

公告

最新自学视频 路飞Python免费小课

人生迷茫问题可以加Alex个人微信听鸡



Alex金角大王



扫一扫上面的二维码图案，加我微信

汤

网管到CEO的10年逆袭之路

Alex给你清晰的事业规划和执行策略

Alex

老男孩Python教学总监 | 路飞学城CEO



面向对象编程

Alex | 前汽车之家架构师



面向对象开发原来如此简单

16人在学 进阶 12小时

Python常用模块

Alex | 前汽车之家架构师



Python开发中最常用的11个模块精讲

10人在学 进阶 6小时

Python开发21天

Alex | 前汽车之家架构师



跟随Alex金角大王3周上手Python开发

124人在学 入门 19小时

昵称： 金角大王

园龄： 5年5个月

粉丝： 10868

关注： 5

+加关注

随笔 - 29 文章 - 64 评论 - 928

Python之路,Day4 - Python基础4 (new版)

本节内容

- 1. 迭代器&生成器
- 2. 装饰器
- 3. Json & pickle 数据序列化
- 4. 软件目录结构规范
- 5. 作业:ATM项目开发

1.列表生成式、迭代器&生成器

列表生成式

孩子，我现在有个需求，看列表[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],我要求你把列表里的每个值加1，你怎么实现？你可能会想到2种方式

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
241
```

我的标签

职业发展(3)

创业(2)

随笔分类

□职业&生活随笔(22)

文章分类

Python全栈开发之路(12)

Python学习目录(4)

Python自动化开发之路(33)

爬虫(6)

最新评论

1. Re:编程要自学或报班这事你都想不明白, 那必然是你智商不够
大王性格直爽, 有目标, 肯付出行动! 偶像! 我要是早几年遇见大王就好了, 现在都已经三十了, 浪费了大好的青春!

--Xiyue666

2. Re:python 之路, Day11 - python mysql and ORM
ALTER mytable ADD INDEX
[indexName] ON (username(length))
这句应改为: alter table mytable add index index...

--原竹

3. Re:Python之路,Day3 - Python基础3
@我的恋人叫臭臭 淫角大王听说很厉害吧? ...

--Xiyue666

4. Re:Python之路,Day1 - Python基础1
哎 我为什么不早点遇到老男孩 遇到alex 老师呢!

--Xiyue666

5. Re:Python 之路 Day5 - 常用模块学习
#思路: 过滤最里面的 () 将里面的数值计算后替换到原有公示字符串, 直到将所有的括号剔除后再计算没有括号的字符串。
import re def mul_div(num_str): #先算乘除, (num_s...

--编程届的小学生

阅读排行榜

1. python 之路, 200行Python代码写了个打飞机游戏! (52279)
2. Django + Uwsgi + Nginx 实现生产环境部署(31745)
3. Python Select 解析(27141)
4. 为什么很多IT公司不喜欢进过培训机构的人呢? (20503)
5. 编程要自学或报班这事你都想不明白, 那必然是你智商不够(16902)

推荐排行榜

1. 给一位做技术迷茫的同学回信(63)

所以, 如果列表元素可以按照某种算法推算出来, 那我们是否可以在循环的过程中不断推算出后续的元素呢? 这样就不必创建完整的list, 从而节省大量的空间。在Python中, 这种一边循环一边计算的机制, 称为生成器: generator。

要创建一个generator, 有很多种方法。第一种方法很简单, 只要把一个列表生成式的[]改成(), 就创建了一个generator:

```
1 >>> L = [x * x for x in range(10)]
2 >>> L
3 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
4 >>> g = (x * x for x in range(10))
5 >>> g
6 <generator object <genexpr> at 0x1022ef630>
```

创建L和g的区别仅在于最外层的[]和(), L是一个list, 而g是一个generator。

我们可以直接打印出list的每一个元素, 但我们怎么打印出generator的每一个元素呢?

如果要一个一个打印出来, 可以通过next()函数获得generator的下一个返回值:

```
1 >>> next(g)
2 0
3 >>> next(g)
4 1
5 >>> next(g)
6 4
7 >>> next(g)
8 9
9 >>> next(g)
10 16
11 >>> next(g)
12 25
13 >>> next(g)
14 36
15 >>> next(g)
16 49
17 >>> next(g)
18 64
19 >>> next(g)
20 81
21 >>> next(g)
22 Traceback (most recent call last):
23   File "<stdin>", line 1, in <module>
24 StopIteration
```

我们讲过, generator保存的是算法, 每次调用next(g), 就计算出g的下一个元素的值, 直到计算到最后一个元素, 没有更多的元素时, 抛出StopIteration的错误。

当然, 上面这种不断调用next(g)实在是太变态了, 正确的方法是使用for循环, 因为generator也是可迭代对象:

```
1 >>> g = (x * x for x in range(10))
2 >>> for n in g:
3 ...     print(n)
4 ...
5 0
6 1
7 4
8 9
9 16
10 25
11 36
12 49
13 64
14 81
```

2. 你做了哪些事，导致老板下调了对你的评价？(51)
3. 关于认识、格局、多维度发展的感触(46)
4. 为什么很多IT公司不喜欢进过培训机构的人呢？(37)
5. 编程要自学或报班这事你都想不明白，那必然是你智商不够(35)

所以，我们创建了一个generator后，基本上永远不会调用`next()`，而是通过for循环来迭代它，并且不需要关心`StopIteration`的错误。

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的for循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
1 def fib(max):
2     n, a, b = 0, 0, 1
3     while n < max:
4         print(b)
5         a, b = b, a + b
6         n = n + 1
7     return 'done'
```

注意，赋值语句：

```
1 a, b = b, a + b
```

相当于：

```
1 t = (b, a + b) # t是一个tuple
2 a = t[0]
3 b = t[1]
```

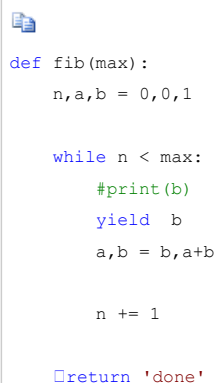
但不必显式写出临时变量t就可以赋值。

上面的函数可以输出斐波那契数列的前N个数：

```
1 >>> fib(10)
2 1
3 1
4 2
5 3
6 5
7 8
8 13
9 21
10 34
11 55
12 done
```

仔细观察，可以看出，`fib`函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把`fib`函数变成generator，只需要把`print(b)`改为`yield b`就可以了：



```
def fib(max):
    n, a, b = 0, 0, 1

    while n < max:
        #print(b)
        yield b
        a, b = b, a+b

        n += 1

    return 'done'
```

这就是定义generator的另一种方法。如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator：

```
>>> f = fib(6)
>>> f
<generator object fib at 0x104feaaa0>
```

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。



```
data = fib(10)
print(data)


print(data.__next__())
print(data.__next__())
print("干点别的事")
print(data.__next__())
print(data.__next__())
print(data.__next__())
print(data.__next__())
print(data.__next__())

#输出
<generator object fib at 0x101be02b0>
1
1
干点别的事
2
3
5
8
13
```




在上面fib的例子，我们在循环过程中不断调用yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用next()来获取下一个返回值，而是直接使用for循环来迭代：



```
>>> for n in fib(6):
...     print(n)
...
1
1
2
3
5
8
```



但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

```
1 >>> g = fib(6)
2 >>> while True:
3 ...     try:
4 ...         x = next(g)
5 ...         print('g:', x)
```

```

6     ...     except StopIteration as e:
7         ...         print('Generator return value:', e.value)
8         ...         break
9     ...
10    g: 1
11    g: 1
12    g: 2
13    g: 3
14    g: 5
15    g: 8
16    Generator return value: done

```

关于如何捕获错误，后面的错误处理还会详细讲解。

还可通过yield实现在单线程的情况下实现并发运算的效果

通过生成器实现协程并行运算

迭代器

我们已经知道，可以直接作用于for循环的数据类型有以下几种：

一类是集合数据类型，如list、tuple、dict、set、str等；

一类是generator，包括生成器和带yield的generator function。

这些可以直接作用于for循环的对象统称为可迭代对象：Iterable。

可以使用isinstance()判断一个对象是否是Iterable对象：

```

1  >>> from collections import Iterable
2  >>> isinstance([], Iterable)
3  True
4  >>> isinstance({}, Iterable)
5  True
6  >>> isinstance('abc', Iterable)
7  True
8  >>> isinstance((x for x in range(10)), Iterable)
9  True
10 >>> isinstance(100, Iterable)
11 False

```

而生成器不但可以作用于for循环，还可以被next()函数不断调用并返回下一个值，直到最后抛出StopIteration错误表示无法继续返回下一个值了。

***可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。**

可以使用isinstance()判断一个对象是否是Iterator对象：

```

1  >>> from collections import Iterator
2  >>> isinstance((x for x in range(10)), Iterator)
3  True
4  >>> isinstance([], Iterator)
5  False
6  >>> isinstance({}, Iterator)
7  False
8  >>> isinstance('abc', Iterator)
9  False

```

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

```

1  >>> isinstance(iter([]), Iterator)
2  True
3  >>> isinstance(iter('abc'), Iterator)
4  True

```

你可能会问,为什么list、dict、str等数据类型不是Iterator?

这是因为Python的Iterator对象表示的是一个数据流,Iterator对象可以被next()函数调用并不断返回下一个数据,直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列,但我们却不能提前知道序列的长度,只能不断通过next()函数实现按需计算下一个数据,所以Iterator的计算是惰性的,只有在需要返回下一个数据时它才会计算。

Iterator甚至可以表示一个无限大的数据流,例如全体自然数。而使用list是永远不可能存储全体自然数的。

小结

凡是可作用于for循环的对象都是Iterable类型;

凡是可作用于next()函数的对象都是Iterator类型,它们表示一个惰性计算的序列;

集合数据类型如list、dict、str等是Iterable但不是Iterator,不过可以通过iter()函数获得一个Iterator对象。

Python的for循环本质上就是通过不断调用next()函数实现的,例如:

```
1 | for x in [1, 2, 3, 4, 5]:
2 |     pass
```

实际上完全等价于:

```
# 首先获得Iterator对象:
it = iter([1, 2, 3, 4, 5])
# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
    except StopIteration:
        # 遇到StopIteration就退出循环
        break
```

2.装饰器

你是一家视频网站的后端开发工程师,你们网站有以下几个版块

```
1 | def home():
2 |     print("---首页----")
3 |
4 | def america():
5 |     print("----欧美专区----")
6 |
7 | def japan():
8 |     print("----日韩专区----")
9 |
10 | def henan():
11 |     print("----河南专区----")
```

视频刚上线初期,为了吸引用户,你们采取了免费政策,所有视频免费观看,迅速吸引了一大批用户,免费一段时间后,每天巨大的带宽费用公司承受不了了,所以准备对比较受欢迎的几个版块收费,其中包括“欧美”和“河南”专区,你拿到这个需求后,想了想,想收费得先让其进行用户认证,认证通过后,再判定这个用户是否是VIP付费会员就可以了,是VIP就让看,不是VIP就不让看就行了呗。你觉得这个需求很是简单,因为要对多个版

块进行认证，那应该把认证功能提取出来单独写个模块，然后每个版块里调用 就可以了，与是你轻轻的就实现了下面的功能。

```

1  #_*_coding:utf-8_*_
2
3
4  user_status = False #用户登录了就把这个改成True
5
6  def login():
7      _username = "alex" #假装这是DB里存的用户信息
8      _password = "abc!23" #假装这是DB里存的用户信息
9      global user_status
10
11     if user_status == False:
12         username = input("user:")
13         password = input("password:")
14
15         if username == _username and password == _password:
16             print("welcome login...")
17             user_status = True
18         else:
19             print("wrong username or password!")
20     else:
21         print("用户已登录，验证通过...")
22
23 def home():
24     print("---首页---")
25
26 def america():
27     login() #执行前加上验证
28     print("----欧美专区----")
29
30 def japan():
31     print("----日韩专区----")
32
33 def henan():
34     login() #执行前加上验证
35     print("----河南专区----")
36
37
38
39 home()
40 america()
41 henan()

```

此时你信心满满的把这个代码提交给你的TEAM LEADER审核，没成想，没过5分钟，代码就被打回来了，TEAM LEADER给你反馈是，我现在有很多模块需要加认证模块，你的代码虽然实现了功能，但是需要更改需要加认证的各个模块的代码，这直接违反了软件开发中的一个原则“开放-封闭”原则，简单来说，它规定已经实现的功能代码不允许被修改，但可以被扩展，即：

- 封闭：已实现的功能代码块不应该被修改
- 开放：对现有功能的扩展开放

这个原则你还是第一次听说，我擦，再次感受了自己这个野生程序员与正规军的差距，BUT ANYWAY,老大要求的这个怎么实现呢？如何在不改原有功能代码的情况下加上认证功能呢？你一时想不出思路，只好带着这个问题回家继续憋，媳妇不在家，去隔壁老王家串门了，你正好落的清静，一不小心就想到了解决方案，不改源代码可以呀，

你师从沙河金角大王时，记得他教过你，高阶函数，就是把一个函数当做一个参数传给另外一个函数，当时大王说，有一天，你会用到它的，没想到这时这个知识点突然从脑子里蹦出来了，我只需要写个认证方法，每次调用需要验证的功能时，直接把这个功能的函数名当做一个参数传给我的验证模块不就行了么，哈哈，机智如我，如是你啪啪啪改写了之前的代码

```

1  #_*_coding:utf-8_*_
2
3
4  user_status = False #用户登录了就把这个改成True
5
6  def login(func): #把要执行的模块从这里传进来
7      _username = "alex" #假装这是DB里存的用户信息
8      _password = "abc!23" #假装这是DB里存的用户信息
9      global user_status
10
11     if user_status == False:
12         username = input("user:")
13         password = input("password:")
14
15         if username == _username and password == _password:
16             print("welcome login...")
17             user_status = True
18         else:
19             print("wrong username or password!")
20
21     if user_status == True:
22         func() # 看这里看这里, 只要验证通过了, 就调用相应功能
23
24 def home():
25     print("---首页---")
26
27 def america():
28     #login() #执行前加上验证
29     print("----欧美专区----")
30
31 def japan():
32     print("----日韩专区----")
33
34 def henan():
35     #login() #执行前加上验证
36     print("----河南专区----")
37
38
39
40 home()
41 login(america) #需要验证就调用 login, 把需要验证的功能 当做一个参数传给login
42 # home()
43 # america()
44 login(henan)

```

你很开心, 终于实现了老板的要求, 不改变原功能代码的前提下, 给功能加上了验证, 此时, 媳妇回来了, 后面还跟着老王, 你两家关系 非常好, 老王经常来串门, 老王也是码农, 你跟他分享了你写的代码, 兴奋的等他看完 夸奖你NB, 没成想, 老王看后, 并没有夸你, 抱起你的儿子, 笑笑说, 你这个代码还是改改吧, 要不然会被开除的, WHAT? 会开除, 明明实现了功能 呀, 老王讲, 没错, 你功能 是实现了, 但是你又犯了一个大忌, 什么大忌?

你改变了调用方式呀, 想一想, 现在没每个需要认证的模块, 都必须调用你的login()方法, 并把自己的函数名传给你, 人家之前可不是这么调用的, 试想, 如果有100个模块需要认证, 那这100个模块都得更改调用方式, 这么多模块肯定不止是一个人写的, 让每个人再去修改调用方式 才能加上认证, 你会被骂死的。。。

你觉得老王说的对, 但问题是, 如何即不改变原功能代码, 又不改变原有调用方式, 还能加上认证呢? 你苦思了一会, 还是想不出, 老王在逗你的儿子玩, 你说, 老王呀, 快给我点思路, 实在想不出来, 老王背对着你问,

老王: 学过匿名函数没有?

你: 学过学过, 就是lambda嘛

老王: 那lambda与正常函数的区别是什么?

你：最直接的区别是，正常函数定义时需要写名字，但lambda不需要

老王：没错，那lambda定好后，为了多次调用，可否也给它命名个名？

你：可以呀，可以写成plus = lambda x:x+1类似这样，以后再调用plus就可以了，但这样不就失去了lambda的意义了，明明人家叫匿名函数呀，你起了名字有什么用呢？

老王：我不是要跟你讨论它的意义，我想通过这个让你明白一个事实

说着，老王拿起你儿子的画板，在上面写了以下代码：

```
1 def plus(n):
2     return n+1
3
4 plus2 = lambda x:x+1
```

老王：上面这两种写法是不是代表 同样的意思？

你：是的

老王：我给lambda x:x+1 起了个名字叫plus2，是不是相当于def plus2(x)？

你：我擦，你别说，还真是，但老王呀，你想说明什么呢？

老王：没啥，只想告诉你，给函数赋值变量名就像def func_name 是一样的效果，如下面的plus(n)函数，你调用时可以用plus名，还可以再起个其它名字，如

```
1 calc = plus
2
3 calc(n)
```

你明白我想传达什么意思了么？

你：。。。。。。这。。。。。。嗯。。。。。。不太。。。。明白。。

老王：。。。。这。。。。呵呵。。。。。。好吧。。。。，那我在给你点一下，你之前写的下面这段调用 认证的代码

```
1 home()
2 login(america) #需要验证就调用 login, 把需要验证的功能 当做一个参数传给login
3 # home()
4 # america()
5 login(henan)
```

你之所改变了调用方式，是因为用户每次调用时需要执行login(henan)，类似的。其实稍一改就可以了呀

```
1 home()
2 america = login(america)
3 henan = login(henan)
```

这样你，其它人调用henan时，其实相当于调用了login(henan)，通过login里的验证后，就会自动调用henan功能。

你：我擦，还真是唉。。。老王，还是你nb。。。不过，等等，我这样写了好，那用户调用时，应该是下面这个样子

```
1 home()
2 america = login(america) #你在这里相当于把america这个函数替换了
3 henan = login(henan)
4
5 #那用户调用时依然写
6 america()
```

但问题在于，还不等用户调用，你的america = login(america)就会先自己把america执行了呀。。。你，你应该等我用户调用 的时候 再执行才对呀，不信我试给你看。。。

老王：哈哈，你说的没错，这样搞会出现这个问题？但你想有没有解决办法 呢？

你：我擦，你指的思路呀，大哥。。。我哪知道 下一步怎么走。。。

老王：算了，估计你也想不出来。。。学过嵌套函数没有？

你：yes,然后呢？

老王：想实现一开始你写的`america = login(america)`不触发你函数的执行，只需要在这个`login`里面再定义一层函数，第一次调用`america = login(america)`只调用到外层`login`，这个`login`虽然会执行，但不会触发认证了，因为认证的所有代码被封装在`login`里的新定义的函数里了，`login`只返回 里层函数的函数名，这样下次再执行`america()`时，就会调用里层函数啦。。。

你：。。。。。。什么？什么个意思，我蒙逼了。。。

老王：还是给你看代码吧。。

```

1  def login(func): #把要执行的模块从这里传进来
2
3      def inner():#再定义一层函数
4          _username = "alex" #假装这是DB里存的用户信息
5          _password = "abc!23" #假装这是DB里存的用户信息
6          global user_status
7
8          if user_status == False:
9              username = input("user:")
10             password = input("password:")
11
12             if username == _username and password == _password:
13                 print("welcome login...")
14                 user_status = True
15             else:
16                 print("wrong username or password!")
17
18             if user_status == True:
19                 func() # 看这里看这里，只要验证通过了，就调用相应功能
20
21         return inner #用户调用login时，只会返回inner的内存地址，下次再调用时加上()才

```

此时你仔细看了老王写的代码，感觉老王真不是一般人呀，连这种奇淫巧技都能想出来。。。心中默默感谢上天赐你一个牛邻居。

你：老王呀，你这个姿势很nb呀，你独创的？

此时你媳妇噗嗤的笑出声来，你也不知道她笑个球。。。

老王：呵呵，这不是我独创的呀当然，这是开发中一个常用的玩法，叫语法糖，官方名称“装饰器”，其实上面的写法，还可以更简单

可以把下面代码去掉

```

1  america = login(america) #你在这里相当于把america这个函数替换了

```

只在你要装饰的函数上面加上下面代码

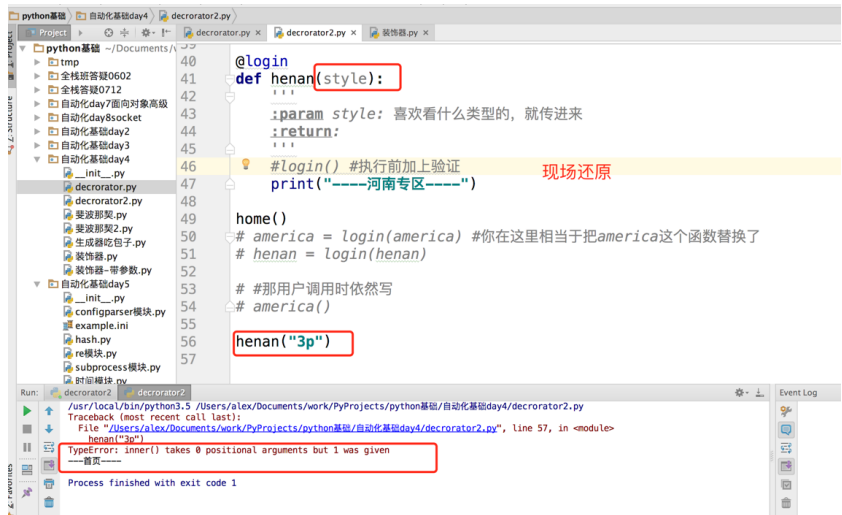
```

1  @login
2  def america():
3      #login() #执行前加上验证
4      print("----欧美专区----")
5
6  def japan():
7      print("----日韩专区----")
8
9  @login
10 def henan():
11     #login() #执行前加上验证
12     print("----河南专区----")

```

效果是一样的。

你开心的玩着老王教你的新姿势，玩着玩着就手贱给你的“河南专区”版块加了个参数，然后，结果 出错了。。。



你：老王，老王，怎么传个参数就不行了呢？

老王：那必然呀，你调用henan时，其实是相当于调用的login，你的henan第一次调用时 henan = login(henan)，login就返回了inner的内存地址，第2次用户自己调用 henan("3p")，实际上相当于调用的inner，但你的inner定义时并没有设置参数，但你给他传了个参数，所以自然就报错了呀

你：但是我的 版块需要传参数呀，你不让我传不行呀。。。

老王：没说不能让你传，稍做改动便可。。



老王：你再试试就好了。

你：果然好使，大神就是大神呀。。。 不过，如果有多个参数呢？

老王：。。。。老弟，你不要什么都让我教你吧，非固定参数你没学过么？

*args,**kwargs...

你：噢。。。还能这么搞？nb,我再试试。

你身陷这种新玩法中无法自拔，竟没注意到老王已经离开，你媳妇告诉你说了为了不打扰你加班，今晚带孩子去跟她姐妹住，你觉得媳妇真体贴，最终，你终于搞定了所有需求，完全遵循开放-封闭原则，最终代码如下。

[+ View Code](#)

此时，你已累的不行了，洗洗就抓紧睡了，半夜，上厕所，隐隐听到隔壁老王家有微弱的女人的声音传来，你会心一笑，老王这家伙，不声不响找了女朋友也不带给我看看，改天

一定要见下真人。。。。

第二天早上，产品经理又提了新的需求，要允许用户选择用qq\weibo\weixin认证，此时的你，已深谙装饰器各种装逼技巧，轻松的就实现了新的需求。

田 带参数的装饰器

3.Json & pickle 数据序列化

参考 <http://www.cnblogs.com/alex3714/articles/5161349.html>

4.软件目录结构规范

为什么要设计好目录结构？

"设计项目目录结构"，就和"代码编码风格"一样，属于个人风格问题。对于这种风格上的规范，一直都存在两种态度：

1. 一类同学认为，这种个人风格问题"无关紧要"。理由是能让程序work就好，风格问题根本不是问题。
2. 另一类同学认为，规范化能更好的控制程序结构，让程序具有更高的可读性。

我是比较偏向于后者的，因为我是前一类同学思想行为下的直接受害者。我曾经维护过一个非常不好读的项目，其实现的逻辑并不复杂，但是却耗费了我非常长的时间去理解它想表达的意思。从此我个人对于提高项目可读性、可维护性的要求就很高了。"项目目录结构"其实也是属于"可读性和可维护性"的范畴，我们设计一个层次清晰的目录结构，就是为了达到以下两点：

1. 可读性高：不熟悉这个项目的代码的人，一眼就能看懂目录结构，知道程序启动脚本是哪个，测试目录在哪儿，配置文件在哪儿等等。从而非常快速的了解这个项目。
2. 可维护性高：定义好组织规则后，维护者就能很明确地知道，新增的哪个文件和代码应该放在什么目录之下。这个好处是，随着时间的推移，代码/配置的规模增加，项目结构不会混乱，仍然能够组织良好。

所以，我认为，保持一个层次清晰的目录结构是有必要的。更何况组织一个良好的工程目录，其实是一件很简单的事儿。

目录组织方式

关于如何组织一个较好的Python工程目录结构，已经有一些得到了共识的目录结构。在Stackoverflow的[这个问题](#)上，能看到大家对Python目录结构的讨论。

这里面说的已经很好了，我也不打算重新造轮子列举各种不同的方式，这里面我说一下我的理解和体会。

假设你的项目名为foo，我比较建议的最方便快捷目录结构这样就足够了：

```
Foo/  
|-- bin/  
|   |-- foo  
|
```

```
|-- foo/
|   |-- tests/
|   |   |-- __init__.py
|   |   |-- test_main.py
|   |
|   |-- __init__.py
|   |-- main.py
|
|-- docs/
|   |-- conf.py
|   |-- abc.rst
|
|-- setup.py
|-- requirements.txt
|-- README
```

简要解释一下：

1. bin/：存放项目的一些可执行文件，当然你可以起名script/之类的也行。
2. foo/：存放项目的所有源代码。（1）源代码中的所有模块、包都应该放在此目录。不要置于顶层目录。（2）其子目录tests/存放单元测试代码；（3）程序的入口最好命名为main.py。
3. docs/：存放一些文档。
4. setup.py：安装、部署、打包的脚本。
5. requirements.txt：存放软件依赖的外部Python包列表。
6. README：项目说明文件。

除此之外，有一些方案给出了更加多的内容。比如LICENSE.txt,ChangeLog.txt文件等，我没有列在这里，因为这些东西主要是项目开源的时候需要用到。如果你想写一个开源软件，目录该如何组织，可以参考[这篇文章](#)。

下面，再简单讲一下我对这些目录的理解和个人要求吧。

关于README的内容

这个我觉得是每个项目都应该有的一个文件，目的是能简要描述该项目的信息，让读者快速了解这个项目。

它需要说明以下几个事项：

1. 软件定位，软件的基本功能。
2. 运行代码的方法：安装环境、启动命令等。
3. 简要的使用说明。
4. 代码目录结构说明，更详细点可以说明软件的基本原理。
5. 常见问题说明。

我觉得有以上几点是比较好的一个README。在软件开发初期，由于开发过程中以上内容可能不明确或者发生变化，并不是一定要在一开始就将所有信息都补全。但是在项目完结的时候，是需要撰写这样的一个文档的。

可以参考Redis源码中[Readme](#)的写法，这里面简洁但是清晰的描述了Redis功能和源码结构。

关于requirements.txt和setup.py

setup.py

一般来说，用setup.py来管理代码的打包、安装、部署问题。业界标准的写法是用Python流行的打包工具[setuptools](#)来管理这些事情。这种方式普遍应用于开源项目中。不过这里的核心思想不是用标准化的工具来解决这些问题，而是说，**一个项目一定要有一个安装部署工具**，能快速便捷的在一台新机器上将环境装好、代码部署好和将程序运行起来。

这个我是踩过坑的。

我刚开始接触Python写项目的时候，安装环境、部署代码、运行程序这个过程全是手动完成，遇到过以下问题：

1. 安装环境时经常忘了最近又添加了一个新的Python包，结果一到线上运行，程序就出错了。
2. Python包的版本依赖问题，有时候我们程序中使用的是一个版本的Python包，但是官方的已经是最新的包了，通过手动安装就可能装错了。
3. 如果依赖的包很多的话，一个一个安装这些依赖是很费时的事情。
4. 新同学开始写项目的时候，将程序跑起来非常麻烦，因为可能经常忘了要怎么安装各种依赖。

setup.py可以将这些事情自动化起来，提高效率、减少出错的概率。"复杂的东西自动化，能自动化的东西一定要自动化。"是一个非常好的习惯。

setuptools的文档比较庞大，刚接触的话，可能不太好找到切入点。学习技术的方式就是看他人是怎么用的，可以参考一下Python的一个Web框架，flask是如何写的：[setup.py](#)

当然，简单点自己写个安装脚本（deploy.sh）替代setup.py也未尝不可。

requirements.txt

这个文件存在的目的是：

1. 方便开发者维护软件的包依赖。将开发过程中新增的包添加进这个列表中，避免在setup.py安装依赖时漏掉软件包。
2. 方便读者明确项目使用了哪些Python包。

这个文件的格式是每一行包含一个包依赖的说明，通常是flask>=0.10这种格式，要求是这个格式能被pip识别，这样就可以简单的通过 `pip install -r requirements.txt`来把所有Python包依赖都装好了。具体格式说明：[点这里](#)。

关于配置文件的使用方法

注意，在上面的目录结构中，没有将conf.py放在源码目录下，而是放在docs/目录下。

很多项目对配置文件的使用做法是：

1. 配置文件写在一个或多个python文件中，比如此处的conf.py。
2. 项目中哪个模块用到这个配置文件就直接通过`import conf`这种形式来在代码中使用配置。

这种做法我不太赞同：

1. 这让单元测试变得困难（因为模块内部依赖了外部配置）
2. 另一方面配置文件作为用户控制程序的接口，应当可以由用户自由指定该文件的路径。
3. 程序组件可复用性太差，因为这种贯穿所有模块的代码硬编码方式，使得大部分模块都依赖conf.py这个文件。

所以，我认为配置的使用，更好的方式是，

1. 模块的配置都是可以灵活配置的，不受外部配置文件的影响。
2. 程序的配置也是可以灵活控制的。

能够佐证这个思想的是，用过nginx和mysql的同学都知道，nginx、mysql这些程序都可以自由的指定用户配置。

所以，不应当在代码中直接`import conf`来使用配置文件。上面目录结构中的conf.py，是给出的一个配置样例，不是在写死在程序中直接引用的配置文件。可以通过给main.py启动参数指定配置路径的方式来让程序读取配置内容。当然，这里的conf.py你可以换个类似的名字，比如settings.py。或者你也可以使用其他格式的内容来编写配置文件，比如settings.yaml之类的。

5.本节作业

作业需求：

模拟实现一个ATM + 购物商城程序

- 1. 额度 15000或自定义
- 2. 实现购物商城，买东西加入 购物车，调用信用卡接口结账
- 3. 可以提现，手续费5%
- 4. 每月22号出账单，每月10号为还款日，过期未还，按欠款总额 万分之5 每日计息
- 5. 支持多账户登录
- 6. 支持账户间转账
- 7. 记录每月日常消费流水
- 8. 提供还款接口
- 9. ATM记录操作日志
- 10. 提供管理接口，包括添加账户、用户额度，冻结账户等。。。
- 11. 用户认证用装饰器

示例代码 https://github.com/triaquae/py3_training/tree/master/atm

简易流程图：

<https://www.processon.com/view/link/589eb841e4b0999184934329>

分类: [Python自动化开发之路](#)

好文要顶

关注我

收藏该文

金角大王

关注 - 5

粉丝 - 10868

+加关注

190

posted @ 2016-08-12 15:12 金角大王 阅读(70463) 评论(17) 编辑 收藏

评论列表

#1楼	2016-08-15 17:27 freedom_dog	回复 引用
多谢。		支持(0) 反对(0)
#2楼	2017-02-10 17:42 super-sos	回复 引用
打死你个龟孙儿		支持(1) 反对(0)
#3楼	2017-02-19 20:39 great_zhi	回复 引用
打死你个龟孙儿		支持(0) 反对(0)
#4楼	2017-06-21 09:55 Frank_srv_world	回复 引用
传参的过程还是有些迷糊		支持(0) 反对(0)
#5楼	2017-07-07 11:30 小温xy	回复 引用
单线程实现并发效果的producer函数不用带参数name吧		支持(0) 反对(1)
#6楼	2017-07-18 15:14 hao_xiaoyu	回复 引用
alex的小故事写的也是津津有味		支持(2) 反对(0)
#7楼	2017-09-14 14:28 SmartMing	回复 引用
楼主没老婆或者经常扮演隔壁老王。让人恶心的段子		支持(6) 反对(6)

- #8楼 2017-12-03 20:23 Byron-He

回复 引用

@ SmartMing
这只是个一简单的举例方式，不知道为什么你这样抨击楼主！

支持(0) 反对(2)
- #9楼 2018-01-07 00:59 彭世瑜

回复 引用

交个作业，做了3天总算做好了，分享给大家
<https://github.com/mouday/Atm>

支持(3) 反对(0)
- #10楼 2018-03-15 21:28 小夕公子

回复 引用

Alex很会编故事。。。。。。

支持(0) 反对(0)
- #11楼 2018-03-28 14:47 pennychenpei

回复 引用

```
#__coding:utf-8__
__author__ = 'Alex Li'

import time
def consumer(name):
    print("%s 准备吃包子啦!" %name)
    while True:
        baozi = yield

    print("包子[%s]来了,被[%s]吃了!" %(baozi,name))

def producer(name):
    c = consumer('A')
    c2 = consumer('B')
    c.__next__()
    c2.__next__()
    print("老子开始准备做包子啦!")
    for i in range(10):
        time.sleep(1)
        print("做了2个包子!")
        c.send(i)
        c2.send(i)

producer("alex")

这个执行报错

File "C:\Users\cw210659\Desktop\python learning\hanshu.py", line 75, in producer
print(c.__next__())
AttributeError: 'generator' object has no attribute '__next__'
```

支持(0) 反对(0)
- #12楼 2018-03-28 15:17 pennychenpei

回复 引用

```
c.__next__()
c2.__next__() 改成next (c) 和next(c2)就是ok的。
```

支持(0) 反对(0)
- #13楼 2018-04-04 10:46 alex_hrg

回复 引用

花了三天时间，终于把作业交上，发个地址大家共同学习<https://github.com/hrghrg/atm>

支持(0) 反对(0)
- #14楼 2018-08-15 17:51 charles_guo

回复 引用

@ pennychenpei
改成c.next()

支持(0) 反对(0)
- #15楼 2018-09-12 13:04 yoocin

回复 引用

@ pennyche换成next (c1) 跟next (c2) 就好了

支持(0) 反对(0)
- #16楼 2019-05-24 05:18 我也不想这么菜

回复 引用

```
import time
def consumer(name):
    print("%s 准备吃包子啦!" %name)
    while True:
        baozi = yield
```



```
print("包子[%s]来了,被[%s]吃了!" %(baozi,name))

def producer(name):
    c = consumer('A')
    c2 = consumer('B')
    c.__next__()
    c2.__next__()
    print("老子开始准备做包子啦!")
    for i in range(10):
        time.sleep(1)
        print("做了2个包子!")
        c.send(i)
        c2.send(i)

producer("alex")
```

哪位大佬能指点一下这里的
c.__next__()
c2.__next__()中的__next__是用老干啥的啊

支持(1) 反对(0)

#17楼 2020-04-22 01:12 zoey_chou [回复](#) [引用](#)

```
def fib(max): n, a, b = 0, 0, 1 while n < max: print(b) a, b = b, a + b n = n + 1 return 'done'
```

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。
提问:目前这个斐波那契还是一个函数,也就是按顺序执行的,为何是先打印print(b),然后计算a,b的值?挪了print b的位置值又发生了变化

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

发表评论

编辑 预览

B

支持 Markdown

[提交评论](#) [退出](#) [订阅评论](#)

[Ctrl+Enter快捷键提交]

- 【推荐】了不起的开发者，挡不住的华为，园子里的品牌专区
- 【推荐】有道智云周年庆，API服务大放送，注册即送100元体验金！
- 【推荐】超50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库
- 【推荐】2019热门技术盛会400则演讲资料全收录

**相关博文：**

- [【Python之路Day4】基础篇](#)
 - [python之路day4](#)
 - [python之路_day4](#)
 - [python之路, Day4-Python基础](#)
 - [Python之路,Day4 - Python基础4](#)
- » [更多推荐...](#)

最新 IT 新闻：

- [Github项目推荐 | retinaface 人脸识别](#)
 - [谷歌被指监视Android设备上的竞争App数据以提升自家服务](#)
 - [收钱码借给他人 可能会成为诈骗共犯！官方提醒：这些行为严禁](#)
 - [你不知道的“抖加淘客团”与抖音上的算法博弈](#)
 - [暑假旅游小高峰 旅游行业都在烧钱赚人气](#)
- » [更多新闻...](#)