ITFP - Intermediate Python - Part2

April 14, 2019

1 List comprehension

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

```
In [1]: squares = []
        for x in range(10):
            squares.append(x**2)

        print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
In [2]: squares = [x**2 for x in range(10)]
        print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprenhesion can be combined with **if** expression to build more complex lists in a concise way

```
In [3]: combs = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
        print(combs)
```

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

This is equivalent to

```
In [ ]: combs = []
        for x in [1,2,3]:
            for y in [3,1,4]:
                if x != y:
                    combs.append((x, y))
        print(combs)
```

2 Functions

Let's define a function that returns the fibonacci sequence up to a specified number.

```
In [4]: def fib(n):
            '''
            Function that return fib sequence up to N
            n = int
            '''
            a, b = 0, 1
            while a < n:
                print(a, end=' ')
                a = b
                b = a + b

In [5]: help(fib)

Help on function fib in module __main__:

fib(n)
    Function that return fib sequence up to N
    n = int



In [6]: fib(100)

0 1 2 4 8 16 32 64
```

As you can notice a function is defined simply as a block of code starting with the keywork **def**

```
def name_of_the_function(args):
    block of code
```

Function could return one or more results using the keywork **return**

```
In [7]: def inverse(num):
            return 1/num

In [8]: inverse(2)

Out[8]: 0.5
```

2.1 Default arguments

Default values for the arguments of a function can be specified in the declaration. Remember that arguments without default values have to be positioned before the other ones.

```
In [9]: def ask_ok(prompt, retries=1, reminder='Please try again!'):
            while True:
                ok = input(prompt)
                if ok in ('y', 'ye', 'yes'):
                    return True
                if ok in ('n', 'no', 'nop', 'nope'):
                    return False
                retries = retries - 1
                if retries < 0:
                    raise ValueError('invalid user response')
                print(reminder)

In [10]: ask_ok("Do you like the tutorial?", reminder="Are you sure?")

Do you like the tutorial? Yes


Are you sure?


Do you like the tutorial? y


Out[10]: True
```

2.2 Keyword arguments

You can always use the name of the arguments when you call the function, to increase readibility

```
In [11]: def do_stuff(a, b, c):
             print("a=", a, "b=", b, "c=", c,)

In [12]: do_stuff(1,2,3)
         do_stuff(1 , c=3, b=2)

a= 1 b= 2 c= 3
a= 1 b= 2 c= 3
```

Be careful! the named arguments should stay after all positional (non-named) arguments

```
In [13]: do_stuff(b=2, c=3, 1)


           File "<ipython-input-13-e27952f54ffe>", line 1
         do_stuff(b=2, c=3, 1)
                            ^
     SyntaxError: positional argument follows keyword argument
```

2.3 Return more than one element

Functions can return more than one element really easily, creating a inplicitly a **tuple** thanks to a comma.

```
In [14]: def sum_diff(a, b):
             return a+b, a-b  # note the use of the comma
```

The result is indeed a tuple.

```
In [15]: result = sum_diff(45, 23)
         print(result)
         print(result[0])
```

```
(68, 22)
68
```

```
In [16]: # You can unpack the result directly
         c, d = sum_diff(45, 23)
         print(c)
         print(d)
```

```
68
22
```

2.4 Pass functions around

As everything in Python, functions are also object, so they can be passed to other functions to create dynamic behaviours.

```
In [17]: def do_stuff(num):
             return num**2 + num - 3

         def do_more_stuff(num):
             return num - num**2

         def work_on_list(l, operation):
             newlist = []
             for elem in l:
                 newlist.append(operation(elem))
             return newlist
```

```
In [18]: l = [1, 2, 3, 4, 5, 6, 7]
         l2 = work_on_list(l, do_stuff)
         l3 = work_on_list(l, do_more_stuff)
```

```
In [19]: print(l2)
         print(l3)
```

```
[-1, 3, 9, 17, 27, 39, 53]
[0, -2, -6, -12, -20, -30, -42]
```

This is only an introduction of this kind of pattern called **functional programming** that is heavily present in Python programming. This type of operation is performed thanks to the global command **map** that we will describe later.

2.5 Variable number of arguments

If your function doesn't need a fixed number of arguments, but you don't want the user to use a list, you can use the keywork __*args__

```
In [20]: def print_stuff(text, *args):
             print(text)
             for t in args:
                 print(">>> ", t)

In [21]: print_stuff("Benvenuti a Milano-Bicocca a: ", "Marco", "Giulia", "Giuseppe", "Mara")

Benvenuti a Milano-Bicocca a:
>>>  Marco
>>>  Giulia
>>>  Giuseppe
>>>  Mara


In [22]: def print_stuff(text, *args, **kargs):
             print(text)
             for t in args:
                 print(">>> ", t)
             for k, value in kargs.items():
                 print("!!! {}:{}".format(k, value))

In [23]: print_stuff("Benvenuti a Milano-Bicocca a: ", "Marco", "Giulia","Giuseppe",
                     voto=10, giorno="giovedi")

Benvenuti a Milano-Bicocca a:
>>>  Marco
>>>  Giulia
>>>  Giuseppe
!!! voto:10
!!! giorno:giovedi
```

3 Work with files

To use write something on a file you have first to open it. In Python this is done with the global function **open()**(https://docs.python.org/3/library/functions.html#open)

```
In [24]: f = open("test_file.txt", "w")
```

the flag *w* means that the file has been opened in *write* mode. To read a file you have to use *r* flag, ora *r+* to read and write on it.

```
In [25]: a = """Questa stringa deve essere salvata su file
         Questo sarà un file
         formato da più righe.
         """
         f.write(a)

         # You have always to close the file when it is no more necessary
         f.close()
```

A file has always to be closed to flush all the content on the disk. This can be automatically done with the Python construct **with**.

```
In [26]: with open("test_file.txt", "r") as f:
             content = f.read()

         print(content)
```

```
Questa stringa deve essere salvata su file
Questo sarà un file
formato da più righe.
```

The **with** constructs is able to close the file at the end of its use, also in case of exceptions.
A file can be read one character at a time (with method **read()**) or by lines with **readlines()**

```
In [27]: with open("test_file.txt", "r") as f:
             lines = f.readlines()

         print(lines)
```

```
['Questa stringa deve essere salvata su file\n', 'Questo sarà un file\n', 'formato da più righe
```

4 Intermezzo: Random module

```
In [28]: import random
         random.seed(1)

In [29]: random.randrange(1, 100)

Out[29]: 18
```

You have a lot of p.d.f. already included

```
In [30]: random.uniform(1, 100)

Out[30]: 57.351183607399015

In [31]: random.gauss(6, 1)

Out[31]: 6.116452682617494
```

Utilizziamo la list comprehension per creare una lista random

```
In [32]: # Lista di numeri random
         l = [random.randrange(100) for i in range(30)]
         print(l)

[15, 63, 97, 57, 60, 83, 48, 26, 12, 62, 3, 49, 55, 77, 97, 98, 0, 89, 57, 34, 92, 29, 75, 13,
```

4.1 Random and list

You can use the random module to pick a random element from a list or do a random shuffle.

```
In [33]: a = ["ciao", 1, 2.3, "come", 2, 3,  "va?"]
         print(random.choice(a))

ciao


In [34]: random.shuffle(a)
         print(a)

[2, 'ciao', 2.3, 'va?', 1, 3, 'come']
```

5 List high-level manipulation

Let's now talk about a big chapter in Python: list manipulation. Python has a lot of easy-to-use methods to manipulate lists: sort them, apply functions, filter them. They are very computational efficient and concise.

5.1 Sorting

You can sort lists using the **sorted()** global command. Every Python data type has a natural ordering.

```
In [35]: # Lista di numeri random
         l = [random.randrange(100) for i in range(30)]
         print(l)

[67, 28, 97, 56, 63, 70, 29, 44, 29, 86, 28, 97, 58, 37, 2, 53, 71, 82, 12, 23, 80, 92, 37, 15
```

```
In [36]: help(sorted)

Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

```
In [37]: print(l)
         print(sorted(l))
         print(sorted(l, reverse=True))

[67, 28, 97, 56, 63, 70, 29, 44, 29, 86, 28, 97, 58, 37, 2, 53, 71, 82, 12, 23, 80, 92, 37, 15
[2, 12, 15, 23, 28, 28, 29, 29, 37, 37, 42, 44, 53, 54, 56, 58, 63, 64, 67, 70, 71, 80, 82, 86
[97, 97, 95, 92, 92, 91, 86, 82, 80, 71, 70, 67, 64, 63, 58, 56, 54, 53, 44, 42, 37, 37, 29, 2
```

The text is ordered in lexicographic order.

```
In [38]: l = ["c", "a", "ciao", "it", "tools"]
         sorted(l)

Out[38]: ['a', 'c', 'ciao', 'it', 'tools']
```

5.1.1 Ordering criteria

You can always define a personalized way to order a list in Python. This is done using the **key** argument of the **sorted()** function. The key arg must be a function that applied to every element returns the value that the sorted method should use to perform the ordering.

For example: we want to sort a list of tuple on the second element. We should define a function that return the second element of every tuple.

```
In [39]: l = [("Z", 100), ("B", 50), ("B", 20), ("C", 1000)]

In [40]: def take_second_element(item):
             return item[1]
```

Let's pass the function (without parenthesis because we are passing the function object, not calling it!) to the **key** parameter.

```
In [41]: sorted(l, key=take_second_element)

Out[41]: [('B', 20), ('B', 50), ('Z', 100), ('C', 1000)]
```

Since this is a really common operation Python has a built-in method for it! It is called *itemgetter* and it's part of the *operator* built-in module. It doesn exactly what we have done with the tuples: it return the nth element of a list.

```
In [42]: from operator import itemgetter

In [43]: getter = itemgetter(3)
         print(getter([1, 2, 3, 4, 5]))
```

```
4
```

```
In [44]: sorted(l, key=itemgetter(1))
```

```
Out[44]: [('B', 20), ('B', 50), ('Z', 100), ('C', 1000)]
```

5.2 Lambda function

Small anonymous functions can be created with the lambda keyword. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition

Let's use for the first time a lambda function for the **key** argument. Again let's extract the second element of each tuple in the list.

```
In [45]: sorted(l, key=lambda item: item[1])
```

```
Out[45]: [('B', 20), ('B', 50), ('Z', 100), ('C', 1000)]
```

In this way we can create personalized ordering without defining a separate function.

```
In [46]: l = [1, 33, 3, 6, 73]
         sorted(l , key=lambda i: 1/i )
```

```
Out[46]: [73, 33, 6, 3, 1]
```

6 Filtering lists

Elements of lists can be filtered (or selected) simply with the **filter()** global function. The filter function needs a function that when executed on each element of a list decides if the element has to be filtered away or not.

Let's create a random dataset made of a list of tuples containing some data. This is a fairly common structure to quickly save some data in Python. Later on we will use dictionaries.

```
In [47]: from pprint import pprint
```

```
In [48]: elements = ["Co57", "Am241", "Ce137"]
         V = [1000, 500, 100]
         A = [10, 20, 30]
         N = 20
         data = []
         for i in range(N):
             data.append(
                 ( i, random.choice(elements), random.choice(V),
                  random.choice(A), random.uniform(1, 100) )
             )
```

```
In [49]: data[:2]

Out[49]: [(0, 'Ce137', 100, 10, 31.033482582358843),
          (1, 'Ce137', 500, 30, 39.94200868391412)]
```

The first argument of the **filter()** method is the filtering function, the second is an iterable. The function must return True or False when executed on each element: here a **lambda** function can be really useful for quick criteria.

```
In [50]: def filter_function(elem):
             return elem[1] == "Co57"

In [51]: result = filter(filter_function, data)

In [52]: for r in result:
             print(r)

(2, 'Co57', 500, 10, 74.62933487402663)
(5, 'Co57', 500, 30, 51.3342223367482)
(6, 'Co57', 100, 20, 37.681743997928926)
(10, 'Co57', 1000, 30, 92.08855705833948)
(14, 'Co57', 500, 30, 51.73749351579584)
(15, 'Co57', 100, 30, 21.34304051237536)
(16, 'Co57', 500, 20, 57.42993405376164)
(17, 'Co57', 100, 20, 49.00758611054568)


In [53]: result = filter(lambda el: el[1]=="Co57" , data )

In [54]: for elem in result:
             print(elem)

(2, 'Co57', 500, 10, 74.62933487402663)
(5, 'Co57', 500, 30, 51.3342223367482)
(6, 'Co57', 100, 20, 37.681743997928926)
(10, 'Co57', 1000, 30, 92.08855705833948)
(14, 'Co57', 500, 30, 51.73749351579584)
(15, 'Co57', 100, 30, 21.34304051237536)
(16, 'Co57', 500, 20, 57.42993405376164)
(17, 'Co57', 100, 20, 49.00758611054568)
```

The filter function doesn't return directly a list, but an object called **generator** that can be iterated to read the results. The reason for this is computation efficiencies when different operation are made one after another as we will see later...

A full list of results can be extracted using the **list()** construct to build the list.

```
In [55]: list(filter(lambda exp: exp[1]=="Am241" , data ))

Out[55]: [(3, 'Am241', 500, 30, 18.12773275632604),
          (18, 'Am241', 500, 20, 1.1568902455970673)]
```

We can make more complex filters.

```
In [56]: for l in filter(lambda exp: exp[1]=="Co57" and exp[3]> 10, data ):
             print(l)

(5, 'Co57', 500, 30, 51.3342223367482)
(6, 'Co57', 100, 20, 37.681743997928926)
(10, 'Co57', 1000, 30, 92.08855705833948)
(14, 'Co57', 500, 30, 51.73749351579584)
(15, 'Co57', 100, 30, 21.34304051237536)
(16, 'Co57', 500, 20, 57.42993405376164)
(17, 'Co57', 100, 20, 49.00758611054568)
```

When the filter becomes more complex it's better to use a separate function instead of lambdas that can contain only one function.

```
In [57]: def choice(item):
             if (item[2]+ item[3] > 9):
                 if (item[3]< 24):
                     return True
             return False

In [58]: for l in filter(choice, data ):
             print(l)

(0, 'Ce137', 100, 10, 31.033482582358843)
(2, 'Co57', 500, 10, 74.62933487402663)
(6, 'Co57', 100, 20, 37.681743997928926)
(7, 'Ce137', 1000, 20, 5.3052417452962155)
(9, 'Ce137', 1000, 10, 50.72161728491483)
(11, 'Ce137', 1000, 20, 51.863394655576066)
(12, 'Ce137', 500, 20, 91.06574657305082)
(16, 'Co57', 500, 20, 57.42993405376164)
(17, 'Co57', 100, 20, 49.00758611054568)
(18, 'Am241', 500, 20, 1.1568902455970673)
```

As you can notice the **generator** returned by the filter function can be iterated as a normal list.

7 Complete example: Lab dataset

It is better to use dictionaries instead of list of tuples to assign a name to *columns*. In the next lesson you will see the Python library **Pandas** that can be use to build and work on complex dataframes. Let's use the *list comprehension* to build again our dataset.

```
In [80]: data =  [{
             "ID"  : i,                                          # measurement ID
             "elem": random.choice(["Co57", "Am", "Na22"]),     # element under measurement
```

```
            "HV": random.gauss(20, 10),                    # voltage
            "GN": random.uniform(1, 59),                   # electronic gain
            "res": random.uniform(1, 100)                  # resolution
            }
             for i in range(10000) ]

        # Add energies
        energies = {
            "Co57":  (122, 10),
            "Am":    (100, 20),
            "Na22":  (1200, 50)
        }
        for el in data:
            el["En"] = random.gauss(*energies[el["elem"]])
```

```
In [81]: data[0:2]
```

```
Out[81]: [{'ID': 0,
           'elem': 'Am',
           'HV': 21.718803194490697,
           'GN': 48.40197731328848,
           'res': 79.40468632615196,
           'En': 92.81821176102201},
          {'ID': 1,
           'elem': 'Am',
           'HV': 28.447423462197843,
           'GN': 30.87754497675246,
           'res': 65.38703670966453,
           'En': 94.19940374711072}]
```

Let's get only measurements with "Na22" and get the resultion from them. We can do all these operations with a single line

```
In [82]: pprint(data[2])
         data[2]["HV"]
```

```
{'En': 109.58892364067567,
 'GN': 55.28594577390318,
 'HV': 25.685590439146896,
 'ID': 2,
 'elem': 'Am',
 'res': 52.239878503803666}


Out[82]: 25.685590439146896
```

```
In [83]: #scriviamo un filtro che selezioni solamente i dati con HV > 15
         results = filter(lambda elem: elem["HV"]> 50, data)
         pprint(list(results)[:3])
```

```
[{'En': 1171.918375323987,
  'GN': 2.894651139221491,
  'HV': 51.23244093573522,
  'ID': 6,
  'elem': 'Na22',
  'res': 40.971102895599216},
 {'En': 90.41403041115365,
  'GN': 40.098978006286536,
  'HV': 52.15681047168499,
  'ID': 207,
  'elem': 'Am',
  'res': 36.82980022201666},
 {'En': 1146.560390702732,
  'GN': 20.265932918988568,
  'HV': 51.384645017914444,
  'ID': 216,
  'elem': 'Na22',
  'res': 25.385736245339505}]
```

```
In [84]: res_na = [  line["res"]  for line in filter(lambda item: item["elem"] =="Na22", data)
```

```
In [85]: print(len(res_na))
         print(res_na[:6])
```

```
3330
[90.60017166491117, 40.971102895599216, 65.51925005582456, 70.71716418747096, 78.39537807698785
```

7.1 Extract sub-dictionaries

We can combine filtering and **dictionary comprehension** to extract a new dataset keeping only some keys.

```
In [86]: keep_columns = ["elem", "HV", "res"]
```

```
In [87]: res_na = [  {k: line[k] for k in keep_columns}
                   for line in filter(lambda item: item["elem"] =="Na22", data) ]
```

```
In [88]: res_na[:3]
```

```
Out[88]: [{'elem': 'Na22', 'HV': 31.72481659024525, 'res': 90.60017166491117},
          {'elem': 'Na22', 'HV': 51.23244093573522, 'res': 40.971102895599216},
          {'elem': 'Na22', 'HV': 10.24971014014126, 'res': 65.51925005582456}]
```

The dictionary comprehension is equivalent to this snippet of code

```
In [89]: res_na = []
         filter_results = filter(lambda item: item["elem"] =="Na22", data)
```

```
        for line in filter_results:
            obj = {}
            for k in keep_columns:
                obj[k] = line[k]
            res_na.append(obj)
```

```
In [90]: res_na[:3]
```

```
Out[90]: [{'elem': 'Na22', 'HV': 31.72481659024525, 'res': 90.60017166491117},
          {'elem': 'Na22', 'HV': 51.23244093573522, 'res': 40.971102895599216},
          {'elem': 'Na22', 'HV': 10.24971014014126, 'res': 65.51925005582456}]
```

We can now sort the results in order of resolution.

```
In [91]: sorted(res_na, key=lambda item: item["res"])[0:3]
```

```
Out[91]: [{'elem': 'Na22', 'HV': 16.26077580384494, 'res': 1.0240228112946566},
          {'elem': 'Na22', 'HV': 15.825347524133164, 'res': 1.0255971138887627},
          {'elem': 'Na22', 'HV': 23.826721968928144, 'res': 1.0343350978059194}]
```

8 Map operations on list

Applying a function on every element of a list is a very common operation. The **map()** function is made for that.

For example if we want to get a percentage resolution given the energy that you have measured within our dataset:

```
In [92]: result = map(lambda el: el["res"]/el["En"], data)
```

We can look at the result

```
In [93]: list(result)[:5]
```

```
Out[93]: [0.8554860605437465,
          0.6941342950025846,
          0.4766894022528204,
          0.5989189035565113,
          0.07350506616742666]
```

But it is better to save the result creating a new dataset and inserting the new values for example

```
In [94]: def add_energy_resolution(el):
             el["res%"] = el["res"] / el["En"]
             return el

         new_data = list(map(add_energy_resolution, data))
```

Since we cannot use a lambda to modify the element of the list on the go we can use a separate function that calculates the resolution and return a new modified element.

```
In [95]: new_data[:3]
```

```
Out[95]: [{'ID': 0,
           'elem': 'Am',
           'HV': 21.718803194490697,
           'GN': 48.40197731328848,
           'res': 79.40468632615196,
           'En': 92.81821176102201,
           'res%': 0.8554860605437465},
          {'ID': 1,
           'elem': 'Am',
           'HV': 28.447423462197843,
           'GN': 30.87754497675246,
           'res': 65.38703670966453,
           'En': 94.19940374711072,
           'res%': 0.6941342950025846},
          {'ID': 2,
           'elem': 'Am',
           'HV': 25.685590439146896,
           'GN': 55.28594577390318,
           'res': 52.239878503803666,
           'En': 109.58892364067567,
           'res%': 0.4766894022528204}]
```

8.1 Chain operations

Map, filter, and sort operations can be chained in a really efficient chain of operation. That's the reason behind the use of generators instead of plain lists. The generators return a single element that is passed through all the steps of the computation chain so that you can work with lists of millions of elements without running out of memory.

8.1.1 Example:

If we want to compute something only on Co57 measurements, using some constraints on Voltages and then sort the result.

```
In [96]: a = sorted(filter(lambda el: el["res%"] < 0.1,                          #>>>>> Fina
                           map(  add_energy_resolution,                        #>>>> Mapping
                               filter (lambda el: el["HV"] > 10,               #>> Second filter
                                   filter(lambda el: el["elem"]== "Co57", data)    # First filter
                                  )
                              )
                       )
                  , key=lambda el: el["HV"], reverse=True)
```

Now we have calculated the energy resolution only on part of the dataset and we have a generator **a**. We can now work on this generator getting for example the measurement with minimum value of the resolution%.

```
In [97]: min(a, key=lambda el: el["res%"])
```

```
Out[97]: {'ID': 3058,
          'elem': 'Co57',
          'HV': 25.95602600829662,
          'GN': 34.32779686289394,
          'res': 1.095129276601117,
          'En': 136.80952585277987,
          'res%': 0.008004773569492382}
```

N.B.: If we run again the instruction above we get an error. That is because the generator *a* has been run once and now it's empty. The elements in the generator are only loaded once when they are requested.

This is a memory saving procedure to work with big dataset. If you want to save intermediate result you have to get explicitly a list out of the generator as we have done before:

```
In [98]: results = list(a)
         print("N results:", len(results))
         pprint(results[0:3])
```

```
N results: 313
[{'En': 133.3966638920495,
  'GN': 49.483544198825115,
  'HV': 47.67683433640678,
  'ID': 4169,
  'elem': 'Co57',
  'res': 1.1313121226386298,
  'res%': 0.008480812710234925},
 {'En': 119.7872096781031,
  'GN': 24.78055464747813,
  'HV': 47.56472574225498,
  'ID': 7013,
  'elem': 'Co57',
  'res': 1.4838323428952789,
  'res%': 0.012387235222213553},
 {'En': 112.19887377358941,
  'GN': 41.009835969912366,
  'HV': 47.482396749653155,
  'ID': 1606,
  'elem': 'Co57',
  'res': 4.230218180681377,
  'res%': 0.03770285777750054}]
```

8.1.2 Exercise

```
In [102]: '''
          - Given measurements with GN > 40,
          - Calculate GN/HV and save the value
          - select measurements with GN/HV > 3
          - return the measurement of Co57 with minimum res%
          '''
          def gnratio(el):
              el["gn/hv"] = el["GN"] / el["HV"]
              return el

          min(filter(lambda k: k["elem"]=="Co57",
                 filter(lambda k: k["gn/hv"] > 3,
                        map(gnratio,
                            filter(lambda el: el["GN"]>=40, data)
                           )
                       )
                ), key=lambda el:el["res%"])
```

```
Out[102]: {'ID': 7765,
           'elem': 'Co57',
           'HV': 4.592995206780179,
           'GN': 50.198283188121515,
           'res': 1.0516476829144863,
           'En': 135.9284960147753,
           'res%': 0.007736771271273193,
           'gn/hv': 10.929313210259574}
```

9 Save data on file

Dataset structured as dictionaries and list can be saved in text file in **json** format. In Python there is a module to work with this format.on**.

```
In [99]: import json
```

We can now read back the data from json file

```
In [100]: data = json.load(open("data.json", "r"))
```

To save on disk:

```
In [101]: json.dump(data, open("data.json", "w"))
```