, 3]
605 result=[{key:len(list(items))}for key ,items in result]
606 #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}
607
608 print(result) #[{1: 5}, {3: 2}, {4: 1}, {2: 2}, {1: 3}, {3: 2}]
609
610 from collections import Counter
611 lst=[1,1,1,1,1,3,3,4,2,2,1,1,1,3,3]
612 result=Counter(lst)
613 print(result) #Counter({1: 8, 3: 4, 2: 2, 4: 1})
614 print(result.most_common(2))#[(1, 8), (3, 4)]
615
616
617
618 #the Great Raymond Hettinger's  Section
619 #Transforming Code into Beautiful, Idiomatic Python + Python
    Class Toolkit
620 # iter(callable_func,sentinel_value)
621 # blocks=[]
622 # for block in iter(functools.partial(f.read,32),''):
623 #     blocks.appem=nd(block)
624 #
625 # for loop ,else:no break
626 dct= {'matthew': 'blue', 'rachel': 'green', 'raymond': 'red'}
627 # 'for key in dct' vs 'for key in list(dct)' --->when you are
    mutating the dictionary.
628
```

```
629 # "if you mutating something while you iter over it, you are
    living in the state of sin, and you deserve whatever happens to
    you"
630 # 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]
631
632 try :
633     for k in dct:
634         if k.startswith('r'):
635             del dct[k]
636 except Exception as e:
637     print(e) #dictionary changed size during iteration
638
639
640 for key in list(dct):
641     if key.startswith('r'):
642         del dct[key]
643 print(dct) # {'matthew': 'blue'} --->works
644
645 dct= {'matthew': 'blue', 'rachel': 'green', 'raymond': 'red'}
646 while dct:
647     key,value = dct.popitem()
648     print(f'I just popped {key}---->{value}')
649 # I just popped raymond---->red
650 # I just popped rachel---->green
651 # I just popped matthew---->blue
652
653
654 # defaultdict for counting (collections.Counter),
    grouping(itertools.groupby
655
656
657
658 colors = ['red', 'green', 'red', 'blue', 'green', 'red']
659
660 # defaultdict
661 from collections import defaultdict
662 result=defaultdict(int)

663 for color in colors:
664     result[color]+=1
665 print(result) #defaultdict(<class 'int'>, {'red': 3, 'green': 2,
    'blue': 1})
666
667 # use Counter
```

```
668 from collections import Counter
669 result=Counter(colors)
670 print(result)#Counter({'red': 3, 'green': 2, 'blue': 1})
671
672 # use nothing(get)
673 result={}
674 for color in colors :
675     result[color]=result.get(color,0)+1
676 print(result) #{'red': 3, 'green': 2, 'blue': 1}
677
678 #group
679
680 #defaultdict
681 names = ['raymond', 'rachel', 'matthew', 'roger','betty',
    'melissa', 'judith', 'charlie']
682 result_1=defaultdict(list)
683 for name in names:
684     key=len(name)
685     result_1[key].append(name)
686 print(result_1)
687 #defaultdict(<class 'list'>, {7: ['raymond', 'matthew',
    'melissa', 'charlie'], 6: ['rachel', 'judith'], 5: ['roger',
    'betty']})
688
689 #use get
690 result_2={}
691 for name in names:
692     key=len(name)
693     result_2[key]=result_2.get(key,[])+[name]#be careful, don't
    use append, becauese it returns nothing,result_2[key] will be
    None
694 print(result_2)
695 # {7: ['raymond', 'matthew', 'melissa', 'charlie'], 6:
    ['rachel', 'judith'], 5: ['roger', 'betty']}
696
697
698 #groupby
699 import itertools
700 result_3=sorted(names,key=len)
701 result_3=itertools.groupby(result_3,key=len)
702 # for key,names in result_3:
703 #     print(key,'--->',list(names))
704 # 5 ---> ['roger', 'betty']
705 # 6 ---> ['rachel', 'judith']
```

```
706 # 7 ---> ['raymond', 'matthew', 'melissa', 'charlie']
707 result_3={key:list(names) for key,names in result_3}
708 print(result_3)
709 # {5: ['roger', 'betty'], 6: ['rachel', 'judith'], 7:
    ['raymond', 'matthew', 'melissa', 'charlie']}
710
711 "Linking dictionaries" 'ChainMap'
712 defaults = {'color': 'red', 'user': 'guest'}
713 envir={'user':'frank','login':'Unknown'}
714 command={'login':True}
715 from collections import ChainMap
716 result=ChainMap(command,envir,defaults) #high to low
717 print(result['color'])#red
718 print(result['login'])#True
719 print(result['user'])#frank
720
721
722
723
724 from functools import wraps
725 # famous cache decorator
726 def cache(my_func):
727     saved={}
728     @wraps(my_func)
729     def wrapper(*args):
730         if args in saved:
731             print('returned from saved')
732             return saved[args]
733         result=my_func(*args)
734         saved[args]=result
735         print('return from func(*args)')
736         return result
737     return wrapper
738
739 @cache
740 def printer(a):
741     print(a.upper())

742 printer('a')
743 # A
744 # return from func(*args)
745 printer('a')
746 # returned from saved
747
748 # this is realy a bad example,because second time 'A' was not
```

```
    printed.so it doesn't not work for all functions.
749
750 @cache
751 def rt(a):
752     return a.upper()
753 print(rt('a'))
754 # return from func(*args)
755 # A
756 print(rt('a'))
757 # returned from saved
758 # A
759 # works good this time:)
760
761
762
763 #the ignored (suppress) contextmanager
764 @contextlib.contextmanager
765 def ignored(*exceptions):
766     try:
767         yield
768     except exceptions:
769         pass
770
771 # Codingbat
772
773 # http://codingbat.com/prob/p118406
774 # 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.
775 def make_bricks(small, big, goal):
776   return small+5*big>=goal and (goal-small)//5<=big and
    goal%5<=small
777
778 # http://codingbat.com/prob/p167025
779 # 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.
780 def sum13(nums):
781     nums=nums+[0]#important
782     result=[num for index,num in enumerate(nums) if not num==13
    and nums[index-1]!=13]
783     return sum(result)
```

```
784
785 # http://codingbat.com/prob/p186048
786 # 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.
787
788 def count_code(str):
789   str=str+' ' #important !'eaacow'
790   result=[x for index,x in enumerate(str) if x=='e' and
    str[index-2]=='o' and str[index-3]=='c' ]
791   return len(result)
792
793
794 # 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.
795 # xyz_there('abcxyz') → True
796 # xyz_there('abc.xyz') → False
797 # xyz_there('xyz.abc') → True
798
799 # the logic of this problem is the highlight
800 def xyz_there(str):
801   str=str.replace('.xyz','wwww')# important can not do
    replace('.xyz','').
802   return 'xyz' in str
803
804
805 '~~~~~~~'
806 "Python's Class Development Toolkit _ Raymond Hettinger"
807 #"Python is consenting as an adult language. We don't leave the
    locks on the door." _ Raymond Hettinger
808
809 """
810 Circles, Inc.
811 """
812
813

814 class Circle:  # python 3 is automatically a new style class.
    2.7 needs to inherit (object)
815     from math import pi
816     """An advvanced circle analytics toolkit"""
817     # don/t skip the elevator pitch ,your doc string.
818     # what is inside a class is effectlly a module ,it is like
```

```
    the code run in its own module.
819
820     print('i am defining a class')  # it will print only by
    defining it.
821     # raymond also talked about you can open file or for loop
    with in the class.
822
823     version = '0.1'  # class variable for shared data,while
    instance var for unique data. use str, or tuple
824     print('dont use bi_floats , try:0.1+0.7,you will get ', 0.1
    + 0.7)  # 0.7999999999999999
825
826     def __init__(self, radius):
827         # "__init__ " is not a constructor. is calling the class
    construct a instance.__init__ is 'poplulate' instance variable.
828         # one thing is for sure, user is gonna make lots of
    instance, i mean a lot .
829         print('i am running __init__')
830         self.radius = radius
831
832     def area(self):
833         return self.radius**2 * pi
834     # so far we are good to go, more method ? until user ask for
    it! before that,YAGNI:) Lean startup.
835
836
837 # First customer: Academia
838 # from random import random, seed
839 # seed(8675309)
840 # print 'Using Circuituous(tm) version', Circle.version
841 # n = 10
842 # circles = [Circle(random()) for i in xrange(n)]
843 # print 'The average area of', n, 'random circles'
844 # avg = sum([c.area() for c in circles]) / n
845 # print 'is %.1f' % avg
846 # print
847
848     def perimeter(self):
849         # new customer wants a perimeter method.
850         return self.radius * 2 * pi
851
852 # Second customer: Rubber sheet company
853 # cuts = [0.1, 0.7, 0.8]
854 # circles = [Circle(r) for r in cuts]
```

```
# for c in circles:
#     print 'A circlet with with a radius of', c.radius
#     print 'has a perimeter of', c.perimeter()
#     print 'and a cold area of', c.area()
#     c.radius *= 1.1
#     print 'and a warm area of', c.area()
#     print


# this customer changed the attribute "c.radius *= 1.1"
"if it is a variable, it is gonna change, sooner or later"  # R.H

# If you expose an attribute, expect users to all kinds of interesting things with it.


# 3rd customer Tire
class Tire(Circle):
    'Tires are circles with a corrected perimeter'
    # again
    "if it is a variable, it is gonna change, sooner or later"
# R.H


def perimeter(self):
    'Circumference corrected for the rubber'
    return Circle.perimeter(self) * 1.25


# t = Tire(22)
# print 'A tire of radius', t.radius
# print 'has an inner area of', t.area()
# print 'and an odometer corrected perimeter of',
# print t.perimeter()
# print


# Next customer: Na;onal graphics company
# bbd = 25.1
# c = Circle(bbd_to_radius(bbd)
# print 'A circle with a bbd of 25.1'
# print 'has a radius of', c.radius
# print 'an an area of', c.area()
```

```
# print

# c = Circle(bbd_to_radius(bbd)) -------> this is Baaaad!
'USE Alternative Constructor'
print(dict.fromkeys(['name', 'age', 'language']))
#{'name': None, 'age': None, 'language': None}

# /lets go back and add the alternative constructor

import math


class Circle:

    'An advanced circle analytic toolkit'
    version = '0.3'

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2.0

    def perimeter(self):
        return 2.0 * math.pi * self.radius

    @classmethod
    # classmethod make sure you use cls , for the subclass usage
    def from_bbd(cls, bbd):
        radius = bbd / 2.0 / math.sqrt(2.0)
        # return Circle(radius) NONO!
        # classmethod make sure you use cls , for the subclass usage
        return cls(radius)


c = Circle.from_bbd(25.1)

# print 'A circle with a bbd of 25.1'
# print 'has a radius of', c.radius
# print 'an an area of', c.area()
# print

# New customer request: add a func
# use staticmethod ,a giveaway is your func does not need 'self'
```

```
    or 'cls'. you use staticmethod for the findability of your func.
939
940
941 class Circle(object):
942     'An advanced circle analytic toolkit'
943     version = '0.4'
944
945     def __init__(self, radius):
946         self.radius = radius
947
948     @staticmethod
949     # attach functions to classes to increase the findability of
    your func.
950     # a giveaway is your func does not need 'self' or 'cls'.
951     def angle_to_grade(angle):
952         'Convert angle in degree to a percentage grade'
953         return math.tan(math.radians(angle)) * 100.0
954
955
956 # Government request: ISO-11110: "you need to use perimeter to
    calc the area" ,like this:
957
958 # class Circle(object):
959 #     'An advanced circle analytic toolkit'
960 #     version = '0.5b'
961 #     def __init__(self, radius):
962 #             self.radius = radius
963 #     def area(self):
964 #         p = self.perimeter()
965 #         r = p / math.pi / 2.0 return math.pi * r ** 2.0
966 #     def perimeter(self):
967 #         return 2.0 * math.pi * self.radius
968
969
970 # that wasnot too bad,really?
971 # the Tire subclass update the perimeter, now you broke their
    code.
972
973 # class Tire(Circle):
974 #     'Tires are circles with an odometer corrected perimeter'
975 # def perimeter(self):
976 # 'Circumference corrected for the rubber' return
    Circle.perimeter(self) * 1.25
977
978
```

```
979 '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.
980 # the idea is to  use classname+methodname.
981 # __perimeter---> Name mangling into--->
    '_(class.__name__)__perimeter'
982
983
984 class Circle:
985     def __init__(self, radius):
986         self.radius = radius
987
988     def perimeter(self):
989         return self.radius * 2 * math.pi
990
991     # make local refernce perimeter
992     __perimeter = perimeter
993
994     # see Ned Batchelder - Facts and Myths about Python names
    and values - PyCon 2015
995     # a=3
996     # b=a
997     # a=4
998     # print(b)---->3
999
1000    def area(self):
1001        p = self.__perimeter()
1002        r = p / (2 * math.pi)
1003        return math.pi * r**2
1004
1005
1006 # Government request: ISO-22220
1007 # •  You're not allowed to store the radius
1008 # •  You must store the diameter instead!

1009
1010 # we get to keep the api the same. still i accept radius in
    __init__, but diameter will be stored instead.
1011
1012
1013 # it breaks our entire class!
1014 #" I just wish everytime i use dot for look up,           it
    will magicly trans into a get method call ()"
1015 #" I just wish everytime I set a radius(even in __init__) ,it
```

```
      will magicly trasn in to s set radius call,--store the
      diameter."
1016  # 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"
1017  # property is for "after the fact , that you dont need to change
      any existing code.and add on the property"
1018
1019
1020  # User request: Many circles
1021  # n = 10000000
1022  # seed(8675309)
1023  # print 'Using Circuituous(tm) version', Circle.version
1024  # circles = [Circle(random()) for i in xrange(n)]
1025  # print 'The average area of', n, 'random circles'
1026  # avg = sum([c.area() for c in circles]) / n
1027  # print 'is %.1f' % avg
1028  # print
1029  # I sense a major memory problem.
1030  # Circle instances are over 300 bytes each!
1031
1032  'Flyweight design paUern: Slots'
1033  # save this for the last.you cant add new attr ,you cant access
      the dictinary no more.no vars() or .__dict__.
1034  # "from the user view, there are no changes at all"_R.H
1035  # dont worry ,subclass does not inherit the slots
1036
1037
1038  class Circle(object):
1039
1040      'An advanced circle analytic toolkit'
1041  # flyweight design pattern suppresses
1042  # the instance dictionary
1043      __slots__ = ['diameter']
1044      version = '0.7'
1045
1046      def __init__(self, radius):
1047
1048          self.radius = radius
1049
1050      @property  # convert dotted access to method calls
1051      def radius(self):
1052          return self.diameter / 2.0
1053
```

```
1054      @radius.setter
1055      def radius(self, radius):
1056          self.diameter = radius * 2.0
1057
1058
1059  """Summary: Toolset for New - Style Classes
1060  1.  Inherit from object().
1061  2.  Instance variables for informa
1062  on unique to an instance.
1063  3.  Class variables for data shared among all instances.
1064  4.  Regular methods need “self” to operate on instance data.
1065  5.  Thread local calls use the double underscore. Gives
      subclasses the freedom to override methods without breaking
      other methods.
1066  6.  Class methods implement alterna
1067  ve constructors. They need “cls” so they can create subclass
      instances as well.
1068  7.  Sta
1069  c methods aUach func
1070  ons to classes. They don’t need either “self” or “cls”. Sta
1071  c methods improve discoverability and require context to be
      specified.
1072  8.  A property() lets geUer and seUer methods be invoked automa
1073  cally by aUribute access. This allows Python classes to freely
      expose their instance variables.
1074  9.  The “__slots__” variable implements the Flyweight Design
      PaUern by suppressing instance dic
1075  onaries."""
1076
1077
1078  # from until_mar_29.py
1079  # classmethod always use cls for subclass
1080  #__repr__ usage
1081  #__from_string__usage, if string contain classname then:
      classname,*info=info_string.split(' ')
1082
1083
1084  class Employee:
1085
1086      def __init__(self, first, last):
1087          self.first = first
1088          self.last = last
1089
1090      @classmethod
1091      def from_str(cls, info_string):
```

```
1092            return cls(*info_string.split(' '))
1093
1094        def __repr__(self):
1095            return f'{self.__class__.__name__}
    {tuple(vars(self).values())}'
1096
1097
1098    emp_1 = Employee.from_str('frank young')
1099    print(emp_1)
1100
1101
1102
1103    'THANKS TO Corey Schafer,Ned Batchelder,Brett Slatkin,and The
    Great Raymond Hettinger'
```