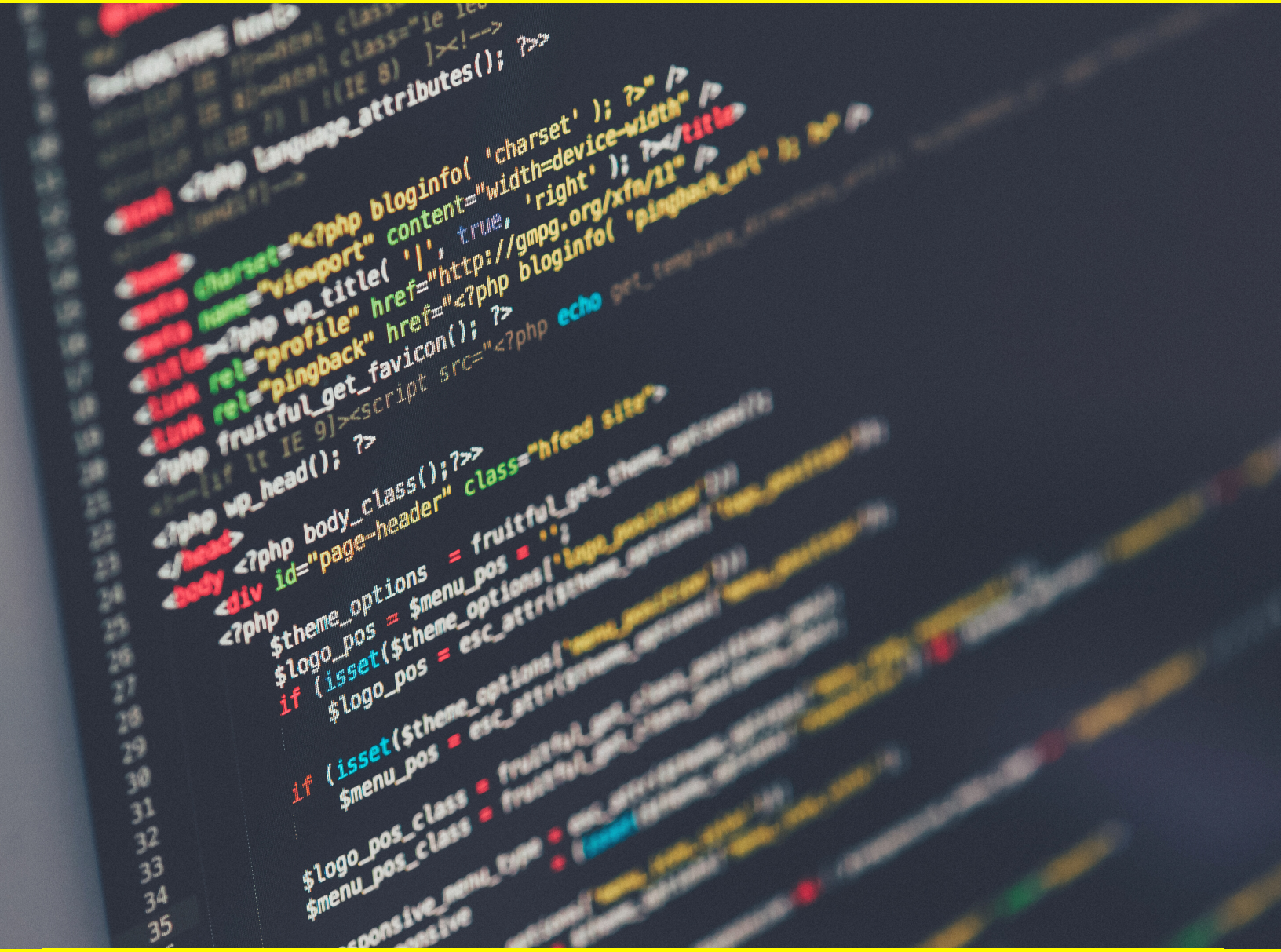
the fast lane to python

PART8



BY:

Arshadul Shaikh

Ref taken from Coursera + Udemy

“LEARNING NEVER EXHAUSTS THE MIND!!!
For the things we have to learn before we can do them, we learn by doing them, and i believe whoever thought that it will be tough to learn Python so quick and with so much clarity must think again :D

So proud that we have reached Part 7, with so much consistency, but we still have a long way to so push ourselves, cause no one else is going to do it for you

too much motivation, LET's ROLL:

Nested Data and Nested Iteration

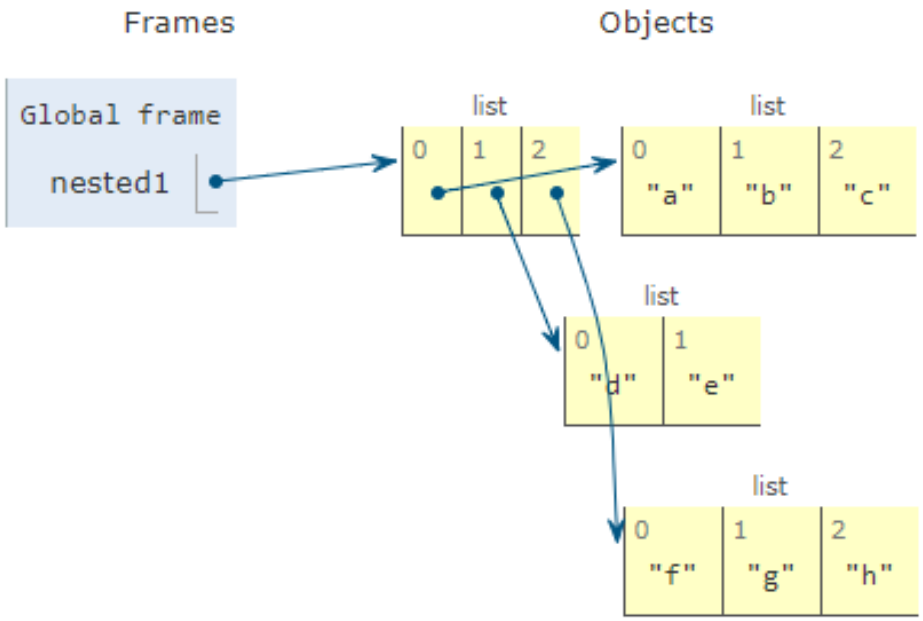
Lists with Complex Items

```
1 complex_eg = [['a', 'b', 'c'],['d', 'e'],['f', 'g', 'h']]
2 print(complex_eg[0])
3 print(len(complex_eg))
4 complex_eg.append(['i'])
5 print("-------")
6 for L in complex_eg:
7     print(L)
8
```

```
['a', 'b', 'c']
3
-------
['a', 'b', 'c']
['d', 'e']
['f', 'g', 'h']
['i']
```

- Line 2 prints out the first item from the list that **complext_eg** is **bound** to. That item is itself a list, so it prints out with square brackets.
- It has length 3, which prints out on line 3.
- Line 4 adds a new item to complex_eg. **It is a list with one element, 'i'** (it a list with one element, it's not just the string 'i').

If you find it tough to visualize what's happening behind the scenes in the program try to use
http://www.pythontutor.com/visualize.html#mode=display
as much as you can for better understanding of flow of interpreter.



For eg, when you try to run it in pythontutor you might observe below points :

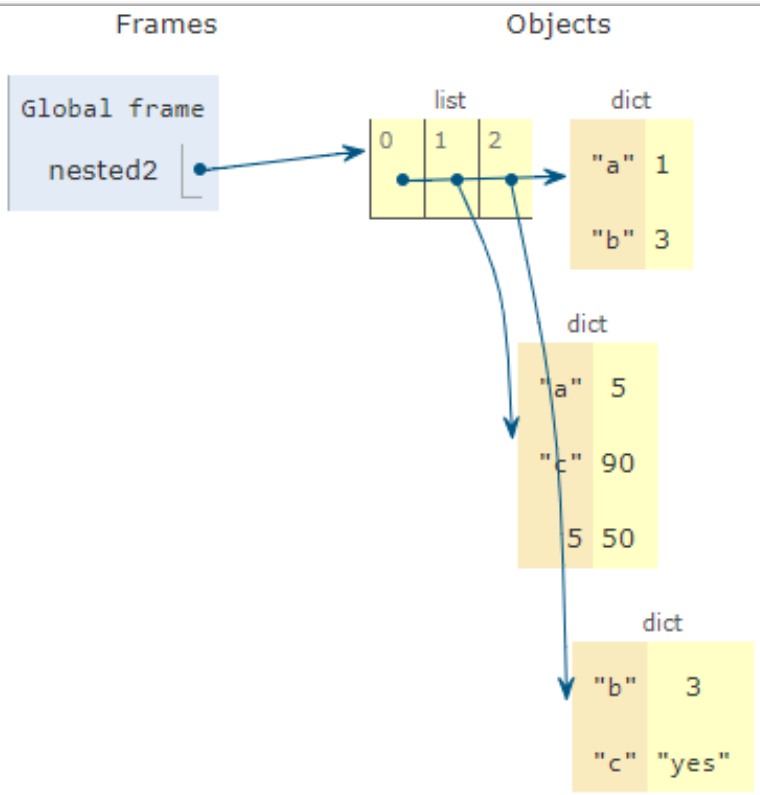
- When you get to step 4 of the execution, take a look at the object that variable complex_eg points to. It is a list of three items, numbered 0, 1, and 2.
- The item in slot 1 is small enough that it is shown right there as a list containing items “d” and “e”. The item in slot 0 didn’t quite fit, so it is shown in the figure as a pointer to another separate list; same thing for the item in slot 2, the list ['f', 'g', 'h'].

```
complex_eg = [['a', 'b', 'c'],['d', 'e'],['f', 'g', 'h']]
y=complex_eg[0] #this will initialize y with 0th element
                #of list
print(y)
print(y[0]) #this will print out the 0th element of y
print([10,20,30][0]) #interpreter will first understand
                     #the context of this command
                     #watching that there is no variable
                     #before [] it will considre the first
                     #square brackets as list and then start
                     #indexing

print(complex_eg[0][0]) #here interpreter will find complex_eg
#before the [] so will load the list which is being pointed by
#the variable and then starte finding the values as per index
#and then sub_index and so on if the list has multiple sub_lists

```

```
['a', 'b', 'c']
a
10
a
```

```
nested2 = [{'a': 1, 'b': 3}, {'a': 5, 'c': 90, 5: 50}, {'b': 3, 'c': "yes"}]
print(nested2)
```

```
[{'a': 1, 'b': 3}, {'a': 5, 'c': 90, 5: 50}, {'b': 3, 'c': 'yes'}]
```



This is how a dictionary will be visualized by python tutor. Let's try to perform some operations over list of dictionaries

```
nested2 = [{'a': 1, 'b': 3}, {'a': 5, 'c': 90, 5: 50}, {'b': 3, 'c': "yes"}]

#write code to print the value associated with key
#'c' in the second dictionary (90)
print(nested2[1]['c'])

#write code to print the value associated with key
#'b' in the third dictionary
print(nested2[2]['b'])
#add a fourth dictionary add the end of the list;
#print something to check your work.
nested2.append('check')


#change the value associated with 'c' in the third
#dictionary from "yes" to "no"; print something to
#check your work
nested2[2]['c']='no'
print(nested2)
```

```
90
3
[{'a': 1, 'b': 3}, {'a': 5, 'c': 90, 5: 50}, {'b': 3, 'c': 'no'}, 'check']
```

We can even have a list of functions (!)

```
1  def square(x): #square fn, will returen sqaure of number
2      return x*x
3
4  #the list has 3 functions
5  #square is the fn that we defined above
6  #abs is the absolute function which will return only +value
7  #lambda expression will take an num and return num +1
8
9  L = [square, abs, lambda x: x+1]
10
11 print("****function names & and its address****")
12 for f in L:
13     print(f)
14
15 print("****call each of them with a actual arguments****")
16 for f in L:
17     print(f(-2))
18
19 print("****just the first one in the list****")
20 print(L[0])
21 print(L[0](3))
```

```
****function names & and its address****
<function square at 0x0000022344212EE0>
<built-in function abs>
<function <lambda> at 0x0000022344212940>
****call each of them with a actual arguments****
4
2
-1
****just the first one in the list****
<function square at 0x0000022344212EE0>
9
```

- Here, L is a list with three items. All those items are functions.
  - The first is the function square that is defined on lines 1 and 2
  - The second is the built-in python function abs.
  - The third is an anonymous function that returns one more than its input.

- In the first for loop which is at line 12, we do not call the functions, we just output their printed representations. The output <function square> confirms that square truly is a function object.
- For some reason, in our online environment, it's not able to produce a nice printed representation of the built-in function abs, so it just outputs <unknown>In
- the second for loop, we call each of the functions, passing in the value -2 each time and printing whatever value the function returns.
- The last two lines just emphasize that there's nothing special about lists of functions. They follow all the same rules for how python treats any other list. Because L[0] picks out the function square, L[0](3) calls the function square, passing it the parameter 3.

Nested Dictionaries

As we have seen in above pages that lists can contain items of any type, the value associated with **a key in a dictionary can also be an object of any type**. In particular, it is often useful to have a list or a dictionary as a value in a dictionary. And of course, those lists or dictionaries can also contain lists and dictionaries. There can be many layers of nesting.

*****KEY POINT**
Only the values in dictionaries can be objects of arbitrary type. The keys in dictionaries must be one of the immutable data types (numbers, strings, tuples).

Let's see an example:

Question : Extract the value associated with the key color and assign it to variable color

```
info = {'personal_data':
         {'name': 'Lauren',
          'age': 20,
          'major': 'Information Science',
          'physical_features':
             {'color': {'eye': 'blue',
                        'hair': 'brown'},
              'height': "5'8"}
         },
        'other':
         {'favorite_colors': ['purple', 'green', 'blue'],
          'interested_in': ['social media', 'intellectual property',
                            'copyright', 'music', 'books']
         }
       }

color=info['personal_data']['physical_features']['color']
print(color)
```

```
{'eye': 'blue', 'hair': 'brown'}
```

As you can see in above example how we found the first key and then the sub key and so on.

Processing JSON results

Well, with the emergence of MicroServices, DevOps, the use of JSON has increased considerably so it becomes a must to understand how can we handle JSON with Python.

JSON stands for JavaScript Object Notation. It looks a lot like the representation of nested dictionaries and lists in python when we write them out as literals in a program, but with a few small differences. When your program receives a JSON-formatted string, generally you will want to convert it into a python object, a list or a dictionary.

Python provides us a **module** for doing this **called json**. We will be using **two functions** in this module, **loads** and **dumps**.

json.loads() takes a **string** as input and **produces a python object** (a dictionary or a list) as output.

Consider, for example, some data that we might get from Apple's iTunes, in the JSON format:

a_string = '{\n "resultCount":25,\n "results": [\n{"wrapperType":"track", "collectionId":10892}]}'

```
#remember to import this module always whenever you have JSON files/format to handle
#you can also import only functions that you will be using
#like here we are using loads & dumps so do
#from json import loads,dumps
#the above import should work fine too

import json
#this is a sample string which we will be to convert into dictionary
a_string = '{\n "resultCount":25,\n "results": [\n{"wrapperType":"track", "collectionId":1089

#loads means load string which takes string as the input
#and converts from json to dictionary
dict=json.loads(a_string)

#this would print the converted dictionary
print(dict)

#this would give the value of key resultCount
print(dict['resultCount'])

#this would give an error because a_string is a string, supports indexing and operations like
#slicing
print(a_string['resultCount'])

```

```
{'resultCount': 25, 'results': [{'wrapperType': 'track', 'collectionId': 10892}]}
25

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-93-2a56e232e195> in <module>
     21 #this would give an error because a_string is a string, supports indexing and operations
     22 #slicing
---> 23 print(a_string['resultCount'])

TypeError: string indices must be integers
```

let's talk about other function named **dumps**. It does the **inverse of loads**.

- It takes a python object, typically a dictionary or a list, and returns a string, in JSON format.
- It has a few other parameters. **Two useful parameters** are **sort_keys** and **indent**.
- When the value **True** is passed for the **sort_keys parameter**, the keys of dictionaries are output in alphabetic order with their values.
- The **indent parameter** expects an **integer**. When it is provided, dumps generates a string suitable for displaying to people, with newlines and indentation for nested lists or dictionaries.

Let's see an example:

```
#as we know json module is important to work with JSON
import json

#we are defining a function which we will take dictionary and return
#a JSON formatted output
def dict_to_json(dict):
    return json.dumps(dict, sort_keys=True, indent=4)
#we can also write using lambda and it will return the same
j_son=lambda d: json.dumps(d,sort_keys=True,indent=4)

dict = {'key1': {'c': True, 'a': 90, '5': 50}, 'key2':{'b': 3, 'c': "yes"}}

print(dict)
print('--------')
print(dict_to_json(dict))
print(j_son(dict))

```

```
{'key1': {'c': True, 'a': 90, '5': 50}, 'key2': {'b': 3, 'c': 'yes'}}
--------
{
    "key1": {
        "5": 50,
        "a": 90,
        "c": true
    },
    "key2": {
        "b": 3,
        "c": "yes"
    }
}
```

Because we can only write strings into a file, if we wanted to convert a dictionary d into a json-formatted string so that we could store it in a file, what would we use?

A. json.loads(d)
B. json.dumps(d)
C.d.json()

Say we had a JSON string in the following format. How would you convert it so that it is a python list?

entertainment = """[{"Library Data": {"count": 3500, "rows": 10, "locations": 3}}, {"Movie Theater Data": {"count": 8, "rows": 25, "locations": 2}}]"""

A. entertainment.json()
B. json.dumps(entertainment)
C. json.loads(entertainment)

Nested Iteration

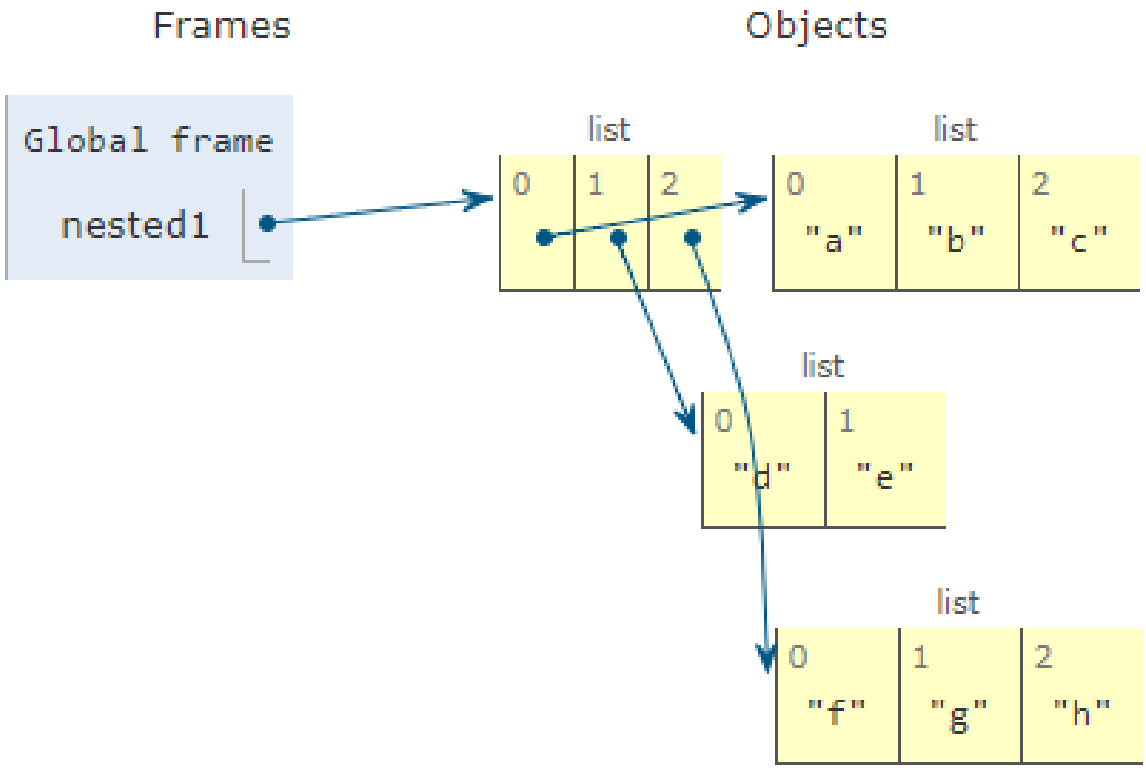
We have been using iterations in terms of for/while loop a lot of times by now, but as we you grow and become a professional in Python, we will have several nested data structures, especially lists and/or dictionaries, and you will frequently need nested for loops to traverse them.

```
1  nested1 = [['a', 'b', 'c'],['d', 'e'],['f', 'g', 'h']]
2
3  for i in nested1:
4      print('level 1 value of current element from list is ',i)
5      index=0
6      for x in i:
7          print('level 2 at index {} is {}'.format(index,x))
8          index +=1
```

```
level 1 value of current element from list is  ['a', 'b', 'c']
level 2 at index 0 is a
level 2 at index 1 is b
level 2 at index 2 is c
level 1 value of current element from list is  ['d', 'e']
level 2 at index 0 is d
level 2 at index 1 is e
level 1 value of current element from list is  ['f', 'g', 'h']
level 2 at index 0 is f
level 2 at index 1 is g
level 2 at index 2 is h
```

what's happening in above program?

- Line 3 executes once for each top-level list, three times in all. With each sub-list, line 5 executes once for each item in the sub-list.



Let's try to solve another problem:

You have a list of lists that contain information about people. Write code to create a new list that contains every person's last name, and save that list as last_names

info = [['Tina', 'Turner', 1939, 'singer'], ['Matt', 'Damon', 1970, 'actor'], ['Kristen', 'Wiig', 1973, 'comedian'], ['Michael', 'Phelps', 1985, 'swimmer'], ['Barack', 'Obama', 1961, 'president']]

We can solve this in 2 ways:

- with 1 iteration
- with nested iteration

```
info = [['Tina', 'Turner', 1939, 'singer'],
        ['Matt', 'Damon', 1970, 'actor'],
        ['Kristen', 'Wiig', 1973, 'comedian'],
        ['Michael', 'Phelps', 1985, 'swimmer'],
        ['Barack', 'Obama', 1961, 'president']]
#FIRST WAY
last_names=[]
for last_name in info:
    #we know from the list that the 2nd element
    #is the last_name so we can directly consider
    #1st index as the last_name
    last_names.append(last_name[1])

print(last_names)

#SECOND WAY
last_names=[]
for lst in info:
    count=0
    #here as well we are using iterations 2ce but
    #the logic stays same that 2nd element is the
    #last_name
    for last_nm in lst:
        if count== 1:
            last_names.append(lst[count])
            break
        count += 1

print(last_names)


```

```
['Turner', 'Damon', 'Wiig', 'Phelps', 'Obama']
['Turner', 'Damon', 'Wiig', 'Phelps', 'Obama']
```

you can understand how we can use nested iteration to solve the problem.

Let's see one more :

we have a list of lists named L. Use nested iteration to save every string containing “b” into a new list named b_strings.

L = [['apples', 'bananas', 'oranges', 'blueberries', 'lemons'], ['carrots', 'peas', 'cucumbers', 'green beans'], ['root beer', 'smoothies', 'cranberry juice']]

there's a lot of ways that you can go wrong when you start writing code to solve this. Let's see the first way

```
L = [['apples', 'bananas', 'oranges', 'blueberries', 'lemons'],
     ['carrots', 'peas', 'cucumbers', 'green beans'],
     ['root beer', 'smoothies', 'cranberry juice']]

b_strings=[]
for fruits in L:
    for b_fruits in fruits:
        if 'b' in b_fruits:
            b_strings.append(b_fruits)
print(b_strings)
```

```
['bananas', 'blueberries', 'cucumbers', 'green beans', 'root beer', 'cranberry juice']
```

We remembered python Identity Operator **In** and we used it as is but what if when you don't remember it and try to solve the problem in some other way?

```
L = [['apples', 'bananas', 'oranges', 'blueberries', 'lemons'],
     ['carrots', 'peas', 'cucumbers', 'green beans'],
     ['root beer', 'smoothies', 'cranberry juice']]

b_strings=[]
for fruits in L:
    for b_fruits in fruits:
        for b_char in b_fruits:

            if b_char == 'b':
                b_strings.append(b_fruits)

print(b_strings)
```

```
['bananas', 'blueberries', 'blueberries', 'cucumbers', 'green beans', 'root beer', 'cranberry juice']
```

In above code we have used 3layers of iterations to find the fruits which contains b, but wait, we can see blueberries twice :O :O

Something is wrong!!!

Blueberries contains 2 b's so in the 3rd loop it entered twice for blueberries, inorder to fix it we would need to use???'

```
L = [['apples', 'bananas', 'oranges', 'blueberries', 'lemons'],
     ['carrots', 'peas', 'cucumbers', 'green beans'],
     ['root beer', 'smoothies', 'cranberry juice']]

b_strings=[]
for fruits in L:
    for b_fruits in fruits:
        for b_char in b_fruits:

            if b_char == 'b':
                b_strings.append(b_fruits)
                break

print(b_strings)
```

```
['bananas', 'blueberries', 'cucumbers', 'green beans', 'root beer', 'cranberry juice']
```

Structuring Nested Data

We far all the nested lists that we saw have seen structure consistent across each level. Meaning list[0], list[1],list[2]... all were sub-list but what if you find a case were have of the elements are numbers or single characters? The nested iteration should fail in that case, isn't it?

Let's see similar scenario

```
nested1 = [1, 2, ['a', 'b', 'c'],['d', 'e'],['f', 'g', 'h']]
for x in nested1:
    print("level1: ")
    for y in x:
        print("level2: {}".format(y))

```

```
level1:
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-1ea8435085b2> in <module>
      2 for x in nested1:
      3     print("level1: ")
----> 4     for y in x:
      5         print("level2: {}".format(y))

TypeError: 'int' object is not iterable
```

We got an error at the very first iteration because the very first element is an integer and we cannot loop on an int object.

How do we fix this?
We need to remember conditional statements will always come in handy in these situations

```
nested1 = [1, 2, ['a', 'b', 'c'],['d', 'e'],['f', 'g', 'h']]
for x in nested1:
    print("level1: ")
    if type(x) is list:
        for y in x:
            print("level2: {}".format(y))
    else:
        print(x)

```

```
level1:
1
level1:
2
level1:
level2: a
level2: b
level2: c
```

- we used if else statement and type function to get rid of this.

Can you think of any other solution?
If yes, try to solve it in your own way.

Deep and Shallow Copies

This topic would be confusing stay focused.

Earlier when we discussed cloning and aliasing lists we had mentioned that simply cloning a list using [:] would take care of any issues with having two lists unintentionally connected to each other.

That was definitely true for making shallow copies (copying a list at the highest level), but as we get into nested data, and nested lists in particular, the rules become a bit more complicated.

We can have second-level aliasing in these cases, which means we need to make deep copies.

When you copy a nested list, you do not also get copies of the internal lists. This means that if you perform a mutation operation on one of the original sublists, the copied version will also change.

We can see this happen in the following nested list, which only has two levels. Let's go through a simple example:

```
original = [['dogs', 'puppies'], ['cats', "kittens"]]
copied_version = original[:]
copy_version=original
copy_copy=original.copy()

print("--------different copied version -----------")
print('copied using slice operator [:] \n',copied_version)
print('Direct assigning with the original list \n',copy_version)
print('Using the copy operator \n',copy_copy)

print("\n--------if variables are pointing to same list -----------")
print('copied from slice is pointing to same list? =',copied_version is original)
print('Directly assigned variable is pointing to same list? =',copy_version is original)
print('Copied using copy operator is pointing to same list? =',copy_copy is original)

print("\n--------if variables are having same data as original_list -----------")
print('Copied from slice = ',copied_version == original)
print('Using assignment operator = ',copy_version == original)
print('Using copy operator = ',copy_copy == original)

original[0].append(["canines"])

print("\n-------- original_list after appending in 0th index sub -list now -----------")
print(original)

print("\n-------- Now look at the copied version -----------")
print('copied using slice now = \n',copied_version)
print('var created by assingment opr now = \n',copy_version)
print('copied using copy fn now = \n',copy_copy)

original.append(["canines"])
print("\n-------- original_list after appending a new element  -----------")
print(original)

print("\n-------- Now look at the copied version -----------")
print('copied using slice now = \n',copied_version)
print('var created by assingment opr now = \n',copy_version)
print('copied using copy fn now = \n',copy_copy)
```

```
--------different copied version -----------
copied using slice operator [:]
 [['dogs', 'puppies'], ['cats', 'kittens']]
Direct assigning with the original list
 [['dogs', 'puppies'], ['cats', 'kittens']]
Using the copy operator
 [['dogs', 'puppies'], ['cats', 'kittens']]
```

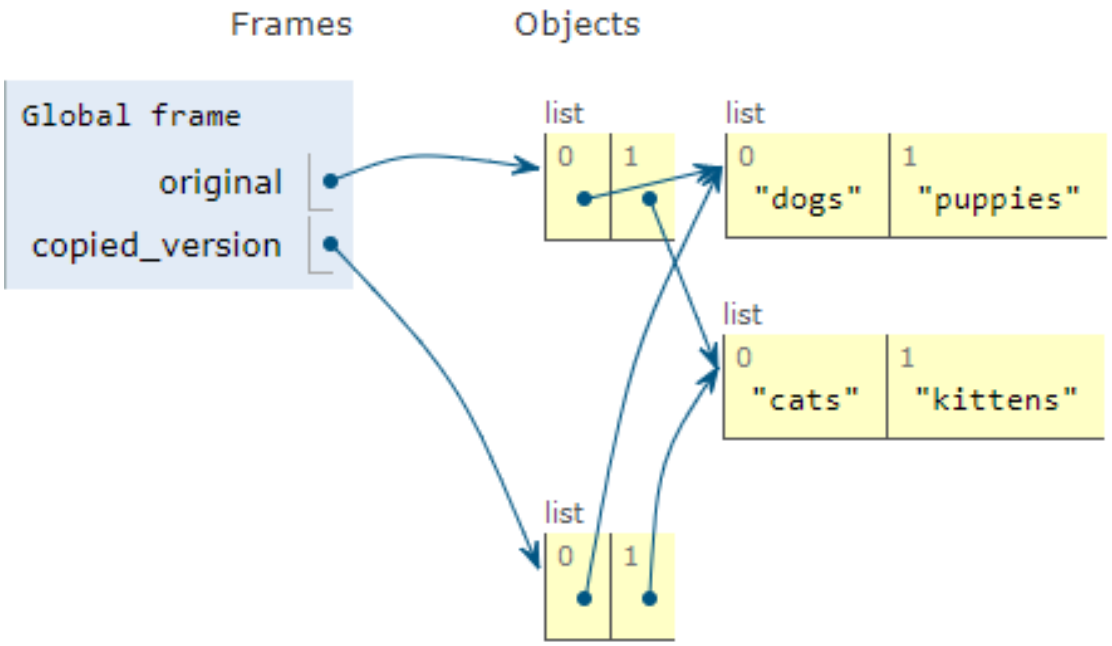
```
--------if variables are pointing to same list -----------
copied from slice is pointing to same list? = False
Directly assigned variable is pointing to same list? = True
Copied using copy operator is pointing to same list? = False
```

```
--------if variables are having same data as original_list -----------
Copied from slice =  True
Using assignment operator =  True
Using copy operator =  True
```

```
-------- original_list after appending in 0th index sub -list now -----------
[['dogs', 'puppies', ['canines']], ['cats', 'kittens']]

-------- Now look at the copied version -----------
copied using slice now =
 [['dogs', 'puppies', ['canines']], ['cats', 'kittens']]
var created by assingment opr now =
 [['dogs', 'puppies', ['canines']], ['cats', 'kittens']]
copied using copy fn now =
 [['dogs', 'puppies', ['canines']], ['cats', 'kittens']]
```

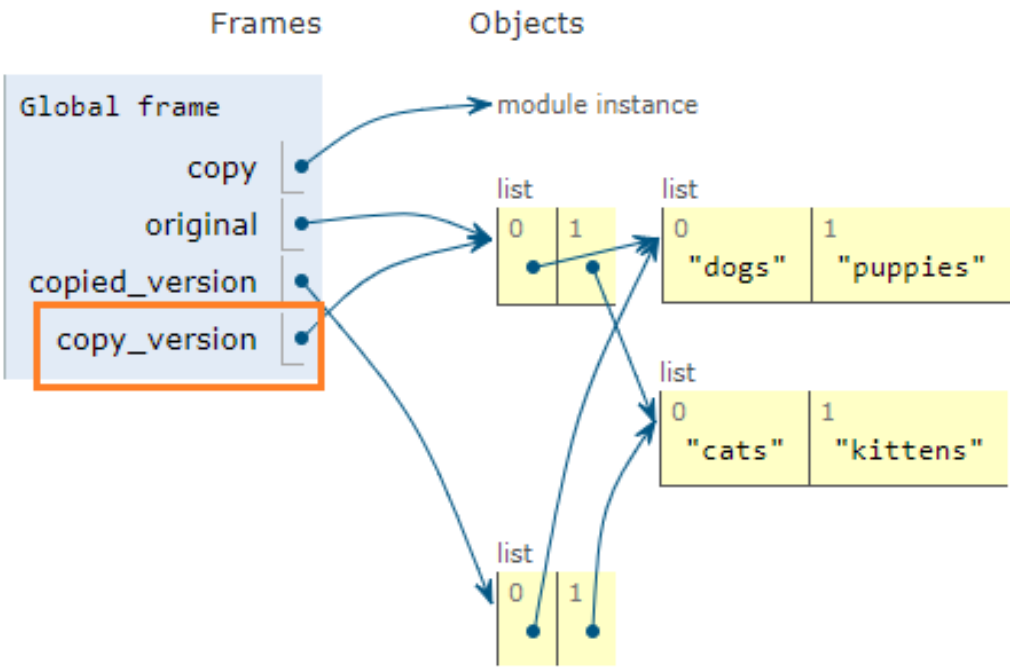
```
-------- original_list after appending a new element  -----------
[['dogs', 'puppies', ['canines']], ['cats', 'kittens'], ['canines']]

-------- Now look at the copied version -----------
copied using slice now =
 [['dogs', 'puppies', ['canines']], ['cats', 'kittens']]
var created by assingment opr now =
 [['dogs', 'puppies', ['canines']], ['cats', 'kittens'], ['canines']]
copied using copy fn now =
 [['dogs', 'puppies', ['canines']], ['cats', 'kittens']]
```

We saw a lot of things happening in the above code and output. Let us try to see what happened when we created a variable using slicing operator:

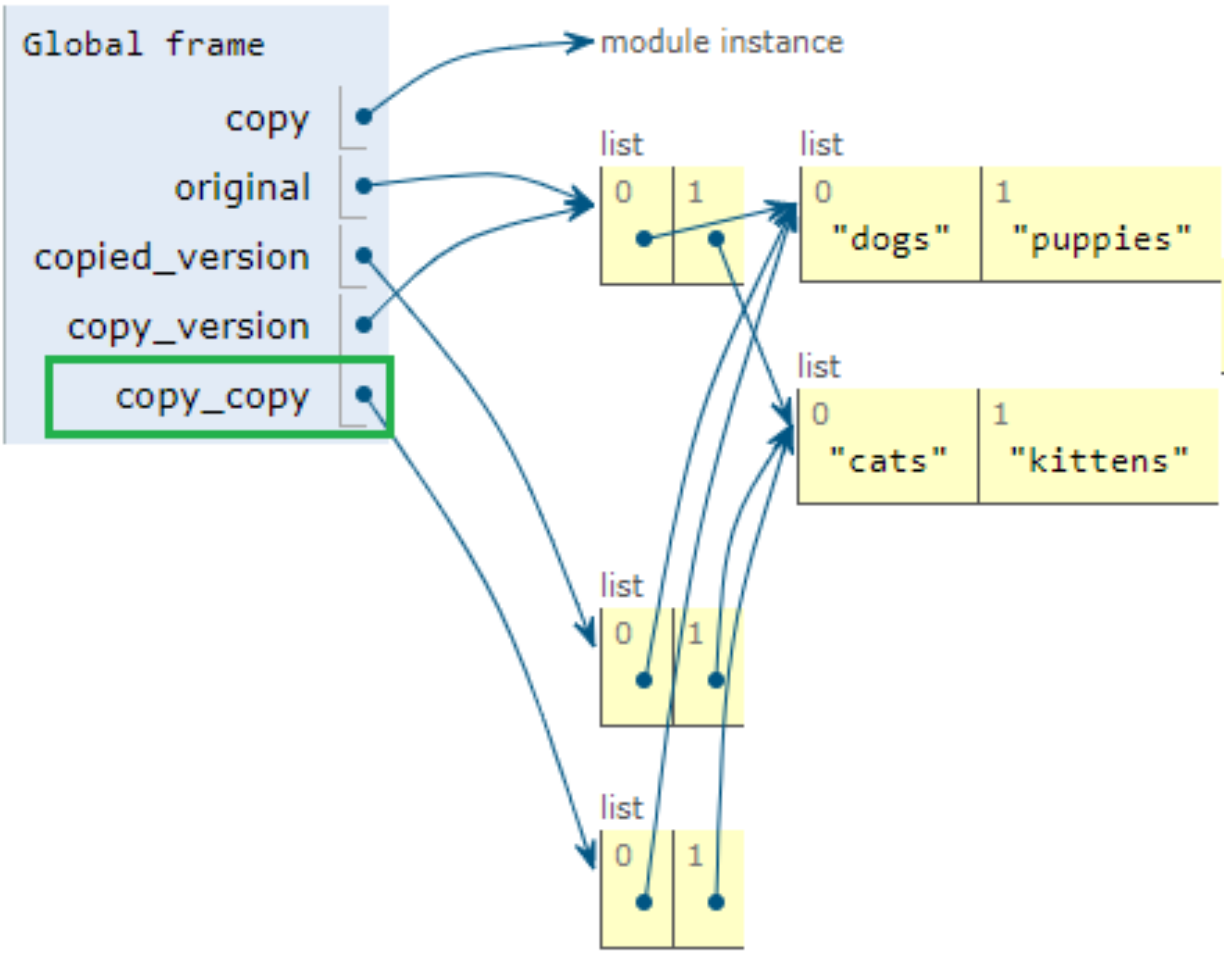


It created a new list, but it did not copy the sublists, hence as you see in the above simulation picture, it created a new object but the internal lists are still pointing to the sub_lists of original list.

For list created using assignment operator , well it is anyway going to point at the exact same list



Copy function as it copied earlier with normal list, fails with nested lists in deep copying too and ends up performing Shallow Copy same as slice operator



What can we do for performing a deep copy ?

We would need to copy sublist again inside the copy list using for loop. let's try to note it down as algorithm.

- create empty outer list
- for loop to open the first element of list
  - create inner empty list
  - nested for loop to copy all sub_lists
  - append the values in the inner list
- copy the inner list to outer list

```python
original = [['dogs', 'puppies'], ['cats', "kittens"]]
copy_outer_list=[]
for inner_list in original:
    copy_inner_list=[]
    for dump_list in inner_list:
        copy_inner_list.append(dump_list)

    copy_outer_list.append(copy_inner_list)
print(copy_outer_list)


original[0].append(['canines'])

print("\n-------- original_list after appending a element in sub_list")
print(original)

print("\n-------- Now look at the deep copied version -----------")
print(copy_outer_list)

original.append(["canines"])
print("\n-------- original_list after appending a new element  -----------")
print(original)

print("\n-------- Now look at the deep copied version -----------")
print(copy_outer_list)


```

```
[['dogs', 'puppies'], ['cats', 'kittens']]

-------- original_list after appending a element in sub_list
[['dogs', 'puppies', ['canines']], ['cats', 'kittens']]

-------- Now look at the deep copied version -----------
[['dogs', 'puppies'], ['cats', 'kittens']]

-------- original_list after appending a new element  -----------
[['dogs', 'puppies', ['canines']], ['cats', 'kittens'], ['canines']]

-------- Now look at the deep copied version -----------
[['dogs', 'puppies'], ['cats', 'kittens']]
```

we successfully deep copied a nested list using nested iterations. But we can do it without nested for loops too, lets see how

```python
original = [['dogs', 'puppies'], ['cats', "kittens"]]
copy_outer_list=[]
for inner_list in original:
    copy_inner_list= inner_list[:]
    copy_outer_list.append(copy_inner_list)
print(copy_outer_list)


original[0].append(['canines'])

print("\n-------- original_list after appending a element in sub_list")
print(original)

print("\n-------- Now look at the deep copied version -----------")
print(copy_outer_list)

original.append(["canines"])
print("\n-------- original_list after appending a new element  ---------
print(original)

print("\n-------- Now look at the deep copied version -----------")
print(copy_outer_list)


```

```
[['dogs', 'puppies'], ['cats', 'kittens']]

-------- original_list after appending a element in sub_list
[['dogs', 'puppies', ['canines']], ['cats', 'kittens']]

-------- Now look at the deep copied version -----------
[['dogs', 'puppies'], ['cats', 'kittens']]

-------- original_list after appending a new element  -----------
[['dogs', 'puppies', ['canines']], ['cats', 'kittens'], ['canines']]

-------- Now look at the deep copied version -----------
[['dogs', 'puppies'], ['cats', 'kittens']]
```

we are using a combination of for loop + slice operator to successfully copy the nested list, but what if we more layers of sub_list?

We cannot keep on writing for loops to copy the inner_most sub_list. So what do we do?

Remember the concept of recursion? If yes, you might know it's kind of confusing still amazing but with multiple layers of lists even recursion method would make our lives pretty tough.

Python contributors knows about it and hence we have the **module copy,** it provides a function named deepcopy which solves our problem. Let's see it in action

```
import copy

original=[['dogs', 'puppies', ['canines']], ['cats', 'kittens'], ['canines']]

shallow_copy_version = original[:]
deeply_copied_version = copy.deepcopy(original)
original.append("Hi there")
original[0].append(["marsupials"])
print("-------- Original -----------")
print(original)
print("-------- deep copy -----------")
print(deeply_copied_version)
print("-------- shallow copy -----------")
print(shallow_copy_version)

```

```
-------- Original -----------
[['dogs', 'puppies', ['canines'], ['marsupials']], ['cats', 'kittens'], ['canines'], 'Hi there']
-------- deep copy -----------
[['dogs', 'puppies', ['canines']], ['cats', 'kittens'], ['canines']]
-------- shallow copy -----------
[['dogs', 'puppies', ['canines'], ['marsupials']], ['cats', 'kittens'], ['canines']]
```

I understand this part was a little confusing and torture to brain if you have tried to understand every bits and pricks of it. I would recommend to practice programs in pythontutor.com to see the simulation of how things are working in backend to understand it better

Extracting from Nested Data

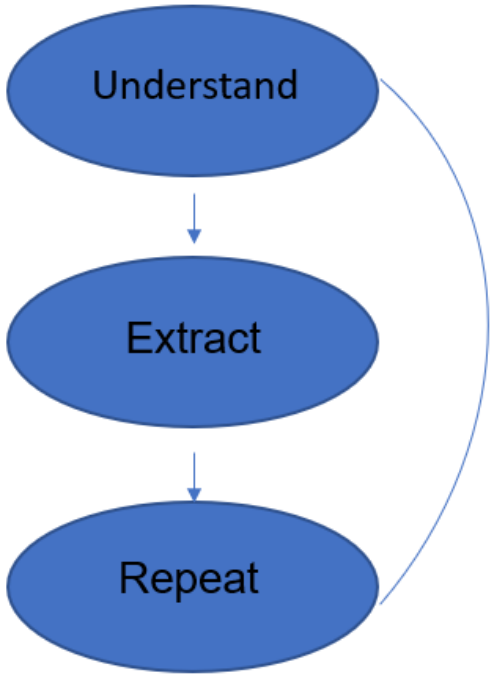
We know that JSON files could look dauting cause they carry so much data, even a single tweet that you post over twitter carries so much data, your tweet, your user details, your location, the time you posted, how many likes, comments, tweets, etc.

It could be a nightmare if you were asked to to extract certain elements from deep inside a nested data structure in JSON format. I can promise sleepless nights is what you will have.

In principle, there's nothing more difficult about pulling something out from deep inside a nested data structure: with lists, you use [] to index or a for loop to get them them all; with dictionaries, you get the value associated with a particular key using [] or iterate through all the keys, accessing the value for each. But it's easy to get lost in the process and think you've extracted something different than you really have.

We need to focus on following concepts:

- Understand the nested data object.
- Extract one object at the next level down.
- Repeat the process with the extracted object



We will use a JSON file present in below location:
https://drive.google.com/file/d/1hhX8sEM_tJjFLLmgV7g1eU5lbrCtAHMJ/view?usp=sharing

Try to download it to understand further problems we would be doing:
''

Step 1 :Understand

At any level of extraction process, the first task is to make sure you understand the current object you have extracted.

1. If the entire object is small enough you may print it to see how it looks and then get an idea how will you extract. But what if the object is a bit larger, then you would have to use dumps(dump string) with indentation as we used above while learning about it, which will show us the nesting of the data.

import json
print(json.dumps(sample,indent=2))

This would give you the sample dictionary in JSON format with indentation 2.

2. If printing the entire object gives you something that's too unwieldy, you have other options for making sense of it.
    - Copy and paste it to a site like https://jsoneditoronline.org/ which will let you explore and collapse levels
    - Print the type of the object.

- **If it's a dictionary:**
  - print the keys

- **If it's a list:**
  - print its length
  - print the type of the first item
  - print the first item if it's of manageable size

```
import json #mandatory import
#we will convert the given dictionary into
#a JSON format string until 100letters
#we can use slice operator cause the ouptut
#is supposed to be a string
print(json.dumps(res, indent=2)[:100])
print('\n--- print the type of res-----')
print(type(res))
print('\n--- print the keys of dict res-----')
print(res.keys())
```

```
{
  "search_metadata": {
    "count": 3,
    "completed_in": 0.015,
    "max_id_str": "5366245192855

--- print the type of res-----
<class 'dict'>

--- print the keys of dict res-----
dict_keys(['search_metadata', 'statuses'])
```

Step 2: Extract

In the extraction phase, you will be diving one level deeper into the nested data.

1. If it's a dictionary, figure out which key has the value you're looking for, and get its value. For example: in above eg we saw res has ['search_metadata','statuses']

we can use any of the keys based on your requirement to enter what is inside that key. like res2=res['status]

2. If the key's value is a list , you will typically be wanting to do something with each of the items(eg. extracting something from each, and accumulating them in a list), you can use for loop over the iterable res2

for res2 in res:

3. But remember the value could be too big to understand, so in order to understand what value does the list has, you may need to start small i.e take one element of the list and simulate over the program.

for res2 in res[:1]

```
1 res2=res['statuses']
2 print(type(res2))
```

```
<class 'list'>
```

Step 3: Repeat

We have cleared the first lever and now have to deal with inner list, let's call it Level 2, we will again do the same thing, understand --> Extract --> repeat (if needed)

As we know the value of statuses is a list, we need to perform these 3 steps

- print its length
- print the type of the first item
- print the first item if it’s of manageable size

```
print(type(res2))
print(len(res2))
res3= res2[0]
print(json.dumps(res3,indent=2)[:100])
print(type(res3))
print(res3.keys())

```

```
<class 'list'>
3
{
  "contributors": null,
  "truncated": false,
  "text": "RT @mikeweber25: I'm decommiting from the
<class 'dict'>
dict_keys(['contributors', 'truncated', 'text', 'in_reply
nates', 'entities', 'in_reply_to_screen_name', 'in_reply_
s', 'user', 'geo', 'in_reply_to_user_id_str', 'lang', 'cr
```

until now we know that in first level we had a dictionary which had 2 keys , we entered one key's value and checked it's type which was found to be a list. Now in above query we see that list also has 3 dictionaries and every dictionary has several keys.

Let's try to further dig inside these dictionaries and try to find out some values

```
res4=res3['user']
#print(res4)
print(type(res4))
print(res4['screen_name'],res4['created_at'])
```

```
<class 'dict'>
31brooks_ Wed Apr 09 14:34:41 +0000 2014
```

Now we know how can we extract the user details. So final code would become very small and will look like

```
for res3 in res['statuses']:
    print(res3['user']['screen_name'], res3['user']['created_at'])

```

```
31brooks_ Wed Apr 09 14:34:41 +0000 2014
froyoho Thu Jan 14 21:37:54 +0000 2010
MDuncan95814 Tue Sep 11 21:02:09 +0000 2012
```

Even with this compact code, we can still count off how many levels of nesting we have extracted from, in this case four.

- res[‘statuses’] says we have descended one level (in a dictionary).
- for res3 in… says we are have descended another level (in a list
- [‘user’] is descending one more level
- [‘screen_name’] is descending one more level.

Question:
Iterate through the list so that if the character ‘m’ is in the string, then it should be added to a new list called m_list. Hint: Because this isn’t just a list of lists, think about what type of object you want your data to be stored in. Conditionals may help you.

```
d = ['good morning', 'hello', 'chair', 'python',
    ['music', 'flowers', 'facebook', 'instagram', 'snapchat',
     ['On my Own', 'monster', 'Words dont come so easily', 'lead me right']],
    'Stressed Out', 'Pauver Coeur', 'Reach for Tomorrow', 'mariners song', 'Wonder sleeps here']
```

```
m_list=[]
for lst in d:
  if type(lst) is list:
    for sub_lst in lst:
      if type(sub_lst) is list:
        for sub_sub_lst in sub_lst:
          if 'm' in sub_sub_lst:
            m_list.append(sub_sub_lst)
          else:
            pass
      else:
        if 'm' in sub_lst:
          m_list.append(sub_lst)

  else:
    if 'm' in lst:
      m_list.append(lst)

print(m_list)
```

Question2:

The nested dictionary, pokemon, shows the number of various Pokemon that each person has caught while playing Pokemon Go. Find the total number of rattatas, dittos, and pidgeys caught and assign to the variables r, d, and p respectively. Do not hardcode. Note: Be aware that not every trainer has caught a ditto.

```
pokemon = {'Trainer1':
                    {'normal': {'rattatas':15, 'eevees': 2, 'ditto':1}, 'water':
                     {'magikarps':3},  'flying': {'zubats':8, 'pidgey': 12}},
      'Trainer2':
      {'normal': {'rattatas':25, 'eevees': 1}, 'water': {'magikarps':7}, 'flying':
{'zubats':3, 'pidgey': 15}},
      'Trainer3':
      {'normal': {'rattatas':10, 'eevees': 3, 'ditto':2}, 'water': {'magikarps':2}, 'flying':
{'zubats':3, 'pidgey': 20}},
      'Trainer4':
      {'normal': {'rattatas':17, 'eevees': 1}, 'water': {'magikarps':9}, 'flying':
{'zubats':12, 'pidgey': 14}}}
```

Below, we have provided a nested list called big_list. Use nested iteration to create a dictionary, word_counts, that contains all the words in big_list as keys, and the number of times they occur as values.

```
big_list = [[['one', 'two'], ['seven', 'eight']], [['nine', 'four'], ['three', 'one']], [['two', 'eight'], ['seven', 'four']], [['five', 'one'], ['four', 'two']], [['six', 'eight'], ['two', 'seven']], [['three', 'five'], ['one', 'six']], [['nine', 'eight'], ['five', 'four']], [['six', 'three'], ['four', 'seven']]]



word_counts={}

for lst in big_list:
    if type(lst) is list:
        for sub_lst in lst:
            if type(sub_lst) is list:
                for sub_sub in sub_lst:
                    if sub_sub not in word_counts:
                        word_counts[sub_sub]=0
                    word_counts[sub_sub]=word_counts[sub_sub] + 1
print(word_counts)
```

Provided is a dictionary that contains pokemon go player data, where each player reveals the amount of candy each of their pokemon have. If you pooled all the data together, which pokemon has the highest number of candy? Assign that pokemon to the variable most_common_pokemon.

```
pokemon_go_data = {'bentspoon':
                    {'Rattata': 203, 'Pidgey': 120,
                     'Drowzee': 89, 'Squirtle': 35,
                     'Pikachu': 3, 'Eevee': 34,
                     'Magikarp': 300, 'Paras': 38},
                   'Laurne':
                   {'Pidgey': 169, 'Rattata': 245, 'Squirtle': 9, 'Caterpie': 38, 'Weedle': 97, 'Pikachu': 6, 'Nidoran': 44, 'Clefairy': 15, 'Zubat': 79, 'Dratini': 4},
                   'picklejarlid':
                   {'Rattata': 32, 'Drowzee': 15, 'Nidoran': 4, 'Bulbasaur': 3, 'Pidgey': 56, 'Weedle': 21, 'Oddish': 18, 'Magmar': 6, 'Spearow': 14}}
```

```
dict={}
for pikachu in pokemon_go_data:
  for key_val in pokemon_go_data[pikachu]:
    if key_val not in dict:
      dict[key_val]=0
    dict[key_val]=dict[key_val] + pokemon_go_data[pikachu][key_val]
most_common_pokemon=sorted(dict, key=lambda e:dict[e], reverse=True)[0]
print(most_common_pokemon)
```

Given below is a list of lists of athletes. Create a list, t, that saves only the athlete's name if it contains the letter "t". If it does not contain the letter "t", save the athlete name into list other.

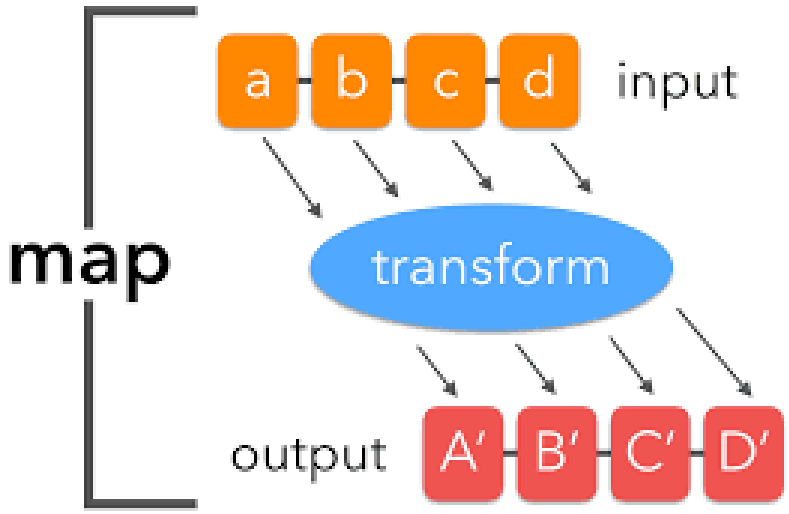
```
athletes = [['Phelps', 'Lochte', 'Schooling', 'Ledecky', 'Franklin'], ['Felix', 'Bolt', 'Gardner', 'Eaton'],
['Biles', 'Douglas', 'Hamm', 'Raisman', 'Mikulak', 'Dalton']]


t=[]
other=[]
for lst in athletes:
  for names in lst:
    if 't' in names:
      t.append(names)
    else:
      other.append(names)
```

Map, Filter, List Comprehensions, and Zip

Wowwww... long journey so far since the time we started using accumulator patterns for **lists , producing another list** from it that contained either a subset of the items or a transformed version of each item. When **each item is transformed** we say that the **operation is a mapping**, or just a map of the original list. When **some items** are **omitted**, we **call it a filter**.

We have built in functions of map and filter which will help us in mapping and filtering operations.

Map(transformer)



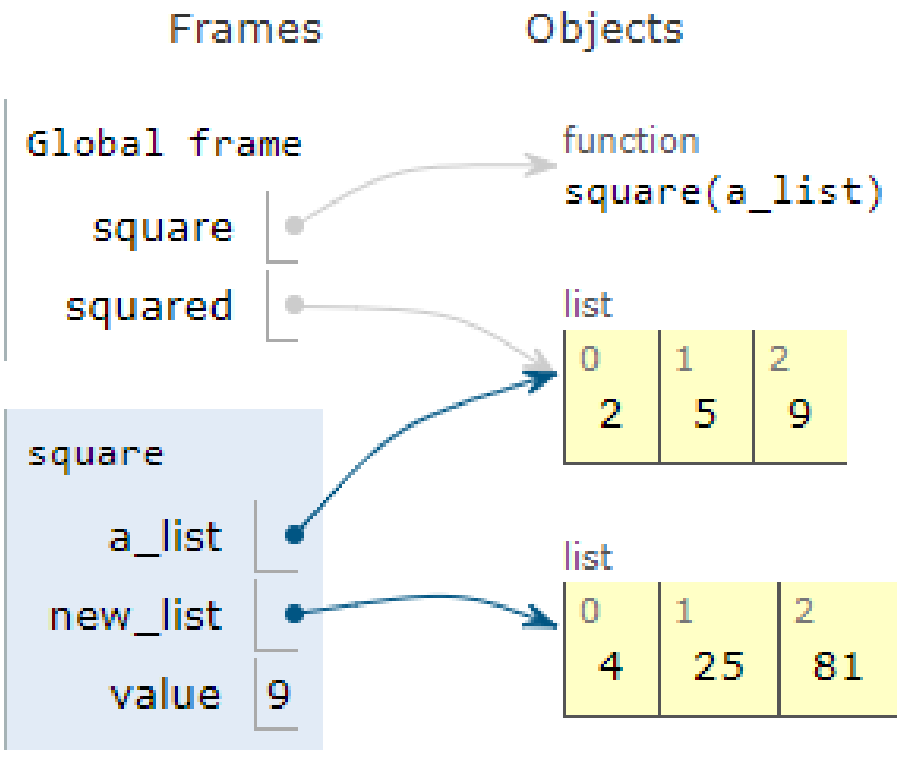
Let's try to understand Map via a program, how we can convert the traditional (using traditional just for an impact :D) way of using for loop into a map

```
1  def square(a_list):
2      new_list=[]
3      for value in a_list:
4          new_list.append(value * value)
5
6      return new_list
7  squared = [2, 5, 9]
8  print(squared)
9  print(square(squared))
10
```

```
[2, 5, 9]
[4, 25, 81]
```

The square function is an example of the accumulator pattern, in particular the mapping pattern.

- On line 2, new_list is initialized. On line 4, the squared value for the current item is produced and is appended to the list at the same time i.e. we're accumulating.
- Line 6 returns the final new _list after accumulating the squared values of numbers in the list.



This pattern of computation is so common that python came up with a more general way to do mappings, which we know by now is map function, that makes it more clear what the overall structure of the computation is.

- map takes two arguments, a function and a sequence.
- The function is the mapper that transforms items.
- It is automatically applied to each item in the sequence.
- You don't have to initialize an accumulator or iterate with a for loop at all, it will be handled internally from the map function.

Another example of why Python is loved by all, cause you need to focus only on the part which is important and will make your coding easy.

Technically, in a proper Python 3 interpreter, the ***map function*** *produces* ***an "iterator", which is like*** *a list but produces the items as they are needed. Most places in Python where you can use a list (e.g., in a for loop) you can use an "iterator" as if it was actually a list. So you probably won't ever notice the difference.If you ever really need a list, you can explicitly turn the output of map into a list: list(map(...)).*

you remember **sorted function ?? rings a bell?** 

As we did when passing a function as a parameter to the sorted function, we can specify a function to pass to map either by referring to a function by name, or by providing a lambda expression.

the syntax looks like :

output= map(function_name, iterable_list)

where you nee to define function

.

```
1 def square(value):
2     return value * value
3
4 squared=[2, 5, 9]
5 print(list(map(square, squared)))
6
```

```
[4, 25, 81]
```

As we saw the saw the complications reduced drastically, by using the map function.
Also it's no longer necessary to even define the square function we can use lambda.

Let's see how

```
1 squared=[2, 5, 9]
2 print(list(map((lambda x: x * x) , squared)))
3
```

```
[4, 25, 81]
```

Questions:
1. Using map, create a list assigned to the variable greeting_doubled that doubles each element in the list lst.
lst = [["hi", "bye"], "hello", "goodbye", [9, 2], 4]

2. Below, we have provided a list of strings called abbrevs. Use map to produce a new list called abbrevs_upper that contains all the same strings in upper case.

abbrevs = ["usa", "esp", "chn", "jpn", "mex", "can", "rus", "rsa", "jam"]

Filter(filtration function)

You know now what map does, it is a transformer which takes every element as an input gives it to a function performs some calculation and returns its new value, we used it to replace the accumulation pattern whenever we will need.

There is another pattern where we provided a list as an input and kept only those items that met certain criteria. This is where we would replace again the so called traditional use of loops with new function called filter. we pass a filtration function which will return only True or False and based on that we have only the required output

Traditional way:

```
def dev_even(lst):
    lst1=[]
    for num in lst:
        if num%2 == 0:
            lst1.append(num)
    return lst1


even_od=[1,2,3,5,6,7,8,9]
print(dev_even(even_od))

```

```
[2, 6, 8]
```

easy but still lengthy , a little complex and yes time consuming. Python is used to increase the speed of the processes and this way isn't going to help us. Let's try now with the filter way

```
1 print(list(filter((lambda x: x%2 == 0),[1,2,3,5,6,7,8,9])))
2
```

```
[2, 6, 8]
```

python will keep giving you the adrenaline flow in your blood all the time with such amazing implementations to make your life easy.

1. Write code to assign to the variable filter_testing all the elements in lst_check that have a w in them using filter.

lst_check = ['plums', 'watermelon', 'kiwi', 'strawberries', 'blueberries', 'peaches', 'apples', 'mangos', 'papaya']

2. Using filter, filter lst so that it only contains words containing the letter “o”. Assign to variable lst2. Do not hardcode this.

lst = ["witch", "halloween", "pumpkin", "cat", "candy", "wagon", "moon"]

List Comprehensions

We learned map and filter, found it useful and less complex!!!

But what if i say, you can make your coding even simpler than what you felt after using map/filter!! Confused, i too was when i learned it for the first time. List comprehensions is the alternative way to do map & filter.

List comprehensions are concise ways to create lists from other lists. The general syntax is:

[<transformer_expression> **for** <loop_var> **in** <sequence> **if** <filtration_expression>]

where the if clause is optional.

If you look closely, you will realize there is a catch in this syntax. It has both expressions, transformer & filtration meaning it would be doing what map and filter did separately. Lets' see how

```
1 things = [2, 5, 9]
2
3 yourlist = [value * 2 for value in things]
4
5 print(yourlist)
```

```
[4, 10, 18]
```

breaking the above syntax with values:
<transformer expression> = value * 2
for loop = for value in things
<filtration expression> = Nothing because we don't have to filter anything in this code.

This is an **alternative way** to perform a mapping operation. As **with map, each item in the sequence is transformed into an item in the new list**. Instead of the iteration happening automatically, however, we have adopted the syntax of the for loop which may make it easier to understand.Just as in a regular for loop, the part of the statement for value in things says to execute some code once for each item in things. Each time that code is executed, value is bound to one item from things.

- The **code** that is **executed each time is the transformer expression**, **value * 2**, **rather than** a **block** of **code indented underneath** the **for statement**.

- The **other difference** from a regular for loop is that each time the expression is evaluated, the resulting value is appended to a list. That happens automatically, without the programmer explicitly initializing an empty list or appending each item.

- The if clause of a list comprehension can be used to do a filter operation. To perform a pure filter operation, the expression can be simply the variable that is bound to each item. For example, the following list comprehension will keep only the even numbers from the original list.

```
1 def keep_evens(nums):
2     return ([num for num in nums if num%2==0])
3 print(keep_evens([3, 4, 6, 7, 0, 1]))
```

```
[4, 6, 0]
```

Same can be done with filter too

```
1  def keep_evens(nums):
2      return ([num for num in nums if num%2==0])
3
4  def keep_with_map_filter(lst):
5      filtered_lst=list(filter((lambda x: x%2==0),lst))
6      return (filtered_lst)
7
8
9  print(keep_evens([3, 4, 6, 7, 0, 1]))
10 print(keep_with_map_filter([3, 4, 6, 7, 0, 1]))
```

```
[4, 6, 0]
[4, 6, 0]
```

What is printed by the following statements?

```
alist = [4,2,8,6,5]
blist = [num*2 for num in alist if num%2==1]
print(blist)
```

3. Write code to assign to the variable compri all the values of the key name in any of the sub-dictionaries in the dictionary tester. Do this using a list comprehension.

```
tester = {'info': [{"name": "Lauren", 'class standing': 'Junior', 'major': "Information Science"}, {'name': 'Ayo', 'class standing': "Bachelor's", 'major': 'Information Science'}, {'name': 'Kathryn', 'class standing': 'Senior', 'major': 'Sociology'}, {'name': 'Nick', 'class standing': 'Junior', 'major': 'Computer Science'}, {'name': 'Gladys', 'class standing': 'Sophomore', 'major': 'History'}, {'name': 'Adam', 'major': 'Violin Performance', 'class standing': 'Senior'}]}
```

Tips: remember to see the dictionary in a better way you can use json format dumps.
ii) As this is a nested list you may have to simulate first with first level of dictionary and then sub level of dictionary

Answer:

```
import json
print(json.dumps(tester['info'], indent=4))
compri=[name['name'] for name in tester['info'] ]
```

Zip

One more common pattern with lists, besides accumulation, is to step through a pair of lists (or several lists), doing something with all of the first items, then something with all of the second items, and so on. For example, given two lists of numbers, you might like to add them up pairwise, taking [3, 4, 5] and [1, 2, 3] to yield [4, 6, 8].

```
L1 = [3, 4, 5]
L2 = [1, 2, 3]
L3 = []

for i in range(len(L1)):
    L3.append(L1[i] + L2[i])
print(L3)
```

```
[4, 6, 8]
```

looks simple? what if we can do it in 1 line? let's see how:

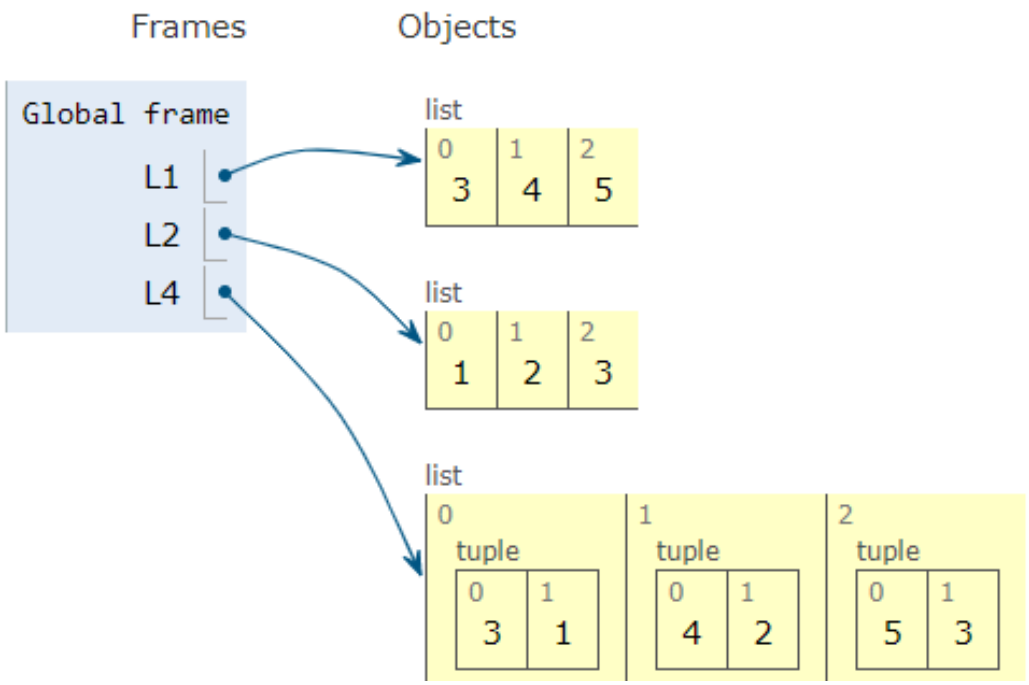
STEP 1:

```
L1 = [3, 4, 5]
L2 = [1, 2, 3]
L3 = list(zip(L1,L2))
print(L3)
```

```
[(3, 1), (4, 2), (5, 3)]
```

The **zip function** takes **multiple lists** and turns them **into a list of tuples** (actually, an iterator, but they work like lists for most practical purposes), pairing up all the first items as one tuple, all the second items as a tuple, and so on. Then we can iterate through those tuples, and perform some operation on all the first items, all the second items, and so on.

View after code visualization



STEP 2:

Here's what happens when you loop through the tuples.

```
L1 = [3, 4, 5]
L2 = [1, 2, 3]
L3 = list(zip(L1,L2))
L4=[]
for x1,x2 in L3:
    L4.append(x1 + x2)
print(L4)
```

```
[4, 6, 8]
```

STEP 3:

simplifying further and bringing it down to 1 line code.

```
L1 = [3, 4, 5]
L2 = [1, 2, 3]
L3 = [x1+x2 for x1,x2 in zip(L1,L2)]
print(L3)
```

```
[4, 6, 8]
```

```
L1 = [3, 4, 5]
L2 = [1, 2, 3]
L3 = list(map ((lambda x: x[0] + x[1]),zip(L1,L2)))
#here zip returns a list of tuple
#now lambda will consider individual gropu of tuples
#henc x will have =(3,1) (4,2) (5,3)
#and we can easily break from x --> x[0], x[1]
print(L3)

```

```
[4, 6, 8]
```

Question:
Consider a function called possible, which determines whether a word is still possible to play in a game of hangman, given the guesses that have been made and the current state of the blanked word.

```
def possible(word, blanked, guesses_made):
    if len(word) != len(blanked):
        return False
    for i in range(len(word)):
        bc = blanked[i]
        wc = word[i]
        if bc == '_' and wc in guesses_made:
            return False
        elif bc != '_' and bc != wc:
            return False
    return True

print(possible("wonderwall", "_on__r__ll", "otnqurl"))
print(possible("wonderwall", "_on__r__ll", "wotnqurl"))
```

write this program in **List comprehension**

```
def possible(word, blanked, guesses_made):
  if len(word) != len(blanked):
    return False
  for (bc, wc) in zip(blanked, word):
    if bc == '_' and wc in guesses_made:
      return False
    elif bc != '_' and bc != wc:
      return False
  return True

print(possible("wonderwall", "_on__r__ll", "otnqurl"))
print(possible("wonderwall", "_on__r__ll", "wotnqurl"))
```

```
def hangman(word,blanked,used):
    print(len(word))
    print(len(blanked))
    if len(word) != len(blanked):
        return False
    #for i in range(len(word)):
     #   print(word[i])
     #   print(blanked[i])
     #   wc=word[i]
     #   bc=blanked[i]
    for (wc,bc) in zip(word,blanked):
        print(wc,bc)
        if bc == '_' and wc in used:
            return False
        elif bc != '_' and wc != bc:
            False
    return True

print(hangman('wonderful','__nder_ul','tnqurl'))
```

```
9
9
w _
o _
n n
d d
e e
r r
f _
u u
l l
True
```

```
def possible(word, blanked, guesses_made):
  if len(word) != len(blanked):
    return False
  for (bc, wc) in zip(blanked, word):
    if bc == '_' and wc in guesses_made:
      return False
    elif bc != '_' and bc != wc:
      return False
  return True

print(possible("wonderwall", "_on__r__ll", "otnqurl"))
print(possible("wonderwall", "_on__r__ll", "wotnqurl"))
```

```
def hangman(word,blanked,used):
    print(len(word))
    print(len(blanked))
    if len(word) != len(blanked):
        return False
    #for i in range(len(word)):
     #   print(word[i])
     #   print(blanked[i])
     #   wc=word[i]
     #   bc=blanked[i]
    for (wc,bc) in zip(word,blanked):
        print(wc,bc)
        if bc == '_' and wc in used:
            return False
        elif bc != '_' and wc != bc:
            False
    return True

print(hangman('wonderful','__nder_ul','tnqurl'))
```

```
9
9
w _
o _
n n
d d
e e
r r
f _
u u
l l
True
```

Write equivalent code using map instead of the manual accumulation below and assign it to the variable test.

```
things = [3, 5, -4, 7]

accum = []
for thing in things:
    accum.append(thing+1)
print(accum)
```

Write a function called longlengths that returns the lengths of those strings that have at least 4 characters. Try it using map and filter.

Answer:

```
def longlengths(strings):
   filtered=list(filter(lambda x:len(x)>4, strings))
   print(filtered)
   return list(map(len,filtered))
```

Write a function that takes a list of numbers and returns the sum of the squares of all the numbers. Try it using map and sum.

Answer:

```
def sumSquares(L):
   return sum((map(lambda x : x*x ,L)))

nums = [3, 2, 2, -1, 1]
```

Write equivalent code using map instead of the manual accumulation below and assign it to the variable test.

```
things = [3, 5, -4, 7]

accum = []
for thing in things:
    accum.append(thing+1)
print(accum)
```

Write a function called longlengths that returns the lengths of those strings that have at least 4 characters. Try it using map and filter.

Answer:

```
def longlengths(strings):
   filtered=list(filter(lambda x:len(x)>4, strings))
   print(filtered)
   return list(map(len,filtered))
```

Write a function that takes a list of numbers and returns the sum of the squares of all the numbers. Try it using map and sum.

Answer:

```
def sumSquares(L):
   return sum((map(lambda x : x*x ,L)))

nums = [3, 2, 2, -1, 1]
```

Use zip and map or a list comprehension to make a list consisting the maximum value for each position. For L1, L2, and L3, you would end up with a list [4, 5, 3, 5].
L1 = [1, 2, 3, 4]
L2 = [4, 3, 2, 3]
L3 = [0, 5, 0, 5]

Answer:
```
maxs = [max(x1,x2,x3) for x1,x2,x3 in zip(L1,L2,L3)]
```

Write code to assign to the variable compri_sample all the values of the key name in the dictionary tester if they are Juniors. Do this using list comprehension.

```
tester = {'info': [{"name": "Lauren", 'class standing': 'Junior', 'major': "Information Science"}, {'name': 'Ayo', 'class standing': "Bachelor's", 'major': 'Information Science'}, {'name': 'Kathryn', 'class standing': 'Senior', 'major': 'Sociology'}, {'name': 'Nick', 'class standing': 'Junior', 'major': 'Computer Science'}, {'name': 'Gladys', 'class standing': 'Sophomore', 'major': 'History'}, {'name': 'Adam', 'major': 'Violin Performance', 'class standing': 'Senior'}]}
```

Answer:
```
compri_sample=[i['name'] for i in tester['info'] if 'Junior' in i.values()]
```

PART 9