Serialization

Suppose we have the following "complex" structure:

In [2]:
```
a={'a': [1,2,3,], 'ffrrf': {'b': [4,4,5,6]}}
```

How can we save `a` in a file? We can convert it to a string in json format:

In [3]:
```
import json
a_json = json.dumps(a)
print (a_json)
```

```
{"a": [1, 2, 3], "ffrrf": {"b": [4, 4, 5, 6]}}
```

In [3]:
```
type(a_json)
```

Out[3]:
```
str
```

Then we can save `a_json` to a file:

In [4]:
```
with open('results.txt', 'w') as f:
    f.write(a_json + '\n')
```

Or else:

In [5]:
```
with open('results.txt', 'w') as f:
    json.dump(a, f)
```

In [6]:
```
!cat results.txt
```

```
{"a": [1, 2, 3], "ffrrf": {"b": [4, 4, 5, 6]}}
```

If you work in Microsoft Windows, you can type:

In [1]:
```
!type results.txt
```

```
/bin/sh: line 0: type: results.txt: not found
```

After we send the file, the recipient can open it, as follows:

In [8]:
```
with open('results.txt') as f:
    a = json.load(f)
print (a)
print (type(a))
```

```
{'a': [1, 2, 3], 'ffrrf': {'b': [4, 4, 5, 6]}}
<class 'dict'>
```

This process is called serialization and allows us to share the data along with its structure.

Not everything can be converted to json:

In [4]:
```
try:
    json.dumps({1,2,3,4}) # Sets cannot be serialized in json
except Exception as e:
    print (e)
```

```
Object of type set is not JSON serializable
```

In [5]:
```python
def f(x):
    return x+1

try:
    json.dumps(f) # Functions cannot be serialized
except Exception as e:
    print (e)
```

```
Object of type function is not JSON serializable
```

The json format is very popular in "structured data" sharing. This includes the data consisting of lists and dictionaries and/or a combination of them. It is also very common for an online database to share its data in this format. For example in this link:

http://mygene.info/v3/query?q=tumor&fields=symbol&size=1000&species=human '

You can query a database to get a list of 1000 genes that have been linked to cancer. This query will return the results in json format.

Apart from json there are other formats used for exchange of structured data. Some examples are XML and YAML .

However, we noticed that json can only accept lists and dictionaries. Another option is the pickle bookcase which can serialize a much larger number of structures. The drawbacks are:

1. It is only purposed for python (you may search for available libraries in other languages)
2. It is not human-readable (unlike json).

   Let's take a look at an example:

In [19]:
```python
import pickle


def a_function(x):
    return x+1

a_set = {1,2,3,4}

a_list = [1,'mitsos', {1:True}, a_function, a_set]

with open('my_data.pickle', 'wb') as f:
    pickle.dump(a_list, f)
```

Notice that `'wb'` means recording in binary format. Unlike plain `'w'` or `'wt'` which marks text format.

Let's un-pickle it now!

In [20]:
```python
with open('my_data.pickle', 'rb') as f:
    data = pickle.load(f)
```

In [21]:
```python
data[3](10) # We can call function that has been un-pickled!
```

Out[21]: 11

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Itertools library

Itertools contains functions that help you do iterations and .. loops! It is one of the most useful libraries, mainly because it helps you to simplify your code. Before attempting to do any complex iterations (for inside for inside for ...), check if any of the itertools functions can help you.

Problem 1 (Cartesian product)

You enter a clothing store. The store has 10 different pairs of shoes in your size and their prices are:

```
shoes = [22, 30, 83, 28, 72, 51, 61, 83, 25]
```

The store has 3 different jeans in your size and their prices are:

```
jeans = [30, 79, 34]
```

The store has 8 different t-shirts in your size. The prices are:

```
shirts = [24, 25, 40, 40, 26, 28, 19]
```

You have 100 euros and you have to buy one of each kind. How many clothing combinations (shoes, jeans and t-shirt) can you buy?

In [22]:
```
shoes = [22, 30, 83, 28, 72, 51, 61, 83, 25]
jeans = [30, 79, 34]
shirts = [24, 25, 40, 40, 26, 28, 19]
```

In [25]:
```
# Classic solution:
c = 0
for x in shoes:
    for y in jeans:
        for z in shirts:
            if x+y+z<=100:
                c += 1
print (c)
```

```
53
```

In [28]:
```
# With itertools:
from itertools import product

c = 0
for x,y,z in product(shoes, jeans, shirts):
    if x+y+z<=100:
        c += 1
print (c)
```

```
53
```

Problem 2 (combinations)

You enter a store that only sells t-shirts. The available types and prices for each t-shirt are:

```
shirts = [
    ('a', 22),
```

```
    ('c', 83),
    ('d', 28),
    ('e', 72),
    ('f', 51),
    ('g', 61),
    ('h', 83),
    ('i', 25),
]
```

You can spend a maximum of 100e and you should get exactly two. Which pairs can you choose?

In [34]:
```
shirts = [
    ('a', 22),
    ('b', 30),
    ('c', 83),
    ('d', 28),
    ('e', 72),
    ('f', 51),
    ('g', 61),
    ('h', 83),
    ('i', 25),
]

#The classic solution:
c = 0
for i_1, (kind_1, price_1) in enumerate(shirts):
    for kind_2, price_2 in shirts[i_1+1:]:
        if price_1 + price_2 <= 100:
            c += 1
print (c)
```

17

In [35]:
```
# With itertools
from itertools import combinations

c = 0
for (kind_1, price_1), (kind_2, price_2) in combinations(shirts, 2):
    if price_1 + price_2 <= 100:
        c += 1
print (c)
```

17

Problem 3 (instead of while)

What is the sum of all prime numbers which are less than 1000?

In [60]:
```python
from itertools import takewhile

# Initially we build a enerator with prime numbers:
def gen_primes():
    yield 1
    n = 2
    while True:
        for i in range(2, n):
            if n%i==0:
                break
        else:
            yield n
        n += 1

# We build a fucntion that checks when we stop:
def f(x):
    return x<1000

# Calculate the sum of the prime numbers, until you find a prime number th
sum(takewhile(f, gen_primes()))
```

Out[60]:
```
76128
```

Regular Expressions

Regular Expressions (or regexp for short) are a basic idea in computer science (existing since 1956 ..). It is essentially a new language with which you can declare some patterns in a string. Special algorithms undertake to detect these patterns at a very high speed. Regexp is implemented in python by the `re` library:

In [61]:
```python
import re # Regular Expression
```

Regular expressions allow you to do complex operations on strings very fast. These functions are:

- Check if a string follows a specific format / pattern (e.g. consists of 4 numbers and 2 letters)
- Get a sub-string. For example, export the year from a date of birth
- Get all the sub-strings that follow a pattern. For example extract all the dates contained in a large text file.
- Replace a pattern contained in a string with another. For convert all dates from the Month/Day/Year format (US system) to Day/Month/Year format (European system).

Regular expressions (regex) are primarily strings. Each regex also indicates a pattern. For example: `\d` denotes "a character that is a number". Let's see it in practice:

In [65]:
```python
re.search(r'\d', '5')
```

Out[65]:
```
<re.Match object; span=(0, 1), match='5'>
```

This command basically says: "Find if there is at least one number in the string". Notice that "something" was returned (we will see later what the returned value is). At this point we can also check what is returned in case it does NOT find the pattern:

In [66]:
```
a = re.search(r'\d', 'a')
print (a)
```

None

If the pattern does not exists, it returns: `None` . We can extend the pattern by asking for a number to be followed by the character "a":

In [67]:
```
a = re.search(r'\da', 'hello5ahello')
print (a)
```

```
<re.Match object; span=(5, 7), match='5a'>
```

In [68]:
```
a = re.search(r'\da', 'hello5hello')
print (a)
```

None

We notice that:

- In the 1st case the pattern was found. By using the search command, we ask to find it **anywhere** in the string.
- In the 2nd case the pattern was not found. There is no number followed by the letter "a".

We continue by asking for a number that is followed by either the letter "a" or the letter "b":

In [69]:
```
a = re.search(r'\d[ab]', 'hello5ahello')
print (a)
```

```
<re.Match object; span=(5, 7), match='5a'>
```

In [70]:
```
a = re.search(r'\d[ab]', 'hello5bhello')
print (a)
```

```
<re.Match object; span=(5, 7), match='5b'>
```

In [82]:
```
a = re.search(r'\d[ab]', 'hello5chello')
print (a)
```

None

By using the brackets we declare a set of characters. In other words, we ask to find the one and only character that belongs to this set.

The next step is to ask for any number followed by any character from `a` up to `k` : In brackets we can declare 1 or more character ranges:

In [83]:
```
a = re.search(r'\d[a-k]', 'hello5dhello')
print (a)
```

```
<re.Match object; span=(5, 7), match='5d'>
```

In [85]:
```
a = re.search(r'\d[a-k]', 'hello5lhello')
print (a)
```

None

Next, we request a number followed by any character other than those belonging to the range a-k. To do this we add the carret (`^`) as the first character in the brackets:

In [86]:
```
a = re.search(r'\d[^a-k]', 'hello5dhello')
print (a)
```

```
None
```

In [87]:
```
a = re.search(r'\d[^a-k]', 'hello5lhello')
print (a)
```

```
<re.Match object; span=(5, 7), match='5l'>
```

We continue by asking for a number that consists of any character! The dot (`.`) means "any character":

In [72]:
```
a = re.search(r'\d.', 'hello5bhello')
print (a)
```

```
<re.Match object; span=(5, 7), match='5b'>
```

In [73]:
```
a = re.search(r'\d.', 'hellohello5')
print (a)
```

```
None
```

We continue by declaring a number and a blank, a tab or a new line. The special character `\s` indicates "white space":

In [77]:
```
a = re.search(r'\d\s', 'hello5 hello')
print (a)
```

```
<re.Match object; span=(5, 7), match='5 '>
```

In [78]:
```
a = re.search(r'\d\s', 'hello5hello')
print (a)
```

```
None
```

We continue by declaring a number followed by any letter which is NOT a special character. Pattern `\w` indicates any character belonging to: `a-z` `A-Z` `0-9` and `_` :

In [79]:
```
a = re.search(r'\d\w', 'hello5hello')
print (a)
```

```
<re.Match object; span=(5, 7), match='5h'>
```

In [81]:
```
a = re.search(r'\d\w', 'hello5$hello')
print (a)
```

```
None
```

More specifically, instead of writing `[0-9]` to denote all numbers and `[a-zA-Z]` to denote all letters we use the following:

`\d` is the same as: `[0-9]`

`\w` is the same as: `[a-zA-Z0-9_]`

`\s` is the same as: `[ \t\n\r\f\v]`

Repeating patterns

We can create a pattern to find multiple repetitions of a character set. For example we can ask: to have 1 or more numbers followed by the letter "a". We do this with the special

In [88]:

```
a = re.search(r'\d+a', 'hello123431ahello') # Many digits after "a"
print (a)
```

```
<re.Match object; span=(5, 12), match='123431a'>
```

In [90]:

```
a = re.search(r'\d+a', 'hello1ahello') # Ony one digit after "a"
print (a)
```

```
<re.Match object; span=(5, 7), match='1a'>
```

In [91]:

```
a = re.search(r'\d+a', 'helloahello') # Zero digits after "a" (does not ma
print (a)
```

```
None
```

If instead of $+$ we use $*$, then we declare: "none or many". Or else, while with $+$ there must be at least 1, with $*$ there may not be any:

In [7]:

```
a = re.search(r'\d*a', 'hello444a') # many numbers after a ! OK!
print (a)
```

```
<re.Match object; span=(5, 9), match='444a'>
```

In [93]:

```
a = re.search(r'\d*a', 'hello4a') # one number after a ! OK!
print (a)
```

```
<re.Match object; span=(5, 7), match='4a'>
```

In [94]:

```
a = re.search(r'\d*a', 'helloa') # no mumber after a ! OK!
print (a)
```

```
<re.Match object; span=(5, 6), match='a'>
```

In [95]:

```
a = re.search(r'\d*a', 'hel555lo') # There is a number but no a. NOT OK!
print (a)
```

```
None
```

We can also declare "one or none". For example we want either no number after "a", or a number after "a". We can do this with the character $?$:

In [103…

```
a = re.search(r'b\d?a', 'b5a') # "b" then one digit then "a" . OK
print (a)
```

```
<re.Match object; span=(0, 3), match='b5a'>
```

In [104…

```
a = re.search(r'b\d?a', 'ba') # "b" then zero digit then "a". OK
print (a)
```

```
<re.Match object; span=(0, 2), match='ba'>
```

In [106…

```
a = re.search(r'b\d?a', 'b65a') # "b" then manydigits then "a". NOT OK
print (a)
```

```
None
```

Finally, we can ask for a given set of characters to have a certain number of repetitions:

In [107…

```
a = re.search(r'ba{3}b', 'baaab') # b, three times a and then b
print (a)
```

```
<re.Match object; span=(0, 5), match='baaab'>
```

In [108…

```
a = re.search(r'ba{3}b', 'baab') # b three times a and and then b
print (a)
```

None

In [109…
```
a = re.search(r'ba{3}b', 'baaaab') # b three times a and then b
print (a)
```
None

or declare a range of repetitions:

In [112…
```
a = re.search(r'ba{2,4}b', 'bab') # from 2 to 4 "a"
print (a)
```
None

In [113…
```
a = re.search(r'ba{2,4}b', 'baab') # from 2 to 4 "a"
print (a)
```
```
<re.Match object; span=(0, 4), match='baab'>
```

In [114…
```
a = re.search(r'ba{2,4}b', 'baaab') # from 2 to 4  "a"
print (a)
```
```
<re.Match object; span=(0, 5), match='baaab'>
```

In [115…
```
a = re.search(r'ba{2,4}b', 'baaaab') # from 2 to 4 "a"
print (a)
```
```
<re.Match object; span=(0, 6), match='baaaab'>
```

In [116…
```
a = re.search(r'ba{2,4}b', 'baaaaab') # from 2 to 4 "a"
print (a)
```
None

In [118…
```
a = re.search(r'ba{2,}b', 'baaaaab') # 2 or more
print (a)
```
```
<re.Match object; span=(0, 7), match='baaaaab'>
```

In [119…
```
a = re.search(r'ba{2,}b', 'bab') # 2 or more
print (a)
```
None

In [120…
```
a = re.search(r'ba{2,}b', 'baaaaaaab') # 2 or more
print (a)
```
```
<re.Match object; span=(0, 9), match='baaaaaaab'>
```

In [122…
```
a = re.search(r'ba{,2}b', 'baaab') # 2 or less
print (a)
```
None

In [123…
```
a = re.search(r'ba{,2}b', 'baab') # 2 or less
print (a)
```
```
<re.Match object; span=(0, 4), match='baab'>
```

In [124…
```
a = re.search(r'ba{,2}b', 'bab') # 2 or less
print (a)
```
```
<re.Match object; span=(0, 3), match='bab'>
```

In [126…

```
a = re.search(r'ba{,2}b', 'bb') # 2 or less
print (a)
```

```
<re.Match object; span=(0, 2), match='bb'>
```

Let's take a look at the following:

In [129…

```
a = re.search(r'ba+b', 'hellobaaaaabhello')
print (a)
```

```
<re.Match object; span=(5, 12), match='baaaaab'>
```

With `.group` function we can find WHAT the match was. Setting the value 0 as a parameter returns all the strings that matched the pattern:

In [133…

```
a.group(0)
```

Out[133…

```
'baaaaab'
```

Here we notice the following: `a+` "matched" all the "a"s, this is called greedy search. Python will generally try to match as many characters as possible. This can cause us problems! For example. Let the string:

In [135…

```
s = 'gene:G1 function: F1, gene:G2 function:F2, gene:G3 function:F3'
```

Let's assume we need the name of the first gene. So the patterns matches anything that starts with `gene:` followed by an indefinite number of characters and ends with the string `function` :

In [137…

```
a = re.search(r'gene:.+function', s)
print (a.group(0))
```

```
gene:G1 function: F1, gene:G2 function:F2, gene:G3 function
```

What happened here? we notice that what is returned actually follows the pattern, since it starts with a `gene` and ends in a `function` . This happened because python tried to return the largest possible match. The string `gene:G1 function` follows the pattern, but it is not the largest possible. We can prevent this behavior by putting one `?` after `+` :

In [138…

```
a = re.search(r'gene:.+?function', s)
print (a.group(0))
```

```
gene:G1 function
```

The character `?` after `+,*,?,{}` instructs python to "match the smallest string possible". See these examples:

In [143…

```
a = re.search(r'b\d+\d', 'b12345')
print (a.group(0))
```

```
b12345
```

In [144…

```
a = re.search(r'b\d+?\d', 'b12345')
print (a.group(0))
```

```
b12
```

In [145…

```
a = re.search(r'b\d*\d', 'b12345')
print (a.group(0))
```

b12345

In [146...
```
a = re.search(r'b\d*?\d', 'b12345')
print (a.group(0))
```
b1

In [147...
```
a = re.search(r'b\d?\d', 'b12345')
print (a.group(0))
```
b12

In [148...
```
a = re.search(r'b\d??\d', 'b12345')
print (a.group(0))
```
b1

In [151...
```
a = re.search(r'b\d{2,4}', 'b12345') # chooses the larger --> 4
print (a.group(0))
```
b1234

In [152...
```
a = re.search(r'b\d{2,4}?', 'b12345') #  chooses the smaller --> 2
print (a.group(0))
```
b12

Start and end

We can state that a pattern must exist in the beginning of the string, if we set in the beginning of the pattern the character `^` :

In [154...
```
a = re.search('^\d', '4hello') # The number must be in the beginning! OK!
print (a)
```
```
<re.Match object; span=(0, 1), match='4'>
```

In [155...
```
a = re.search('^\d', 'h4ello') # The number must be in the beginning! NOT (
print (a)
```
None

Similarly, we can declare that the pattern will be at the end with the character `$` :

In [156...
```
a = re.search('\d$', 'hello4') # The number must be in the end! OK!
print (a)
```
```
<re.Match object; span=(5, 6), match='4'>
```

In [157...
```
a = re.search('\d$', 'hell4o') # The number must be in the end! NOT OK!
print (a)
```
None

The operator or -> `|`

Sometimes we need a pattern to match SOMETHING or SOMETHING ELSE. This is done by putting the two patterns in parentheses and using the `|` operator in between:

In [158...
```
a = re.search(r'(ab)|(kl)', 'ab') # ab or kl
print (a)
```
```
<re.Match object; span=(0, 2), match='ab'>
```

In [159…
```
a = re.search(r'(ab)|(kl)', 'kl') # ab or kl
print (a)
```

```
<re.Match object; span=(0, 2), match='kl'>
```

In [160…
```
a = re.search(r'(ab)|(kl)', 'al') # ab or kl
print (a)
```

None

We can create nested `|` :

In [165…
```
a = re.search(r'(a(12)|(34)b)|(1(ab)|(kl)2)', 'a12')
print (a)
a = re.search(r'(a(12)|(34)b)|(1(ab)|(kl)2)', '34b')
print (a)
a = re.search(r'(a(12)|(34)b)|(1(ab)|(kl)2)', '1ab')
print (a)
a = re.search(r'(a(12)|(34)b)|(1(ab)|(kl)2)', 'kl2')
print (a)
```

```
<re.Match object; span=(0, 3), match='a12'>
<re.Match object; span=(0, 3), match='34b'>
<re.Match object; span=(0, 3), match='1ab'>
<re.Match object; span=(0, 3), match='kl2'>
```

Extracting fields from patterns

Some times we want to extract subfields from a string. We do this by inserting parentheses in the pieces of the pattern we want to export:

In [8]:
```
plate_number = ' This is my plate number: ABE 1234 hello'

a = re.search(r'(\w+) (\d+)', plate_number)
# notice the difference with this:
# a = re.search(r'\w+ \d+', plate_number)
```

Next, we use the group to get these fields:

The whole string that matched:

In [168…
```
a.group(0)
```

Out[168… 'ABE 1234'

The string that matched the 1st parenthesis:

In [169…
```
a.group(1)
```

Out[169… 'ABE'

the string that matched the 2nd parenthesis:

In [170…
```
a.group(2)
```

Out[170… '1234'

Functions `match`, `exactmatch`, `findall` and `sub`:

The `search` function we have seen so far is used to find a pattern within a string.

 which matches **only if the** whole string matches the pattern. Notice the difference:

In [171…

```
a = re.search('rs\d+', 'This is a mutation: rs123456')
print (a)
```

```
<re.Match object; span=(20, 28), match='rs123456'>
```

In [175…

```
a = re.fullmatch('rs\d+', 'This is a mutation: rs123456')
print (a)
```

```
None
```

In [177…

```
a = re.fullmatch('rs\d+', 'rs123456')
print (a)
```

```
<re.Match object; span=(0, 8), match='rs123456'>
```

`fullmatch` does the same with the `search` function, if we enclose the pattern in `^$`
:

In [179…

```
a = re.search('rs\d+', 'This is a mutation: rs123456')
print (a)
```

```
<re.Match object; span=(20, 28), match='rs123456'>
```

In [181…

```
a = re.search('^rs\d+$', 'This is a mutation: rs123456')
print (a)
```

```
None
```

In [183…

```
a = re.search('^rs\d+$', 'rs123456')
print (a)
```

```
<re.Match object; span=(0, 8), match='rs123456'>
```

In [185…

```
a = re.fullmatch('rs\d+', 'rs123456')
print (a)
```

```
<re.Match object; span=(0, 8), match='rs123456'>
```

The `match` function checks whether the pattern is in the beginning of the string. This is equivalent to using the `search` function and adding a `^` in front of the pattern:

In [187…

```
a = re.match(r'\d+', '123hello')
print (a)
```

```
<re.Match object; span=(0, 3), match='123'>
```

In [188…

```
a = re.match(r'\d+', 'hello123')
print (a)
```

```
None
```

In [189…

```
a = re.search(r'^\d+', '123hello')
print (a)
```

```
<re.Match object; span=(0, 3), match='123'>
```

In [190…

```
a = re.search(r'^\d+', 'hello123')
print (a)
```

```
None
```

Finally, the `sub` function changes the part that has been matched to one string with another string. In this way, we can "capture" the groups with parentheses in the pattern and then refer to it with the character `\` followed by the number of the capturing

In [198…
```
s = 'Name: James Bond'
re.sub(r'Name: (\w+) (\w+)', r'My name is \2, \1 \2', s)
```

Out[198…
```
'My name is Bond, James Bond'
```

Finally, with the `findall` function we get all the matches of a pattern in a string:

In [199…
```
s = 'dg +5aaghqq4  ajdfhal+48f4++85tyru+4867dhgjghi4yifhl4i8+hdji74rl48ru'
re.findall(r'\+\d+', s) #  All numbers preceeded with the character "+"
```

Out[199…
```
['+5', '+48', '+85', '+4867']
```

What is this `r` that you put in front of each pattern?

As we have seen we can put "special" characters inside a string. For example we can put "Enter" (or otherwise new line):

In [200…
```
s = 'a\nb'
print (s)
```

```
a
b
```

Similarly, if we want a string to have the character `\` , we have to put it twice:

In [204…
```
s = 'a\\b'
print (s)
```

```
a\b
```

But what if we want to see if a string contains the `\` character? This character is a special character for both python but also for regular expressions (Note the `\+` that we put above in `findall` to match the character `+`). So to declare the character `\` in a pattern we have to escape it and make the pattern: `\\` . That is, just as we matched the `+` with the pattern `\+` , so we match the `\` with the `\\` . So we have to "make" a string which when we print it should print: `\\` . This string is:

In [211…
```
s = '\\\\'
print (s)
```

```
\\
```

So, in order to to match the character `\` we need to write the following:

In [213…
```
s = 'a\\b'
a = re.search('a\\\\b', s)
print (a)
```

```
<re.Match object; span=(0, 3), match='a\\b'>
```

This can be quite confusing and be a source of error. Unfortunately, this problem is common to all programming languages for many years. Collectively, this problem is referred to as leaning toothpick syndrome (!!). One solution that python has is to be able to declare a string as raw. A raw string is declared by putting r in front and means that it does not contain any other character than the ones inside the string (no special character)!

In [9]:
```
s = r'a\nb'
print (s)
```

```
a\nb
```

In [10]:
```
# Notice the difference:
s = 'a\nb'
print(s)
```

```
a
b
```

This way we can declare regular expressions without worrying about the special characters of python being confused with the special characters of regular expressions:

In [214…
```
s = 'a\\b'
a = re.search(r'a\\b', s)
print (a)
```

```
<re.Match object; span=(0, 3), match='a\\b'>
```

In case all this sounds difficult and confusing (this is ok..) you can remember the following: **When we use regular expressions, we always declare the patterns as raw strings by putting an r in front.**

Also the official python documentation explains the raw strings very nicely.

More on regular expressions

This is less than half of the theory regarding regular expressions and how python supports them. You can read more about:

- Look ahead and look behind regular expressions
- Named groups
- No capturing parenthesis
- Compilation flags . For example, should the dot "catch" enter? Or how can I match without ignoring the difference between uppercase and lowercase letters?
- Greedy Vs. non-greedy (we explained it a bit here)
- compile . You can compile a complex regexp to make it match strings much faster
- debug . There are also special sites that help you write and correct your regexp: debuggex , pythex
- comments . You can put comments **into** a regexp

But well how complicated can a regexp become? Answer: Quite complicated , but this should not scare you. Most of the time (99.99%) a regexp with <20 characters will be the solution to your problem!

Also extremely good source to learn regexp properly: https://github.com/ziishaned/learn-regex

Example:

As an example, suppose we have a mutation in HGVS format

In [219…
```
s = 'NG_007400.1:g.8638G>T'
```

Let's create a function without regular expression to validate this kind of string:

In [233…

```
def validate(s):
    if s.count(':') != 1:
        return False

    s1,s2 = s.split(':')
    if s1.count('.') != 1:
        return False

    s11, s12 = s1.split('.')
    try:
        int(s12)
    except ValueError:
        return False

    if s2.count('.') != 1:
        return False

    s21, s22 = s2.split('.')
    if s21 not in ['c', 'g']:
        return False

    s221 = s22[:-3]
    s222 = s22[-3:]

    try:
        int(s221)
    except ValueError:
        return False

    if s222.count('>') != 1:
        return False

    s2221, s2222 = s222.split('>')

    bases = set('ACGT')

    if not s2221 in bases:
        return False

    if not s2222 in bases:
        return False

    return True
```

In [234…

```
validate('NG_007400.1:g.8638G>T')
```

Out[234… True

In [235…

```
validate('NG_007400.1:g.8638H>T')
```

Out[235… False

The same function with regular expressions:

In [231…

```python
def validate(s):
    m = re.fullmatch(r'\w+\.\d+:[cg]\.\d+[ACGT]>[ACGT]', s)
    return bool(m)

validate('NG_007400.1:g.8638G>T')
```

Out[231… True

In [232…

```python
validate('NG_007400.1:g.8638H>T')
```

Out[232… False

It is clear that regular expression offer fast and simple solution at string parsing and validation.