

# Python\_Introduction\_Part\_3

October 19, 2020

Week 3

This week we will be examining the basic functionality that enables you to build up more complex algorithms, which will help you to create routines that can begin to do set tasks; and help us to improve our work-flow.

This involves working with control statements and developing your own functions.

---

## 1 Conditional Statements

In order to create programs that are able to perform useful tasks for us, we will need to invoke conditional statements and loops (in combination with relational and logical operators). I have already used a couple in previous sections.

### 1.1 The *if* statement

The *if* statement is a basic conditional statement.

The format is:

```
if condition:
    expression
```

Here is a basic example:

```
[1]: x=2
      if x==2:                # If x equals 2 then carry out expression
          print('Hi there')  # This line has to be indented
```

Hi there

The *if* statement basically evaluates the expression in the condition. If the expression is True, then the following indented lines of code are run. Here, if the expression is false, then nothing happens.

## 1.2 if else

You may actually want one of two actions to be performed depending on whether the condition is true or false. We will then need to include the *else* statement

```
[5]: bird1=0
      bird2=0

      spotted='pidgeon'

      if spotted=='pidgeon':
          bird1 +=1 # This operation adds 1 onto the previous value of the array
      ↪
      else:
          bird2 +=1

      print(bird1,bird2)
```

1 0

Change the spotted variable to 'crow' and see what happens.

The above code can be thought of as *if (logic satisfied) do task1 else do task2*.

## 1.3 if else elif

Sometimes there are more than 2 choices, and we can use the *elif* statement

```
[1]: choice = 'cat'

      if choice == 'cat':
          print("I like cats too! We should be friends.")
      elif choice == 'dog':
          print("Dogs are okay...")
      elif choice == 'rat':
          print("Who likes rats?!")
      else:
          print("That pet was not an option.")
```

I like cats too! We should be friends.

Change the variable choice to either 'dog' or 'rat' to see what happens.

## 2 Loops

### 2.1 *for* loop

A *for* loop will loop over a section of code for the number of times specified. The *for* loops in Python might look slightly different from those in other languages, e.g., Matlab, IDL. The *for* loops in Python are ‘collection-based’ and they iterate over a collection of objects (see [here for additional details](#) ).

A simple example is:

```
[7]: for x in range(5): # object with elements [0,1,2,3,4]
      print('The number is ', x)
```

```
The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
```

The range function here is similar to *arange* in numpy - it generates a sequence of evenly spaced samples. The *for* loop then iterates over the integers.

Here is another example with strings in a list:

```
[40]: a = ['apple', 'banana', 'carrot']

for fruit in a: # Note that you can name the iterator what you
    print(fruit) # like, changing fruit to, e.g. goat will not
                # effect the execution.
```

```
apple
banana
carrot
```

In both of these cases we have created an *iterator*, which provides a consistent way to iterate over sequences. Here is another example with a dictionary (similar to an example shown previously in week 1):

```
[2]: some_dict = {'a': 1, 'b': 3, 'c': 5}

for keys in some_dict:
    print(keys)
```

```
c
b
a
```

The code written above does not allow of to iterate over keys and values in the dictionary. You can try a *for* loop of the form: `>>for keys,values in some_dict:`

however, it will not work! To iterate over both keys and values we need to use the method `dict.items()`

```
[4]: for keys, values in some_dict.items():  
      print(keys, values)
```

```
c 5  
b 3  
a 1
```

You can also control the flow of *for* loops with *break* and *continue* statements.

A *break* statement will cause a loop to terminate. In the following example we will create a loop break when a conditional statement is satisfied.

```
[9]: for x in range(5):  
      if x == 3:          # If our iterator gets to 3 then break  
          break  
      print('The number is ', x)
```

```
The number is 0  
The number is 1  
The number is 2
```

A *continue* statement terminates the current iteration but the loop continues.

```
[10]: for x in range(5):  
       if x == 3:  
           continue  
       print('The number is ', x)
```

```
The number is 0  
The number is 1  
The number is 2  
The number is 4
```

You may also want to keep track of the iteration number during the process. A neat way to do this is use the function `enumerate`.

```
[3]: for index, keys in enumerate(some_dict):  
      print(index, keys)
```

```
0 c  
1 b  
2 a
```

## 2.2 While loops

Like the *for* statement, the *while* statement starts a loop. However, the looping does not stop after a set number of iterations. It only stops when a given conditional statement is `False`, i.e., loops

while it is True. Essentially, the while loop can be thought of as a looping *if* statement.

```
[41]: prompt = "Can you guess what number I am thinking of? "

number=input(prompt)          # The input function enables some user
    ↪interaction

                                # - number variable is string
while number != "42":         # This says ' loop while number is not equal to
    ↪42'
    number = input('No, keep going! ')

print("Yes, well done!")
```

```
Can you guess what number I am thining of? 3
No, keep going! 2
No, keep going! 42
Yes, well done!
```

It is worth noting that a *while* loop can potentially go on forever!! This will likely be due to poor construction of the loop. If you find yourself stuck in a while loop, then you can stop the execution with ‘ctrl-c’.

---

## 3 Functions

So far, we have been using various pre-existing functions in Python. However, we can, and will often want to, develop our own functions to perform specific tasks. There are two methods for defining functions. The second method is discussed in Section ??.

### 3.1 In-line functions

The first method is one that can be done ‘in-line’, in the interpreter.

We have to use the *def* keyword when defining the function. This is then followed by the name of the function and parenthesis, between which you give any parameters for the function.

```
[1]: def welcome_me(name):      # function name - welcome_me, parameter - name
    print('Hello ' + name)      # These two lines are the function definition

welcome_me('Richard')          # Here I am calling the function
```

```
Hello Richard
```

Information is passed into the function as an argument. Here, *name* is the parameter and ‘Richard’ is the argument.

The function can also return information, and that information be stored in variables.

```
[9]: def square_number(x):  
      return x**2  
  
y=square_number(4)  # Here x in the function takes on the value 4  
print(y)
```

16

In theory, you can have as many arguments as you want, but when inputting them you will have respect the order they are input.

You can also assign arguments by their names, and here order doesn't matter.

```
[3]: def subtract_numbers(x, y):  
      return x-y  
  
# Here x in the function takes on the value 4 and y is 3  
z = subtract_numbers(4, 3)  
print(z)  
  
# Here x in the function takes on the value 3 and y is 4  
z = subtract_numbers(3, 4)  
print(z)  
  
z = subtract_numbers(y=3, x=4)  # assign argument by name  
print(z)
```

1  
-1  
1

### 3.1.1 A small aside

Note that depending upon whether the argument that is passed to the function is mutable or immutable, one of two things can happen.

In programming, arguments passed to functions are either *passed-by-reference* or *passed-by-value*.

If the arguments are passed-by-reference, the function gets an explicit reference to the argument (rather than a copy) - hence if your function changes what the argument variable refers to, the change reflects back in the calling function. This is the case for mutable data types.

This is slightly technical, but is illustrated with a simple example.

Remember, the `id()` function gives us the unique id number for the Python object.

```
[30]: def change_list(lis):  
      lis.append([1, 2, 'cat'])  
      print('Inside function ', id(lis))
```

```

    return

lis = ['dog', 3, 4]
print('Outside function ', id(lis))
change_list(lis)
print(lis)

```

```

Outside function  4377955720
Inside function   4377955720
['dog', 3, 4, [1, 2, 'cat']]

```

For immutable data types, e.g., integers, the arguments are passed-by-value. Hence, the function effectively creates a copy of the variable to use within function. There is no change to the variables outside the function.

Again, a simple example highlights this.

```

[31]: def multiply_number_by_five(x):
        x *= 5
        print('Inside function ', id(x))
        return x

x = 4
print('Outside function ', id(x))
print(multiply_number_by_five(x))
print(x)                                # x is unchanged

```

```

Outside function  4316223952
Inside function   4316224464
20
4

```

Further reading on this can be found here [https://www.python-course.eu/python3\\_passing\\_arguments.php](https://www.python-course.eu/python3_passing_arguments.php)

## 3.2 Returning multiple values

A function can also return multiple values, and there are a number of ways to access these multiple values. The next two examples use the same function but show two different ways to access the values.

In this first example we provide a single variable to assign the function output to. The result is that the variable is a tuple.

```

[33]: def issac_details():
        name = "Issac Newton"
        dob = "4th January 1943"

```

```
pob = "Woolsthorpe"
return name, dob, pob

details = issac_details()
print(details, type(details))
```

('Issac Newton', '4th January 1943', 'Woolsthorpe') <class 'tuple'>

This time, we provide 3 variables to assign the function output to.

```
[36]: name, dob, place = issac_details()
print(name)
print(dob)
print(place)
```

```
Issac Newton
4th January 1943
Woolsthorpe
<class 'str'>
```

Note that the first method returns a tuple, hence we can't alter the contents of elements in it (tuples are immutable).

### 3.3 Global vs local variables

Not all variables are accessible everywhere in a script.

Variable names in the main body of script, i.e. defined outside of functions, are called *global* variables (said to be in the global scope).

Local variables are defined within the function (local scope). The local variables cease to exist once the function has finished.

Depending on how the function is defined, the function can access these global variables.

First the local scope is checked to see whether a variable exists, then the global scope.

The following function will use global variables:

```
[38]: def print_x():
      print(x)      # x is not defined in the function

x = 10
print_x()
```

10

In the following function, we create a *local* variable in the function (through the argument definition). Even though it has the same name as the global variables, Python assumes we want the



local variables - hence gives us an error

```
[39]: def print_x(x):      # x is defined in the function header,
      print(x)           # so a local variables exists.

x = 10
print_x()
```

```

      □
↳ -----

      TypeError                                Traceback (most recent call
↳ last)

      <ipython-input-39-d628756ed6ce> in <module>()
          3
          4 x=10
      ----> 5 print_x()

      TypeError: print_x() missing 1 required positional argument: 'x'
```

The following will also give an error, even though the local variable is defined after the print statement.

```
[40]: def print_x():
      print(x)
      x = 20

x = 10
print_x()
```

```

      □
↳ -----

      UnboundLocalError                        Traceback (most recent call
↳ last)

      <ipython-input-40-f4a48215c6c0> in <module>()
          4
          5 x=10
      ----> 6 print_x()
```

```

<ipython-input-40-f4a48215c6c0> in print_x()
      1 def print_x():
----> 2     print(x)
      3     x=20
      4
      5 x=10

```

UnboundLocalError: local variable 'x' referenced before assignment

What happens if we now just define a *local* variable in the function that doesn't correspond to a global variable.

```

[42]: def f(y):
      temp = y

x = 10
f(x)
print(temp)

```

```

      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
    18
```

customisable when the need arises. Hence, one way to achieve this flexibility is to define default arguments. The default arguments are given by a keyword and the default value.

In the following, I define an optional parameter defined by the keyword *value* with the default value of 1. First, I don't give the function anything for *value*. I then provide the optional parameter in two ways.

```
[45]: def price_of(value='£1000'):
        print('The price of an Ipad is '+value)

price_of()
price_of('£500')
price_of(value='£400')
```

```
The price of an Ipad is £1000
The price of an Ipad is £500
The price of an Ipad is £400
```

You can also define the argument as None - which is a null value. Note, this is not the same as 0.

```
[9]: def price_of(value=None):
        if value is None:
            value = '£500'

        print('The price of an Ipad is '+value)

price_of()
price_of('$200')
```

```
The price of an Ipad is £500
The price of an Ipad is $200
```

### 3.5 Flexible arguments

You may also want to write a function that takes in a number of arguments, but you don't know in advance how many will be given.

To do this we add `__*args__` to the function header. This then passes all arguments as a tuple to the function body.

```
[1]: def names(*args):
        name_string = ''           # initialize empty string
        for name in args:         # Loop over variables in tuple
            name_string += name

        return name_string
```

```
names('Richard', 'Jeff', 'David', 'Brian')
```

```
[1]: 'RichardJeffDavidBrian'
```

You can also do something similar for keyword arguments. This time, however, you require a double star, `**kwargs`. The keywords and values are then turned into a dictionary in the function body.

```
[4]: def some_func(**kwargs):  
  
    for keys, values in kwargs.items():  
        print("Key: "+keys+', Value:', str(values))  
  
some_func(Lost='cat', Reward='£500')
```

```
Key: Reward, Value: £500
```

```
Key: Lost, Value: cat
```

Note, it is not the names args and kwargs that make this work - it is the single and double asterixes. However, using the names args and kwargs is standard practice.

## 4 Modules & Imports

We have already discussed running a list of instructions in a Python file (i.e. a file ending in `.py`).

However we can also create and use modules (or packages), which are formatted `.py` files.

We have already come across modules, such as `numpy`, and modules are imported as:

```
import some_module
```

There are many such modules, each containing special functions, methods or types and are designed to undertake a particular problem.

We can create our own modules simply. Enter the following in a new python file and save it as `test_module.py`

```
[ ]: # test_module.py  
const = 4.67  
  
def addFour(x):  
    return x+4  
  
def multip(a, b):  
    return a*b
```

We can load the module in with `import`, i.e. `> >import test_module`

and use the constant and functions in the module as such:

```
>res=test__module.addFour(7)
>print(test__module.const)
```

We can also import the functions as

```
>from test__module import addFour, multip, const
>res=multip(5,const)
```

Or we can do: > >import test\_\_module as tm

```
>res=tm.addFour(tm.const)
```

## 5 Error handling & exceptions

A final word in this notebook is on error messages.

As we have seen, Python will throw error (or exception) messages if you do something incorrectly. The error messages are useful for understanding what has gone wrong, and will help you correct any problems with your code.

Any time you see an error message you don't understand - trying [googling](#) it.

They may appear cryptic, but will help you fix any issues with you code! The quicker you start to use and understand the error messages, the easier it will be diagnose and fix problems.

Take the following:

```
[5]: float('1.234') # turns string to a float
```

```
[5]: 1.234
```

```
[6]: float('word')
```

```

      □
↳-----
ValueError                                Traceback (most recent call↳
↳last)

    <ipython-input-6-cb46d0ec6998> in <module>()
----> 1 float('word')

ValueError: could not convert string to float: 'word'
```

Here a `ValueError` has been raised because the `float` function cannot convert the string 'word' to a number.

You can also add error and exception handling to your functions. The following is called a *try/except* block.

```
[14]: def add(a,b):  
    try:  
        return a+b  
    except:  
        return 'Check arguments are both integers or floats'  
  
add(1,'y')
```

```
[14]: 'Check arguments are both integers or float'
```

The `except` part is only executed if the `try` part raises an exception.

```
[ ]:
```