Serialization

Ας υποθέσουμε ότι έχουμε τη παρακάτω "πολύπλοκη" δομή:

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

Πως μπορούμε να αποθηκεύσουμε το a σε ένα αρχείο; Μπορούμε να τα αποθηκεύσουμε σε μορφή json:

In [2]:
```
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
```

Οπότε μπορούμε να σώσουμε το a_json σε ένα αρχείο:

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

Ή αλλιώς:

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]}}
```

Για όσους είναι στα windows, μπορείτε να γράψετε:

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

Αφού στείλουμε το αρχείο μπορεί ο παραλήπτης να το ανοίξει:

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'>
```

Αυτή η διαδικασία ονομάζεται serialization και μας επιτρέπει να διαμοιραζόμαστε δεδομένα μαζί με τη δομή τους.

Δεν μπορούν όλα να γίνουν json:

In [12]:
```
try:
    json.dumps({1,2,3,4}) # Sets cannot be serialized
except Exception as e:
    print (e)
```

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

In [13]:
```
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
```

Το φορμάτ json είναι πολύ δημοφιλές στον διαμοιρασμό "δομημένων δεδομένων". Δηλαδή δεδομένα που αποτελούνται από λίστες και λεξικά ή/και συνδοιασμό αυτών. Επίσης είναι πολύ συνηθισμένο μία βάση δεδομένων να μοιράζεται τα δεδομένα της σε αυτό το φορμάτ. Για παράδειγμα σε αυτό το link:

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

Μπορείτε να ρωτήσετε μία βάση δεδομένων για να σας δώσει μία λίστα από 1000 γονίδια τα οποία έχουν συσχετιστεί με τον καρκίνο. Το αποτέλεσμα το επιστρέφει σε φορμάτ json.

Εκτός από το json υπάρχουν και άλλα φορμάτ για την ανταλλαγή δομημένων δεδομένων. Μερικά παραδείγματα είναι το XML και το YAML.

Είδαμε όμως ότι το json μπορεί να δεχτεί μόνο λίστες και dictionaries. Μία άλλη επιλογή είναι η βιβλιοθήλη pickle η οποία μπορεί να κάνει serialize πολύ μεγαλύτερο πλήθος από δομές. Τα αρνητικά της είναι:

1. Είναι μόνο για python (ίσως αν ψάξετε να βρείτε και βιβλιοθήκες για άλλες γλώσσες)
2. Δεν είναι αναγνώσιμο απο ανθρώπους (σε αντίθεση με τη json).

Ας δούμε ένα παράδειγμα:

In [19]:
```
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)
```

Προσέξτε το 'wb' το οποίο σημαίνει εγγραφή σε δυαδικό (binary) φορμάτ. Σε αντίθεση με το σκέτο 'w' ή το 'wt' το οποίο σημείνει φορμάτ κειμένου (text).

Ας το κάνουμε τώρα un-pickle!

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

In [21]:
```
data[3](10) # Καλούμε τη συνάρτηση f !
```

Out[21]:
```
11
```

Η βιβλιοθήκε itertools

Η itertools περιέχει συναρτήσεις για να σας βοηθήσουν να κάνετε iterations και.. λούπες! Είναι μία από τις πιο χρησιμες βιβλιοθήκες κυρίως γιατί σας βοηθάει να απλοποιήσετε τον κώδικά σας. Πριν επιχειρήσετε να κάνετε κάποια πολύπλοκη επανάληψη (for μέσα σε for μέσα σε for...) ελέγξτε αν κάποια από τις συναρτήσεις της itertools μπορεί να σας βοηθήσει.

Πρόβλημα 1 (καρτεσιανό γινόμενο)

Μπαίνετε σε ένα μαγαζί με ρούχα. Το μαγαζί έχει 10 διαφορετικά ζευγάρια παπούτσια στο νούμερό σας και η τιμή τους είναι:

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

Το μαγαζί έχει 3 διαφορετικά τζην στο νούμερό σας και η τιμή τους είναι:

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

Το μαγαζί έχει 8 διαφορετικά μπλουζάκια στο νούμερό σας. Η τιμή του είναι:

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

Έσείς έχετε πάνω σας 100 ευρώ και πρέπει να πάρετε ένα από κάθε είδος. Πόσοι συνδυασμοί (παπούτσια, τζην και μπλουζάκι) ρούχων μπορείτε να αγοράσετε;

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]:
```
# Κλασσική λύση:
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]:
```
# Με 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
```

Πρόβλημα 2 (συνδοιασμοί)

Μπαίνετε σε ένα μαγαζί το οποίο πουλάει μόνο μπλουζάκια. Το είδος και η τιμή για το κάθε μπλουζάκι είναι:

```
shirts = [
```

```
    ('a', 22),
    ('b', 30),
    ('c', 83),
    ('d', 28),
    ('e', 72),
    ('f', 51),
    ('g', 61),
    ('h', 83),
    ('i', 25),
]
```

Εσείς μπορείτε να ξοδέψετε το πολύ 100ε και πρέπει να πάρετε ακριβώς 2. Ποια ζευγάρια μπορείτε να επιλέξετε;

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

#Ο κλασσικός τρόπος:
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]:
```
# Με 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
```

Πρόβλημα 3 (αντί while)

Ποιος είναι το άθροισμα όλων των πρώτων αριθμών που είναι μικρότεροι από 1000;

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

# Αρχικά φτιάχνουμε ένα generator για πρώτους αριθμούς:
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

# Φτιάχνουμε μία συνάρτηση που ελέγχει πότε θα σταματήσουμε:
def f(x):
    return x<1000

# Υπολόγισε το άθροισμα των πρώτων αριθμών, μέχρι να βρεθεί
# κάποιος πρώτος που να μην ικανοποιεί τη f
sum(takewhile(f, gen_primes()))
```

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

Regular Expressions

Τα Regular Expressions (ή αλλιώς regexp για συντομία) είναι μια βασική ιδέα στην επιστήμη υπολογιστών (υπάρχουν απο το 1956..). Είναι στην ουσία μία νέα γλώσσα με την οποία μπορείς να δηλώσεις κάποια patterns μέσα σε ένα string. Ειδικοί αλγόριθμοι αναλαμβάνουν να εντοπίσουν αυτά τα patterns με πολύ μεγάλη ταχύτητα. Τα regexp υλοποιούνται στη python στη βιβλιοθήκη `re` :

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

Με τα regular expressions μπορούμε να κάνουμε πάρα πολύ γρήγορα πολύπλοκες λειτουργίες πάνω σε strings. Αυτέ οι λειτουργίες είναι:

- Έλεγχος αν ένα string ακολουθεί ένα συγκεκριμένο format / pattern (π.χ. αποτελείται από 4 αριθμούς και 2 γράμματα
- Να πάρουμε ένα υπο-string. Για παράδειγμα από μια ημερομηνία γέννησης να εξάγουμε τη χρονιά
- Να πάρουμε όλα τα υπο-strings τα οποία ακολουθούν ένα πρότυπο. Για παράδειγμα από ένα τεράστιο αρχείο με κείμενο να εξάγουμε όλες τις ημερομηνίες που περιέχει.
- Να αντικαταστάσήσουμε ένα pattern από ένα string μέ κάποιο άλλο. Για παράδειγμα κάνε όλες τις ημερομηνίες οι οποίες είναι Μήνας/Μέρα/Χρονιά (αμερικανικό σύστημα) να είναι Μέρα/Μήνας/Χρονιά (ευρωπαϊκό σύστημα).

Τα regular expressions (regex) είναι πρωτίστως strings. Κάθε regex δηλώνει και ένα pattern. Για παράδειγμα το regexp: '\d' δηλώνει "ένας χαρακτήρας που είναι αριθμός". Ας το δούμε στη πράξη:

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

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

Με αυτή την εντολή στην ουσία λέμε: "Ψάξε αν υπάρχει τουλάχιστον ένας αριθμός μέσα στο string". Παρατηρούμε ότι μας επέστρεψε "κάτι". Θα δούμε στη συνέχεια τι είναι αυτό. Για αρχή μπορούμε να ελέγξουμε τι επιστρέφει αν ΔΕΝ βρει το pattern:

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

```
None
```

Αν δεν υπάρχει το pattern επιτρέφει None. Μπορούμε να επεκτείνουμε το pattern ζητώντας ένας αριθμό που να ακολουθείται με τον χαρακατήρα "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
```

Παρατηρούμε ότι:

- Στο 1ο το βρήκε. Με την εντολή search ζητάμε να βρει κάπου **οπουδήποτε** μέσα στο string.
- Στο δεύτερο δεν το βρήκε. Δεν υπάρχει πουθενά ένας αριθμός που να ακολουθείται από από το γράμμα "a".

Συνεχίζουμε ζητώντας ένας αριθμός ο οποίος να ακολουθείται είτε από το γράμμα "a" είτε με το γράμμα "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
```

Με τις αγκύλες λοιπόν δηλώνουμε ένα set από χαρακτήρες. Ζητάμε δηλαδή να βρεί έναν και μόνο ένα χαρακτήρα οποίος να ανήκει σε αυτό το set.

Συνεχίζουμε ζητώντας έναν οποιοδήποτε αριθμό ακολουθούμενο από οποιοδήποτε χαρακτήρα από a μέχρι και το k: Μέσα σε αγγύλες μπορούμε να δηλώσουμε 1 ή παραπάνω έυρη χαρακτήρων:

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
```

Συνεχίζουμε ζητώντας ένας αριθμό ο οποίος ακολουθείται από οποιοδήποτε χαρακτήρα εκτός από αυτούς που ανήκουν στο εύρος a-k. Για να το κάνουμε αυτό βάζουμε το carret (^) μέσα στις αγκύλες:

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'>
```

Συνεχίζουμε ζητώντας έναν αριθμό που να αποτελείται από οποιοδήποτε χαρακτήρα! Η τελεία "." σημαίνει "οποιοσδήποτε χαρακτήρας":

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
```

Συνεχίζουμε δηλώνοντας έναν αριθμό και ένα κενό ή tab ή new line. Ο ειδικός χαρακτήρας \s δηλώνει "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
```

Συνεχίζουμε δηλώνοντας έναν αριθμό ακολουθούμενο από οποιοδήποτε γράμμα το οποίο ΔΕΝ είναι ειδικός χαρακτήρας. Το pattern \w δηλώνει οποιοδήποτε χαρακτήρα από τα: `a-z` `A-Z` `0-9` και `_` :

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
```

Δηλαδή αντί να γράφουμε `[0-9]` για να δηλώνουμε όλους τους αριθμούς και `[a-zA-Z]` για δηλώνουμε όλα τα γράμματα χρησιμοποιούμε τα εξής:

`\d` είναι το ίδιο με: `[0-9]`

`\w` είναι το ίδιο με: `[a-zA-Z0-9_]`

`\s` είναι το ίδιο με: `[ \t\n\r\f\v]`

Επαναλαμβανόμενα μοτίβα

Μπορούμε να ζητήσουμε σε ένα pattern να βρεί πολλαπλές επαναλήψεις από ένα σύνολο χαρακτήρων. Για παράδειγμα μπορούμε να ρωτήσουμε: να έχει 1 ή παραπάνω αριθμούς ακολουθούμενοι από το γράμμα "a". Αυτό το κάνουμε με τον ειδικό χαρακτήρα `+` :

In [88]:
```
a = re.search(r'\d+a', 'hello123431ahello') # Polloi ari8moi meta "a"
print (a)
```

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

In [90]:
```
a = re.search(r'\d+a', 'hello1ahello') # enas ari8mos meta "a"
print (a)
```

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

In [91]:
```
a = re.search(r'\d+a', 'helloahello') # Kanenas ari8mos meta "a" (den kanei
print (a)
```

```
None
```

Αν αντί για `+` βάλουμε `*` . Τότε δηλώνουμε: "κανένα ή πολλά". Δηλαδή ενώ με το `+` πρέπει υποχρεωτικά να υπάρχει τουλάχιστον 1, με το `*` μπορεί να μην υπάρχει και κανένα:

In [92]:
```
a = re.search(r'\d*a', 'hello444a') # polloi ari8moi kai meta to a ! OK!
print (a)
```

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

In [93]:
```
a = re.search(r'\d*a', 'hello4a') # enas ari8mow kai meta to a ! OK!
print (a)
```

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

In [94]:
```
a = re.search(r'\d*a', 'helloa') # kanenas ari8mos kai meta to a ! PALI OK!
print (a)
```

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

In [95]:
```
a = re.search(r'\d*a', 'hel555lo') # Yparxei ari8mos alla den yparxei to a.
print (a)
```

```
None
```

Μπορούμε επίσης να δηλώσουμε "ένα ή κανένα". Για παράδειγμα θέλουμε είτε κανένα αριθμό και μετά το "a", είτε έναν αριθμό και μετά το "a". Αυτό το κάνουμε με τον χαρακτήρα `?` :

In [103…
```
a = re.search(r'b\d?a', 'b5a') # yparxei to b meta enas ari8mos kai meta to
print (a)
```

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

In [104…
```
a = re.search(r'b\d?a', 'ba') # yparxei to b meta kanenas ari8mos kai meta
print (a)
```

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

In [106…
```
a = re.search(r'b\d?a', 'b65a') # yparxei to b meta polloi ari8moi kai meta
print (a)
```

```
None
```

Τέλος μπορούμε να ζητήσουμε ένα σύνολο από χαρακτήρες να υπάρχει συγκεκριμμένο

αριθμό από επαναλήψεις!

In [107…
```
a = re.search(r'ba{3}b', 'baaab') # b treis fores to a kai me ta b
print (a)
```
```
<re.Match object; span=(0, 5), match='baaab'>
```

In [108…
```
a = re.search(r'ba{3}b', 'baab') # b treis fores to a kai me ta b
print (a)
```
```
None
```

In [109…
```
a = re.search(r'ba{3}b', 'baaaab') # b treis fores to a kai me ta b
print (a)
```
```
None
```

ή να δηλώσουμε ένα εύρος από επαναλήψεις:

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

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

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

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

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

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

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

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

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

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

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

In [126…
```
a = re.search(r'ba{,2}b', 'bb') # Ζητάμε 2 ή λιγότερα
print (a)
```
```
<re.Match object; span=(0, 2), match='bb'>
```

Ας δούμε λίγο το παρακάτω:

In [129…
```
a = re.search(r'ba+b', 'hellobaaaaabhello')
print (a)
```
```
<re.Match object; span=(5, 12), match='baaaaab'>
```

Με τη συνάρτηση .group μπορούμε να βρούμε ΤΙ έκανε match. Βάζοντας σαν παράμετρο τη τιμή 0 μας επιστρέφει όλο το string το οποίο έκανε match:

In [133…
```
a.group(0)
```
Out[133…
```
'baaaaab'
```

Εδώ παρατηρούμε το εξής: το a+ "έπιασε" όλα τα "a" αυτό ονομάζεται greedy search. Η python γενικότερα θα προσπαθήσει να κάνει match όσο το δυνατόν περισσότερους χαρακτήρες γίνεται. Αυτό μπορεί να μας δημιουργήσει προβλήματα!. Για παράδειγμα. Έστω το string:

In [135…
```
s = 'gene:G1 function: F1, gene:G2 function:F2, gene:G3 function:F3'
```

Θέλουμε να πάρουμε το όνομα του πρώτου γονιδίου. Άρα να αρχίζει από `gene:` μετά ένα ακαθόριστο πλήθος από χαρακτήρες και μετά το 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
```

Τι έγινε εδώ;; παρατηρούμε ότι αυτό που επέστρεψε όντως ακολουθεί το πρότυπο αφού ξεκινά από gene και τελειώνει σε function. Αυτό έγινε επειδή η python προσπάθησε να επιστρέψει το μεγαλύτερο δυνατό match. Δηλαδή και το string `gene:G1 function` ακολουθεί το πρότυπο που βάλαμε, αλλά δεν είναι το μεγαλύτερο δυνατό. Μπορούμε να αποτρέψουμε αυτή τη συμπεριφορά βάζοντας ένα `?` μετά το `+` :

In [138…
```
a = re.search(r'gene:.+?function', s)
print (a.group(0))
```
```
gene:G1 function
```

Ο χαρακτήρας `?` μετά από `+,*,?,{}` λέει στη python "φέρουμε το μικρότερο δυνατό". Δείτε αυτά τα παραδείγματα:

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') # epilegei to megalutero --> 4
print (a.group(0))
```
```
b1234
```

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

Αρχή και τέλος

Μπορούμε να δηλώσουμε ότι ένα pattern πρέπει να υπάρχει στην αρχή του string, αν βάλουμε στην αρχή του pattern τον χαρακτήρα ^ :

In [154…
```
a = re.search('^\d', '4hello') # Prepei o ari8mos na einai sthn arxh! OK!
print (a)
```
```
<re.Match object; span=(0, 1), match='4'>
```

In [155…
```
a = re.search('^\d', 'h4ello') # Prepei o ari8mos na einai sthn arxh! NOT O
print (a)
```
```
None
```

Ομοίως μπορούμε να δηλώσουμε ότι το pattern θα είναι στο τέλος με τον χαρακτήρα $:

In [156…
```
a = re.search('\d$', 'hello4') # Prepei o ari8mos na einai sto telos! OK!
print (a)
```
```
<re.Match object; span=(5, 6), match='4'>
```

In [157…
```
a = re.search('\d$', 'hell4o') # Prepei o ari8mos na einai sto telos! NOT O
print (a)
```
```
None
```

Ο τελεστής ή --> |

Πολλές φορές θέλουμε ένα pattern να κάνει match ΚΑΤΙ ή ΚΑΤΙ ΑΛΛΟ. Αυτό γίνεται βάζοντας σε παρενθέσεις τα δύο patterns και χρησιμοποιώντας τον τελεστή | :

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

In [159…
```
a = re.search(r'(ab)|(kl)', 'kl') # ab ή kl
print (a)
```
```
<re.Match object; span=(0, 2), match='kl'>
```

In [160…
```
a = re.search(r'(ab)|(kl)', 'al') # ab ή kl
print (a)
```
```
None
```

Μπορούμε να κάνουμε εμφωλευμένα | :

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'>
```

Παίρνοντας πεδία μέσα από patterns

Πολλές φορές θέλουμε να εξάγουμε υπο-πεδία από ένα string. Για να το κάνουμε αυτό βάζουμε παρενθέσεις στα κομμάτια του pattern που θέλουμε να εξάγουμε:

In [167…
```
plate_number = ' This is my plate number: ABE 1234 hello'

a = re.search(r'(\w+) (\d+)', plate_number)
```

Στη συνέχεια μπορούμε να χρησιμοποιήσουμε τη group για να πάρουμε αυτά τα πεδία:

Όλο το string το οποίο έκανε match:

In [168…
```
a.group(0)
```
Out[168…
```
'ABE 1234'
```

Το string που έκανε match στη 1η παρένθεση:

In [169…
```
a.group(1)
```
Out[169…
```
'ABE'
```

το string που έκανε match στη 2η παρένθεση:

In [170…
```
a.group(2)
```
Out[170…
```
'1234'
```

Οι συναρτήσεις match, exactmatch, findall και sub:

Η συνάρτηση search που έχουμε δει μέχρι στιγμής χρησιμοποιείται για να βρούμε ένα pattern μέσα σε ένα string. Μία άλλη χρήσιμη συνάρτηση είναι η `fullmatch` η οποία κάνει match **μόνο αν** όλο το string κάνει match το pattern:

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'>
```
Η fullmatch κάνει το ίδιο με τη search αν εσωκλείσουμε το pattern μέσα σε ^$:

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'>
```
Η συνάρτηση match κοιτάει να δει αν το pattern είναι στην αρχή του string. Ισοδυναμεί με το να χρησιμοποιήσουμε τη search και να βάλουμε το ^ μπροστά από το 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
```
Τέλος η συνάρτηση sub, αλλάζει το κομμάτι που έχει γίνει match σε ένα string με ένα άλλο string. Μπορούμε λοιπόν να κάνουμε "capture" τα groups μέ παρενθέσεις μέσα στο pattern και μετά να αναφερθούμε σε αυτό με τον χαρακτήρα \ και τον αριθμό της παρένθεσης:

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'
```

Τέλος με τη συνάρτηση findall μπορούμε να βρούμε όλα τα matches ενός pattern μέσα σε string:

```
In [199… s = 'dg +5aaghqq4  ajdfhal+48f4++85tyru+4867dhgjghi4yifhl4i8+hdji74rl48ru'
re.findall(r'\+\d+', s) #  Όλοι οι αριθμοί που έχουν ένα "+" μπροστά τους
```

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

Μα τι είναι επιτέλους αυτό το r που βάζεις μπροστά από κάθε pattern;

Όπως έχουμε δει μπορούμε να βάλουμε "special" χαρακτήρες μέσα σε ένα string. Για παράδειγμα μπορούμε να βάλουμε το "Enter" (ή αλλιώς new line):

```
In [200… s = 'a\nb'
print (s)
```

```
a
b
```

Ομοίως αν θέλουμε ένα string να έχει τον χαρακτήρα `\` , πρέπει να τον βάλουμε δύο φορές:

```
In [204… s = 'a\\b'
print (s)
```

```
a\b
```

Τι γίνεται όμως αν θέλουμε να δούμε αν ένα string περιέχει τον χαρακτήρα `\` ; Ο χαρακτήρας αυτός είναι ειδικός χαρακτήρας ΚΑΙ για τη python αλλά ΚΑΙ για τα regular expressions (Προσέξτε το `\+` που βάλαμε παραπάνω στην findall για να κάνουμε match τον χαρακτήρα `+`). Για να δηλώσουμε λοιπόν τον χαρακτήρα `\` μέσα σε ένα pattern θα πρέπει να τον κάνουμε escape και να φτιάξουμε το pattern: `\\` . Δηλαδή όπως ακριβώς κάναμε match το `+` με το pattern `\+` , έτσι κάνουμε match το `\` με το `\\` . Άρα θα πρέπει να "φτιάξουμε" ένα string το οποίο όταν το τυπώσουμε να τυπώσει: `\\` . Αυτό το string είναι:

```
In [211… s = '\\\\'
print (s)
```

```
\\
```

Άρα για να κάνουμε match τον χαρακτήρα `\` πρέπει να γράψουμε το εξής:

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

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

Αυτό μπορεί να είναι αρκετά.. μπερδεψηματικό και να είναι πηγή λάθους. Δυστυχώς αυτό το πρόβλημα είναι κοινό για όλες τις γλώσσες προγραμματισμού εδώ και πολλά χρόνια. Συλλογικά αυτό το πρόβλημα αναφέρεται σαν το σύνδρομο της στραβής οδοντογλυφίδας(!!). Μία λύση που έχει η python είναι να μπορείς να δηλώσεις ένα string σαν raw (ωμό). Ένα raw string δηλώνεται βάζοντας το r μπροστά και σημαίνει ότι δεν περιέχει κανέναν άλλο χαρακτήρα πέρα από αυτούς που έχει!

In [209…
```
s = r'a\nb'
print (s)
```

```
a\nb
```

Με αυτόν τον τρόπο μπορούμε να δηλώνουμε regular expressions χωρίς να ανησυχούμε μήπως οι ειδικοί χαρακτήρες της python συγχέονται με τους ειδικούς χαρακτήρες των 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'>
```

Σε περίπτωση που όλα αυτά είναι δυσνόητα (πολύ λογικό) μπορείτε να θυμάστε το εξής: **Όταν χρησιμοποιούμε regular expressions, πάντα δηλώνουμε τα patterns σαν raw strings βάζοντας ένα r μπροστά.**

Επίσης το επίσημο documentation της python εξηγεί πολύ όμορφά τα raw strings.

Συνέχεια

Έχουμε πει λιγότερα από τα μισά που περιέχει τόσο η θεωρία των regular expressions, όσο και η υποστήριξή τους από τη python. Μπορείτε να διαβάσετε περισσότερα για:

- Look ahead and look begind regular expressions
- Named groups
- Non capturing parenthesis
- Compilation flags. Για παράδειγμα η τελεία θα πρέπει να "πιάνει" το enter; Ή πως μπορώ να κάνω match αγνοώντας τη διαφορά μεταξύ κεφαλαίων και μικρών γραμμάτων;
- Greedy vs. non-greedy (τα εξηγήσαμε λίγο εδώ)
- compile. Μπορείτε να κάνετε compile ένα πολύπλοκο regexp για να το κάνετε match πολύ πιο γρήγορα
- debug. Υπάρχουν και ειδικά site που σας βοηθάν να γράψετε και να διορθώσετε το regexp σας: debuggex, pythex
- comments. Μπορείτε να βάζετε comments **μέσα** σε ένα regexp

Μα καλά πόσο πολύπλοκο μπορεί να γίνει ένα regexp; Απάντηση: Αρκετά πολύπλοκο, αλλά αυτό δεν πρέπει να σας τρομάζει. Τις περισσότερες φορές (99.99%) ένα regexp με <20 χαρακτήρες θα είναι η λύση στο πρόβλημά σας!

Επίσης εξαιρετικά καλή πηγή για να μάθετε σωστά regexp: https://github.com/ziishaned/learn-regex

Παράδειγμα:

Σαν παράδειγμα ας υποθέσουμε ότι έχουμε μία μετάλλαξη σε HGVS format

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

Ας φτιάξουμε μία συνάρτηση χωρίς regular expression για να κάνουμε validate τέτοιου είδους 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

Η ίδια συνάρτηση με regular expressions:

In [231…

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

Out[232… False