Ελληνικό Μεσογειακό Πανεπιστήμιο
Σχολή Μηχανικών / Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

ΗΜΥ01Κ06
Επιστημονικός Προγραμματισμός με Python

Διάλεξη Δέκατη
Κανονικές εκφράσεις

Φθινόπωρο 2025

Κανονικές Εκφράσεις (Regular Expressions)

- Μια χρήσιμη γλωσσολογική διευκρίνιση πριν ξεκινήσουμε -

regular:

1. *normal, usual (κανονικός, φυσιολογικός)*
2. *methodical, systematic, structured, well ordered, well organized*

- Στην περίπτωσή μας το *regular* ερμηνεύεται ως *methodical, systematic* κλπ.

- Επομένως ο όρος "*κανονικές εκφράσεις*" δεν σημαίνει φυσιολογικές ή συνηθισμένες εκφράσεις, αλλά μάλλον *δομημένες εκφράσεις που ορίζονται και διέπονται από συγκεκριμένους κανόνες (και μάλιστα ιδιαίτερα αυστηρούς)*

- Παρόλο που έχει επικρατήσει ο όρος "*κανονικές εκφράσεις*", θα ήταν πιο δόκιμο αν ονομαζόταν *"κανονιστικές εκφράσεις"* ή *"συστηματικές εκφράσεις"*

Κανονικές Εκφράσεις

- Τεχνική που αναπτύχθηκε στη θεωρητική επιστήμη των υπολογιστών και στη *θεωρία τυπικών γλωσσών* (*formal language theory*)
- Οι *Κανονικές Εκφράσεις*(*) *(εφεξής για συντομία Κ.Ε.)* είναι μια *ακολουθία χαρακτήρων* που ορίζουν έναν *κανόνα / μοτίβο / πρότυπο αναζήτησης* (*search pattern*)
- Χρησιμοποιούνται για να αναλύουν συμβολοσειρές και να εντοπίζουν αν υπακούουν σε συγκεκριμένους *κανόνες που ονομάζονται μοτίβα ή πρότυπα* (patterns)
- Υποστηρίζονται από πολλές γλώσσες προγραμματισμού, αποτελώντας ουσιαστικά μια *μικροσκοπική, πολύ εξειδικευμένη* γλώσσα *ενσωματωμένη* σε αυτές
- Στην Python διατίθεται μέσω του δομοστοιχείου `re` και αποτελούν *εξειδίκευση / επέκταση* των μεθόδων χειρισμού συμβολοσειρών `find()`, `index()` και `replace()`

(*) Στα Αγγλικά μια Κ.Ε. ονομάζεται με διάφορους τρόπους: *Regular Expression* ή *RE*, ή *regex*, ή *regex pattern*

Αναζήτηση κειμένου με τη βοήθεια κανονικών εκφράσεων

Παραδοσιακός τρόπος αναζήτησης *(αυτός που ξέραμε μέχρι τώρα)*

`txt` : Αρχική συμβολοσειρά που περιέχει κάποιο βασικό κείμενο πχ `'engineering'`

`s` : Απλό κείμενο αναζήτησης *(μοναδικό και αμετάβλητο)* πχ. `'gin'`

`txt.find(s)` Ελέγχει αν η συμβολοσειρά `txt` περιέχει το κείμενο `s` *επιστρέφει 2 (γιατί;)*

Αναζήτηση με Κανονικές Εκφράσεις (Κ.Ε.)

`txt` : Αρχική συμβολοσειρά που περιέχει κάποιο βασικό κείμενο

`r` : Κείμενο αναζήτησης με μορφή Κανονικής Έκφρασης(*) πχ. `'χά[πλδ]ι'`

(*)Κ.Ε: συμβολοσειρά στην οποία συγκεκριμένοι χαρακτήρες ερμηνεύονται ως χαρακτήρες ελέγχου επιτρέποντας έτσι πολλαπλές παραλλαγές στο ταίριασμα

`re.search(r,txt)` Ελέγχει αν η συμβολοσειρά `txt` περιέχει το κείμενο `'χάπι'` είτε το κείμενο `'χάλι'` είτε το κείμενο `'χάδι'`

Μια Κ.Ε. χωρίς ειδικούς χαρακτήρες, δεν διαφέρει από ένα απλό κείμενο αναζήτησης

Παράδειγμα: Έλεγχος εγκυρότητας διευθύνσεων email

πχ έστω η συμβολοσειρά `m = 'john_smith98@gmail.com'`

Με τις μεθόδους χειρισμού συμβολοσειρών που ξέρουμε, μπορούμε να κάνουμε πράγματα όπως:

`m.find('@')` επιστρέφει την τιμή `12` (το @ είναι ο 13ος χαρακτήρας)

`m.replace('@', '#')` επιστρέφει `'john_smith98#gmail.com'`

Όμως πώς μπορούμε να ελέγξουμε αν η συμβολοσειρά αυτή είναι μια έγκυρη διεύθυνση email;

Μια *έγκυρη διεύθυνση email* πρέπει να υπακούει σε συγκεκριμένους κανόνες όπως πχ.

- να είναι της μορφής *<όνομα χρήστη>* @ *<όνομα περιοχής>*
- το *<όνομα χρήστη>* και *<όνομα περιοχής>* να περιέχουν μόνο αλφαριθμητικούς χαρακτήρες και τους χαρακτήρες "." και "_"
- ...

Χρειαζόμαστε ισχυρότερα εργαλεία για να κάνουμε τέτοιους ελέγχους εύκολα και αποτελεσματικά

Παράδειγμα: Έλεγχος εγκυρότητας διευθύνσεων email

Έστω ότι μια διεύθυνση email είναι έγκυρη εφόσον υπακούει στους εξής κανόνες:

- είναι της μορφής *<όνομα χρήστη>* @ *<όνομα περιοχής>*
- το *<όνομα χρήστη>* περιέχει μόνο αλφαριθμητικούς χαρακτήρες και προαιρετικά μια τελεία ή κάτω παύλα η οποία πρέπει να βρίσκεται ανάμεσά τους

  πχ. `john_smith98`, `Kostas.19`, `MontyPython` …

- το *<όνομα περιοχής>* περιέχει μόνο αλφαριθμητικούς χαρακτήρες και υποχρεωτικά μια και μόνο τελεία η οποία πρέπει να βρίσκεται ανάμεσά τους. Μετά την τελεία ακολουθούν υποχρεωτικά 2 έως 3 χαρακτήρες

  πχ. `gmail.com`, `hmu.gr` …

Μπορούμε να δημιουργήσουμε μια *έκφραση που να περιγράφει αυτούς τους κανόνες*, *(κανονική έκφραση)* και να την χρησιμοποιούμε για να ελέγχουμε την εγκυρότητα μιας διεύθυνσης email; *Φυσικά και μπορούμε όπως θα δούμε αμέσως τώρα!*

Παράδειγμα: Έλεγχος εγκυρότητας διευθύνσεων email

Για το παράδειγμά μας, αυτό γίνεται ως εξής:

```
import re  #Δομοστοιχείο που περιέχει συναρτήσεις και μεθόδους για χρήση  regular expressions

email_addr = 'john_smith98@gmail.com'

regex = '^[a-z0-9]+[._]?[a-z0-9]+[@][a-z0-9]+[.][a-z0-9]{2,3}$'

if re.search(regex, email_addr):

    print('Email is valid')

else:

    print('Email is NOT valid')
```

θα εκτυπώσει `Valid Email`

Η ερμηνεία της θα ακολουθήσει

Αν πχ είχαμε μια ολόκληρη λίστα mail_list με διευθύνσεις email θα μπορούσαμε εύκολα να ελέγξουμε την εγκυρότητά τους ως εξής:

```
for m in mail_list

    if re.search(regex,m):

        print(m,'is a valid email address')

    else:

        print(m,'is NOT a valid email address')
```

Ερμηνεία της κανονικής έκφρασης του παραδείγματος

```
regex = '^[a-z0-9]+[._]?[a-z0-9]+[@][a-z0-9]+[.][a-z0-9]{2,3}$'
```

1 2 3 4 5 6 7

Ερμηνεία:

1. `^[a-z0-9]+` *αρχίζει από ένα ή περισσότερα* πεζά αγγλικά γράμματα η/και ψηφία 0…9
2. `[._]?` ακολουθεί *0 ή 1 φορές (δηλαδή προαιρετικά)* είτε μια τελεία, είτε μια κάτω παύλα
3. `[a-z0-9]+` ακολουθούν ένα ή περισσότερα πεζά αγγλικά γράμματα η/και ψηφία 0…9
4. `[@]` ακολουθεί -υποχρεωτικά- το σύμβολο @
5. `[a-z0-9]+` ακολουθούν ένα ή περισσότερα πεζά αγγλικά γράμματα η/και ψηφία 0…9
6. `[.]` ακολουθεί -υποχρεωτικά- μια τελεία
7. `[a-z0-9]{2,3}$` *τελειώνει με 2 ή 3* πεζά αγγλικά γράμματα η/και ψηφία 0…9

`{1,}` συμβολίζεται και με `+` `{0,1}` συμβολίζεται και με `?` `{0,}` συμβολίζεται και με `*`

Δημιουργία κανονικών εκφράσεων

Στην απλούστερη δυνατή μορφή, μια Κ.Ε. είναι μια *συνηθισμένη συμβολοσειρά.* Στην περίπτωση αυτή η χρήση της δεν διαφέρει από τις απλές μεθόδους εύρεσης / αντικατάστασης της Python

```
txt = 'Contestant'    # contestant σημαίνει 'διαγωνιζόμενος' πχ σε τηλεπαιχνίδι
regex = 'test'
if re.search(regex ,txt):
    print('Found', regex, 'in', txt )
else:
    print('Not found', regex, 'in', txt )
```

με άλλα λόγια η έκφραση `re.search(regex ,txt)` θα πάρει την τιμή `True`

θα εκτυπώσει `Found test in Contestant`

Εναλλακτικά μπορώ να ζητήσω να μου εκτυπώσει το αποτέλεσμα της μεθόδου `re.search`

```
a = re.search(regex,txt)
print(a)
<re.Match object; span=(3, 7), match='test'>
```

Αργότερα θα δούμε πως μπορούμε να εξάγουμε όλες τις χρήσιμες πληροφορίες που περιέχονται σε αυτό το αντικείμενο

Ειδικοί χαρακτήρες σε μια Κανονική Έκφραση

Οι επόμενοι χαρακτήρες ονομάζονται *μεταχαρακτήρες* (*metacharacters*)
και έχουν *ειδική σημασία* όταν βρίσκονται μέσα σε μια κανονική έκφραση

[ ]	class	Ορίζουν μια κλάση χαρακτήρων (character class) μέσα σε μια Κ.Ε.
{ }	repeat	Καθορίζουν το πόσες φορές μπορεί να επαναλαμβάνεται ο προηγούμενος χαρακτήρας, κλάση ή ομάδα χαρακτήρων
? + *		Ειδικοί χαρακτήρες για συντομογραφία απλών επαναλήψεων
^	start	Ταίριασμα μόνο αν η συμβολοσειρά "αρχίζει από"
$	end	Ταίριασμα μόνο αν η συμβολοσειρά "τελειώνει σε"
.	any	Οποιοσδήποτε χαρακτήρας εκτός από NEWLINE (/n)
|	or	Συνένωση δυο κανονικών εκφράσεων
( )	capturing groups	Ορίζουν περιοχές της Κ.Ε. για τμηματική επιστροφή κατά το ταίριασμα
/	/x (Esc chars)	Ειδικοί χαρακτήρες διαφυγής. *(Το x παίρνει μόνο συγκεκριμένες τιμές)*

Κλάσεις χαρακτήρων [ ]

Ορίζονται μέσα σε [ ] και χρησιμοποιούνται όταν θέλουμε να δηλώσουμε ότι ενας χαρακτήρας σε μια Κ.Ε. μπορεί να είναι *οποιοσδήποτε* από μια συγκεκριμένη *συλλογή χαρακτήρων*

Πχ. η Κ.Ε. `'χά[πλδ]ι'` "ταιριάζει με" *(εντοπίζει)* τις συμβολοσειρές `'χάπι'` `'χάλι'` `'χάδι'`

Μια κλάση μπορεί να περιέχει είτε μεμονωμένους χαρακτήρες πχ. `[asdf]` είτε περιοχές χαρακτήρων *(από - έως)* πχ. `[a-k]`, είτε οποιονδήποτε συνδυασμό τους

Πχ. `[a-zA-Z0-9]` *(όλοι οι Αγγλικοί αλφαριθμητικοί χαρακτήρες – κεφαλαία & πεζά)*

Παράδειγμα: η Κ.Ε. `'test[1-4!]0'` "ταιριάζει με" *(μας εντοπίζει)* τις συμβολοσειρές `'test10'` `'test20'` `'test30'` `'test40'` και `'test!0'`

Οι μεταχαρακτήρες που βρίσκονται μέσα σε μια κλάση χαρακτήρων (δηλαδή μέσα σε [ ]) *χάνουν την ειδική σημασία τους* και αντιμετωπίζονται σαν *συνηθισμένοι χαρακτήρες*. Έτσι π.χ. η κλάση `[a.m$]` ταιριάζει τους χαρακτήρες `'a'`, `'.'`, `'m'` και `'$'` παρόλο που το `'.'` και το `'$'` έχουν ειδική σημασία *(είναι μεταχαρακτήρες)*. Εξαίρεση φυσικά αποτελούν οι μεταχαρακτήρες [και] που χρησιμοποιούνται για να οριοθετήσουν μια κλάση. Αν χρειαστεί να τους περιλάβουμε στην κλάση, τους γράφουμε ως \[και \] αντίστοιχα

Ειδικοί χαρακτήρες επανάληψης { }

Τοποθετούνται αμέσως μετά έναν χαρακτήρα *(ή μια κλάση)* [ ] για να ορίσουν μια ***επανάληψη***, δηλαδή το πόσες φορές μπορεί αυτός *(αυτή)* να επαναλαμβάνεται

Γενικά `x{n1,n2}` σημαίνει ότι ο χαρακτήρας *(ή η κλάση)* `x` μπορεί να επαναλαμβάνεται από `n1` μέχρι ***και*** `n2` φορές *και όχι μέχρι n2 - 1 που ενδεχομένως θα περιμέναμε όπως ισχύει γενικότερα στην Python (δείτε το σαν εξαίρεση)*

Κ.Ε.	Συμβολοσειρές που "ταιριάζει" *(εντοπίζει)*
`'ab{0,3}c'`	`'ac'`, `'abc'`, `'abbc'` και `'abbbc'`
`'a[bc]{1,2}d'`	`'abd'`, `'acd'`, `'abbd'`, `'abcd'`, `'acbd'` και `'accd'`

Αν στις { } : παραλείψουμε *το κάτω όριο*, εννοείται το *μηδέν*, παραλείψουμε *το πάνω όριο* εννοείται το *άπειρο*

Έτσι πχ οι Κ.Ε. `'ab{0,3}c'` και `'ab{,3}c'` είναι ισοδύναμες

Ειδικοί χαρακτήρες επανάληψης ? + και *

`?` : Συντομογραφία του `{0,1}`

Επανάληψη του χαρακτήρα, της κλάσης ή του τμήματος 0 ή 1 φορές. (*)

`+` : Συντομογραφία του `{1,}`

Επανάληψη του χαρακτήρα, της κλάσης ή του τμήματος 1 ή περισσότερες φορές

`*` : Συντομογραφία του `{0,}`

Επανάληψη του χαρακτήρα, της κλάσης ή του τμήματος 0 ή περισσότερες φορές

Κ.Ε.	Συμβολοσειρές που "ταιριάζει" *(εντοπίζει)*
'ab?c'	'ac' και 'abc'
'ab+c'	'abc', 'abbc', 'abbbc', 'abbbc' ...
'ab*c'	'ac', 'abc', 'abbc', 'abbbc', 'abbbc' ...

(*) Επί της ουσίας αυτό σημαίνει *προαιρετική χρήση* του χαρακτήρα της κλάσης ή της ομάδας

Ειδικοί χαρακτήρες ^ και $

`^` : Ταιριάζει *(εντοπίζει)* ένα μοτίβο χαρακτήρων στην *αρχή* μιας γραμμής σε μια συμβολοσειρά

πχ η Κ.Ε. `'^be'` ταιριάζει *(εντοπίζει)* τα `'bear'`, `'be'`, `'beep'`, αλλά όχι το `'rebel'`

`$` : Ταιριάζει *(εντοπίζει)* ένα μοτίβο χαρακτήρων στο *τέλος* μιας γραμμής σε μια συμβολοσειρά

πχ η Κ.Ε. `'ing$'` ταιριάζει *(εντοπίζει)* τα `'following'`, `'making'`, αλλά όχι το `'kings'`

ΠΡΟΣΟΧΗ: Αν το ^ είναι πρώτο ***μέσα σε μια κλάση*** (π.χ. `[^abc]`) τότε αποκτά διαφορετική ερμηνεία, και σημαίνει *"όλοι εκτός τους επόμενους χαρακτήρες"*
(Η Κ.Ε. στο παράδειγμά μας ταιριάζει οποιονδήποτε χαρακτήρα εκτός των a, b και c)

Ειδικοί χαρακτήρες `.` και `|`

`.` : (Τελεία) Ταιριάζει οποιονδήποτε χαρακτήρα εκτός από *newline*.

Μπορούμε να αναγκάσουμε την τελεία να ταιριάζει και το newline με το flag re.DOTALL *(βλ παρακάτω)*

πχ η Κ.Ε. `'A.t'` ταιριάζει *(εντοπίζει)* όλες τις συμβολοσειρές 3 χαρακτήρων που αρχίζουν από `A` και τελειώνουν σε `t`.

`'Ant'` `'Art'` `'Alt'` `'Apt'` `'A#t'` κλπ.

Θυμόμαστε όμως ότι: αν ενας ειδικός χαρακτήρας είναι μέσα σε μια *κλάση* *(δηλαδή μέσα σε `[]`)*, τότε *χάνει την ειδική σημασία του*

πχ. η Κ.Ε. `'S[A.t]'` ταιριάζει τις συμβολοσειρές `'SA'`, `'S.'` και `'St'`

`|` : Τελεστής **or** - λογική διάζευξη Κ.Ε. Αν reg_A και reg_B είναι Κ.Ε., τότε reg_A | reg_B επιστρέφει όποια συμβολοσειρά ταιριάζει είτε με την reg_A είτε με την reg_B

πχ η Κ.Ε. `'^This|^That'` ταιριάζει *(εντοπίζει)* τις συμβολοσειρές που ξεκινάνε είτε από `'This'` είτε από `'That'`

Οι παρενθέσεις σε μια κανονική έκφραση (capturing groups)

Όταν μια Κ.Ε. "ταιριάσει" *(εντοπίσει)* ένα τμήμα μιας συμβολοσειράς, το τμήμα αυτό επιστρέφεται ολόκληρο

πχ. αν η Κ.Ε. `'TH209.5'` εφαρμοστεί στη συμβολοσειρά `'Ο Α.Μ είναι TH20945'` θα ταιριάσει *(εντοπίσει)* και θα μας επιστρέψει το τμήμα `'TH20945'`

Βάζοντας παρενθέσεις `(  )` σε ένα -ή και περισσότερα- τμήματα μιας Κ.Ε. *δεν αλλάζει σε κάτι ο τρόπος λειτουργίας της Κ.Ε.*, όμως σε περίπτωση εντοπισμού μας επιστρέφει *μόνο εκείνα τα κομμάτια που αντιστοιχούν στις παρενθέσεις* *(capturing groups*)*

Επομένως μια Κ.Ε. με *παρενθέσεις* όπως η `'TH(209.5)'` αν εφαρμοστεί στην αρχική συμβολοσειρά, θα ταιριάσει *(εντοπίσει)* και πάλι το ίδιο τμήμα της , όμως αυτή τη φορά θα μας επιστρέψει *μόνο το κομμάτι που αντιστοιχεί στις παρενθέσεις*, δηλ. το `'20945'`

Αντίστοιχα μια Κ.Ε. με *δυο ζεύγη παρενθέσεων* όπως η `'(TH)20(9.5)'` θα μας επιστρέψει *μια πλειάδα με τα δυο κομμάτια* που αντιστοιχούν σε αυτές τις παρενθέσεις `('TH','945')`

*Τα capturing groups επιστρέφονται είτε μέσω της ιδιότητας `.group()` του αντικειμένου `matchobj` που επιστρέφουν οι συναρτήσεις `re.search()` και `re.match()`, είτε ως πλειάδες στη λίστα που επιστρέφει η `re.findall()`

Παράδειγμα με απλό capturing group

Εξαγωγή των Α.Μ. φοιτητών του ΗΜΜΥ/ΕΛΜΕΠΑ από μια συμβολοσειρά s με διάφορα emails

```
s = 'th25932@edu.hmu.gr th30915@edu.hmu.gr th23918@edu.hmu.gr ab123@gmail.com'

r1 = '(th2[0-9]{4})@edu.hmu.gr'  # Έγκυρο email φοιτητή ΗΜΜΥ/ΕΛΜΕΠΑ
print(re.findall(r1, s))
```

θα εκτυπώσει `['th25932', 'th23918']` *# τα ΑΜ των φοιτητών του ΗΜΜΥ*

```
r2 = 'th(2[0-9]{4})@edu.hmu.gr'
print(re.findall(r2, s))
```

θα εκτυπώσει `['25932', '23918']` *# μόνο οι αριθμοί χωρίς το 'th'*

Η μέθοδος `findall` *(θα την δούμε αργότερα αναλυτικά)*, επιστρέφει μια λίστα με όλα τα τμήματα μιας συμβολοσειράς που "ταιριάζουν" με μια Κ.Ε.

Παράδειγμα με πολλαπλά capturing groups

Εξαγωγή user-name και domain-name από συμβολοσειρά που περιέχει emails

```
r = 'email: ([a-z0-9._]+)@([a-z0-9]+.[a-z0-9]{2,3})'


s1 = 'email: abc_1234@hmu.gr  '
print(re.findall(r, s1))
[('abc_1234', 'hmu.gr')]
```

Μας επιστρέφει λίστα από πλειάδες δυο στοιχείων της μορφής (user_name, domain_name)

```
s2 = 'First email: abc_1234@hmu.gr second email: def.5678@gmail.com  '
print(re.findall(r, s2))
[('abc_1234', 'hmu.gr'), ('def.5678', 'gmail.com')]
```

Μας επιστρέφει λίστα από πλειάδες δυο στοιχείων της μορφής (user_name, domain_name)

Παρενθέσεις που δεν επιστρέφουν κάτι (non capturing groups)

Κάποιες φορές χρειάζεται να χρησιμοποιήσουμε παρενθέσεις για να *ομαδοποιήσουμε ένα σύνολο χαρακτήρων* σε μια Κ.Ε., όμως δεν θέλουμε οι παρενθέσεις αυτές να δημιουργήσουν ένα *capturing group*

Στην περίπτωση αυτή *προσθέτουμε ένα ερωτηματικό και άνω-κάτω τελεία* αμέσως μετα την αριστερή παρένθεση ενώ η δεξιά παρένθεση παραμένει ως έχει `'(?: έκφραση)'`

```
r = 'email: ([a-z0-9._]+)@(?:[a-z0-9]+.[a-z0-9]{2,3})'
                 |--- capturing group --|      |---------------- non- capturing group ----------------|

s1 = 'email: abc_1234@hmu.gr  '
print(re.findall(r, s1))
['abc_1234']

s2 = 'First email: abc_1234@hmu.gr second email: def.5678@gmail.com  '
print(re.findall(r, s2))
['abc_1234', 'def.5678']
```

Χαρακτήρες διαφυγής στις κανονικές εκφράσεις της Python

(Escape characters in regular expressions)

Έχουν τη γενική μορφή `\x` όπου x συγκεκριμένοι χαρακτήρες που αποκτούν ειδική έννοια όταν ακολουθούν το backslash (\ )

Για παράδειγμα

- το `\w` μέσα σε μια Κ.Ε. ταιριάζει οποιοδήποτε αγγλικό αλφαριθμητικό χαρακτήρα κεφαλαίο ή πεζό, η την κάτω παύλα.
- το `\d` μέσα σε μια Κ.Ε. ταιριάζει οποιοδήποτε ψηφίο από 0 έως 9

Χρησιμοποιούνται για να περιγράψουμε με συνοπτικό τρόπο συνηθισμένες περιπτώσεις ταιριάσματος που συναντώνται αρκετά συχνά

έτσι πχ το `\w` είναι ισοδύναμο με την κλάση χαρακτήρων `[a-zA-Z0-9_]`

Σημαντικό: Στην Python, το backslash έχει σημασία *και στις απλές* συμβολοσειρές (πχ /n).
Γι’ αυτό οι Κ. Ε. σχεδόν πάντα πρέπει να γράφονται ως raw strings (`r'...'`)

```
r'\d+'  # σωστό
'\d+'   # θα χρειαζόταν διπλό backslash: '\\d+'
```

Το `r'\n'` είναι δύο χαρακτήρες, όχι newline.
Το `'\n'` είναι newline.

Χαρακτήρες διαφυγής στις κανονικές εκφράσεις της Python

\w : (πεζό w) Ενας οποιοσδήποτε χαρακτήρας που είναι αγγλικό γράμμα (κεφαλαίο ή πεζό) , ψηφίο ή κάτω παύλα *(ισοδύναμο με την κλάση χαρακτήρων* [a-zA-Z0-9_]*)*

\W : (Κεφαλαίο W) Ενας οποιοσδήποτε χαρακτήρας που *δεν ανήκει* στο προηγούμενο σύνολο \w

\s : (πεζό s) Ενας οποιοσδήποτε χαρακτήρας *τύπου whitespace* όπως το *κενό, tab, return, newline*

\S : (Κεφαλαίο S) Ενας οποιοσδήποτε χαρακτήρας που *δεν ανήκει* στο προηγούμενο σύνολο \s

\d : (πεζό d) Ενα οποιοδήποτε ψηφίο από 0…9 *(ισοδύναμο με την κλάση* [0-9]*)*

\D : (Κεφαλαίο D) Ενας οποιοσδήποτε χαρακτήρας που *δεν ανήκει* στο προηγούμενο σύνολο \d

\t : Ταιριάζει *(εντοπίζει)* τον χαρακτήρα *tab*

\b : Ταιριάζει *(εντοπίζει)* τον χαρακτήρα *backspace*

\n : Ταιριάζει *(εντοπίζει)* τον χαρακτήρα *newline*

\r : Ταιριάζει *(εντοπίζει)* τον χαρακτήρα *return*

\A : Ταιριάζει *(εντοπίζει)* ένα μοτίβο στην αρχή μιας συμβολοσειράς (ισοδύναμο με ^) (*)

\Z : Ταιριάζει *(εντοπίζει)* ένα μοτίβο στο τέλος μιας συμβολοσειράς (ισοδύναμο με $) (*)

(*) μόνο στην περίπτωση που η συμβολοσειρά δεν εκτείνεται σε περισσότερες από γραμμές (multiline).
Αυτό καθορίζεται από το flag re.M που θα δούμε αργότερα

Διαφορετικές χρήσεις του Backslash

\ : (backslash) Ο πιο ποικιλόμορφος χαρακτήρας στις κανονικές εκφράσεις

- Όταν σε μια κανονική έκφραση προηγείται ενός χαρακτήρα που έχει οριστεί σαν *χαρακτήρας διαφυγής (βλ. προηγ. διαφάνεια)* τότε ο συνδυασμός τους θεωρείται σαν *ειδική εντολή* και *ερμηνεύεται αντίστοιχα.* (πχ `\n`, `\w`, `\d` κ.λπ.)

- Όταν σε μια Κ.Ε. θέλουμε να χρησιμοποιήσουμε έναν *ειδικό χαρακτήρα (πχ.* [$ … *)* ως απλό χαρακτήρα για ταίριασμα, τότε βάζουμε μπροστά του backslash \

  πχ η Κ.Ε. `r'\('` ταιριάζει *(εντοπίζει)* όλες τις *αριστερές παρενθέσεις* σε ένα κείμενο

  <u>*Ειδική περίπτωση*</u>: μέσα σε μια *κλάση* [ ] όλοι οι ειδικοί χαρακτήρες *θεωρούνται απλοί* οπότε η χρήση του \ είναι *προαιρετική*. <u>*Εξαίρεση*</u> αποτελούν και επομένως χρειάζονται \ οι ειδικοί χαρακτήρες [και] *(για προφανείς λόγους)* καθώς και ο χαρακτήρας ^ αν είναι πρώτος μέσα σε μια κλάση *(σημαίνει <u>όχι</u> οι επόμ.χαρακτήρες)*

- Όταν σε μια Κ.Ε. θέλουμε να χρησιμοποιήσουμε έναν *το ίδιο το backslash* ως απλό χαρακτήρα για ταίριασμα, τότε γράφουμε διπλό backslash \\

  χρειάζεται προσοχή καθώς αυτό <u>δεν υλοποιείται με τον ίδιο τρόπο</u> σε όλες τις πλατφόρμες και γλώσσες που υποστηρίζουν Κ.Ε. (Python, Java, smart editors…)

Το δομοστοιχείο re

Περιλαμβάνει συναρτήσεις /μεθόδους για την δημιουργία και χρήση *κανονικών εκφράσεων* (*regular expressions*) στην Python, σύμφωνα με τους κανόνες που έχουμε δει μέχρι τώρα.

Οι σημαντικότερες συναρτήσεις / μέθοδοι που περιέχονται στο δομοστοιχείο re είναι οι ακόλουθες:

`match()`	Ελέγχει αν *η αρχή μιας συμβολοσειράς* ταιριάζει με μια δοσμένη Κ.Ε. Τα αποτελέσματα επιστρέφονται σε ένα αντικείμενο τύπου *matchobj*
`search()`	Διατρέχει μια συμβολοσειρά, και εντοπίζει *το πρώτο σημείο της* (substring) που ταιριάζει με μια δοσμένη Κ.Ε. Τα αποτελέσματα επιστρέφονται σε ένα αντικείμενο τύπου *matchobj*
`findall()`	Διατρέχει μια συμβολοσειρά, και εντοπίζει *όλα τα σημεία της* (substrings) που ταιριάζουν με μια δοσμένη Κ.Ε Τα αποτελέσματα επιστρέφονται *σε μορφή λίστας*
`finditer()`	Όμοια με την `findall()`, όμως τα αποτελέσματα επιστρέφονται *σε μορφή επαναλήπτη* (iterator)
`sub()`	*Εύρεση και αντικατάσταση* Διατρέχει μια συμβολοσειρά, και αφού εντοπίσει *όλα τα σημεία της* (substrings) που ταιριάζουν με μια δοσμένη Κ.Ε., τα αντικαθιστά με μια συμβολοσειρά της επιλογής του χρήστη (κείμενο αντικατάστασης) και *επιστρέφει την τροποποιημένη αρχική συμβολοσειρά*
`subn()`	Όμοια με την `sub()`, όμως μαζί με την τροποποιημένη αρχική συμβολοσειρά επιστρέφεται και *ο αριθμός των αντικαταστάσεων* που πραγματοποιήθηκαν (πλειάδα δυο στοιχείων: συμβολοσειρά, αρ. αντικαταστάσεων)
`split()`	*Διαχωρίζει* (σπάει) μια συμβολοσειρά στα σημεία που ταιριάζουν με μια δοσμένη Κ.Ε. και *επιστρέφει μια λίστα* με τα κομμάτια που προέκυψαν από το σπάσιμο
`compile()`	*Μετατρέπει* μια κανονική έκφραση από *συμβολοσειρά* σε ένα αντικείμενο τύπου *regular expression pattern*

re.match()

Ελέγχει αν *η αρχή μιας συμβολοσειράς* ταιριάζει με μια δοσμένη Κ.Ε.

```
re.match(pattern, string, [flags])
```

`pattern` : Κανονική Έκφραση

`string` : Συμβολοσειρά που εφαρμόζεται η Κ.Ε.

`flags` : Προαιρετικές παράμετροι ρυθμίσεων

Επιστρέφει `None` αν δεν υπάρχει ταίριασμα. Διαφορετικά επιστρέφει ένα αντικείμενο τύπου `matchobj` το οποίο περιέχει όλη την πληροφορία για την έκβαση του ταιριάσματος αλλά και για τις παραμέτρους εισόδου που οδήγησαν σε αυτήν (pattern, string…) *(λεπτομέρειες αμέσως μετά)*

Το ίδιο το `matchobj` έχει πάντα την τιμή `True` οπότε είναι εύκολο να ελέγξουμε με ένα `if` αν υπήρξε ή όχι ταίριασμα

```
m_obj = re.match(pattern, string)
if m_obj:
    ...     # η αρχή της συμβολοσειράς ταιριάζει με την Κ.Ε., οπότε συνέχισε να κάνεις ό,τι χρειάζεται
else:
    print('No match') # το m_obj έχει τιμή None
```

Βασικές μέθοδοι / ιδιότητες του αντικειμένου matchobj

Έστω `m_obj` είναι ένα αντικείμενο τύπου `matchobj` που επιστρέφεται από την `match()`

- `m_obj.re` : Επιστρέφει την Κ.Ε. που χρησιμοποιήθηκε για το ταίριασμα
- `m_obj.string` : Επιστρέφει την συμβολοσειρά πάνω στην οποία εφαρμόστηκε η Κ.Ε
- `match.group()` : Επιστρέφει το *αποτέλεσμα* του ταιριάσματος
- `match.start()` : Επιστρέφει τον δείκτη της θέσης που *αρχίζει* το ταίριασμα
- `match.end()` : Επιστρέφει τον δείκτη της *επόμενης* θέσης που *τελειώνει* το ταίριασμα
- `match.span()` : Επιστρέφει μια *πλειάδα* με τους δυο παραπάνω δείκτες (αρχή, τέλος)

Οι πιο βασικές. (υπάρχουν πολύ περισσότερες)

```
r= r'[ac]+'  # Η Κ.Ε. ως raw string
s = 'academy'
m_obj = re.search(r,s)
if m_obj :
    print(m_obj)
    print(m_obj.re, m_obj.string)
    print(m_obj.group(), m_obj.span())
    print(m_obj.start(), m_obj.end())
else: print ('--- No match ---')
```

Αποτελέσματα

```
<re.Match object; span=(0, 3), match='aca'>
re.compile('[ac]+') academy
aca (0, 3)
0 3
```

re.search()

Διατρέχει μια συμβολοσειρά, και εντοπίζει *το πρώτο σημείο της* που ταιριάζει με μια δοσμένη Κ.Ε.

```
re.search(pattern, string, [flags])
```

`pattern` : Κανονική Έκφραση

`string` : Συμβολοσειρά που εφαρμόζεται η Κ.Ε.

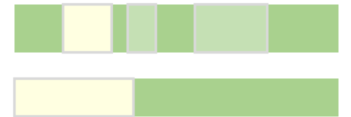
`flags` : Προαιρετικές παράμετροι ρυθμίσεων

Επιστρέφει `None` αν δεν υπάρχει ταίριασμα. Διαφορετικά επιστρέφει ένα αντικείμενο τύπου *`matchobj`*

Βασική διαφορά των μεθόδων search και match

`search` : το σημείο ταιριάσματος της Κ.Ε. μπορεί να είναι *οπουδήποτε μέσα στην συμβολοσειρά*

`match` : το σημείο ταιριάσματος της Κ.Ε. μπορεί να είναι *μόνο στην αρχή της συμβολοσειράς*



```
r = r'test'
s = 'Contestant'

if re.search(r,s):
    print("Found", r, "in", s )
else:
    print("Not found", r, "in", s )

Found test in Contestant
```

```
r = r'test'
s = 'Contestant'

if re.match(r,s):
    print("Found", r, "in", s )
else:
    print("Not found", r, "in", s )

Not found test in Contestant
```

re.findall()

Διατρέχει μια συμβολοσειρά, και εντοπίζει *όλα τα σημεία της* που ταιριάζουν με μια δοσμένη Κ.Ε.

```
re.findall(pattern, string, [flags])
```

- `pattern` : Κανονική Έκφραση
- `string` : Συμβολοσειρά που εφαρμόζεται η Κ.Ε.
- `flags` : Προαιρετικές παράμετροι ρυθμίσεων

Επιστρέφει *μια λίστα* με όλα τα ταιριάσματα

```
r = r'[0-9]+'  # ή ισοδύναμα r = r'/d+'
s = '25η Μαρτίου 1821'
print(re.findall(r,s))

['25', '1821']
```

Αν η Κ.Ε. περιλαμβάνει *περισσότερα από ένα capturing groups* (ζεύγη παρενθέσεων), τότε κάθε στοιχείο της λίστας είναι *μια πλειάδα* με τμηματικά ταιριάσματα (βλ. παράδειγμα στη σελ. 18)

re.finditer()

Διατρέχει μια συμβολοσειρά, και εντοπίζει *όλα τα σημεία της* που ταιριάζουν με μια δοσμένη Κ.Ε.

```
re.finditer(pattern, string, [flags])
```

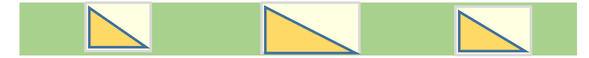
- `pattern` : Κανονική Έκφραση
- `string` : Συμβολοσειρά που εφαρμόζεται η Κ.Ε.
- `flags` : Προαιρετικές παράμετροι ρυθμίσεων

Επιστρέφει *έναν επαναλήπτη* που παράγει ένα-ένα τα ταιριάσματα σε μορφή `matchobj`

```
r= r'[0-9]+'
s = 'Την 25η Μαρτίου 1821'
iter = re.finditer(r,s)
print(iter)
for i in iter:
    print(i.group(), i.span())
```

```
<callable_iterator object at 0x03370A18>
25 (4, 6)
1821 (16, 20)
```

re.sub() / re.subn()



re.sub(): διατρέχει μια συμβολοσειρά, και αφού εντοπίσει *όλα τα σημεία της* (substrings) που ταιριάζουν με μια δοσμένη Κ.Ε., τα αντικαθιστά με μια συμβολοσειρά της επιλογής του χρήστη και *επιστρέφει την τροποποιημένη αρχική συμβολοσειρά.*

```
re.sub(pattern, repl, string, [n], [flags])
```

- pattern : Κανονική Έκφραση
- repl : Κείμενο αντικατάστασης
- string : Συμβολοσειρά πάνω στην οποία εφαρμόζεται η Κ.Ε.
- n : Προαιρετικό πλήθος αντικαταστάσεων *(αν απουσιάζει, γίνονται όλες οι αντικαταστάσεις)*
- flags : Προαιρετικές παράμετροι ρυθμίσεων

```
r = r'\([^)]*\)'
```

re.subn(): *επιστρέφει επιπλέον* και τον αριθμό των αντικαταστάσεων που έγιναν (πλειάδα δυο στοιχείων)

Παράδειγμα: Εύρεση και αντικατάσταση παρενθέσεων μέσα σε κείμενο με χρήση κανονικών εκφράσεων

```
r = r'\([^)]*\)'  #εντοπισμός κειμένου σε παρένθεση
s = 'Αθήνα (Ελλάδα) Ρώμη (Ιταλία).'
print(re.sub(r, 'XXX', s))
print(re.subn(r, 'XXX', s))
Αθήνα XXX Ρώμη XXX.  επιστρέφει συμβολοσειρά
('Αθήνα XXX Ρώμη XXX.', 2)  επιστρέφει πλειάδα
```

```
r = r'\([^)]*\)'  #εντοπισμός κειμένου σε παρένθεση
s = 'Αθήνα (Ελλάδα) Ρώμη (Ιταλία).'
print(re.sub(r, 'XXX', s, 1))
Αθήνα XXX Ρώμη (Ιταλία).
```

μια μόνο φορά

re.split()

Διαχωρίζει (σπάει) μια συμβολοσειρά στα σημεία που ταιριάζουν με μια δοσμένη Κ.Ε. και *επιστρέφει μια* ***λίστα*** με τα κομμάτια που προέκυψαν από το σπάσιμο

```
re.split(pattern, string, [flags])
    pattern  : Κανονική Έκφραση
    string   : Συμβολοσειρά που εφαρμόζεται η Κ.Ε.
    flags    : Προαιρετικές παράμετροι ρυθμίσεων
```

Παραδείγματα

α. Με διαχωριστικά στοιχεία τα whitespace , '(' και ')'

```
r = r'[\s()]+'  #εντοπισμός whitespace , '(' και ')'
s = 'Αθήνα (Ελλάδα) Ρώμη (Ιταλία).'
a = re.split(r, s)
print(a)

['Αθήνα', 'Ελλάδα', 'Ρώμη', 'Ιταλία', '.']
```

```
r = r'[\s()]+'
```

β. Με διαχωριστικά στοιχεία τα κείμενα σε παρένθεση

```
r = r '\([^)]*\)'  #εντοπισμός κειμένου σε παρένθεση
s = 'Αθήνα (Ελλάδα) Ρώμη (Ιταλία).'
a = re.split(r, s)
print(a)

['Αθήνα ', ' Ρώμη ', '.']
```

```
r = '\([^)]*\)'
```

Ειδική περίπτωση: η συνάρτηση re.compile()

Μετατρέπει (μεταγλωττίζει – *compiles*) μια κανονική έκφραση από *συμβολοσειρά* σε ένα αντικείμενο τύπου *regular expression pattern <class 're.Pattern'>*. Ένα αντικείμενο αυτής της μορφής έχει ως μεθόδους όλες τις συναρτήσεις που είδαμε μέχρι τώρα, (`match`, `search`, `sub`, `split`…)

<u>*Παράδειγμα*</u>:

Η έκφραση

```
result = re.match(pattern, string, [flags])
```

είναι ισοδύναμη με

```
regex = re.compile(pattern)
result = regex.match(string, [flags])
```

```
nums_str = r'[0-9]+'
s = 'Την 25η Μαρτίου 1821'
nums_re = re.compile(nums_str)
print(nums_re.findall(s))
print(nums_re.sub('%%', s))

['25', '1821']
Την %%η Μαρτίου %%
```

Βασικός λόγος για την μετατροπή μιας Κ.Ε. από συμβολοσειρά σε αντικείμενο είναι η ταχύτητα εφαρμογής της

Αποτελεί καλή πρακτική όταν έχουμε να εφαρμόσουμε μια Κ.Ε σε ένα μεγάλο πλήθος συμβολοσειρών

Παράδειγμα χρήσης του re.compile()

Δημιουργία έτοιμων (μεταγλωττισμένων) Κ.Ε. για τον έλεγχο email διαφόρων τμημάτων του ΕΛΜΕΠΑ

```
re_HMMY = re.compile(r'th2[/d]{4}@edu.hmu.gr') #Έγκυρο email φοιτητή Τμήματος Ηλεκτρολόγων ΕΛΜΕΠΑ
re_MECH = re.compile(r'tm2[/d]{4}@edu.hmu.gr') #Έγκυρο email φοιτητή Τμήματος Μηχανολόγων ΕΛΜΕΠΑ
re_MSC = re.compile(r'mtp[/d]{3}@edu.hmu.gr') #Έγκυρο email  μεταπτυχ. φοιτητή Σχολής Μηχανικών ΕΛΜΕΠΑ
```

Κώδικας για την ταξινόμηση μιας γενικής λίστας με email φοιτητών του ΕΛΜΕΠΑ σε διαφορετικές λίστες ανάλογα με το Τμήμα στο οποίο ανήκουν *(τα άγνωστα email τοποθετούνται στην ξεχωριστή λίστα* UNKNOWN_List*)*

```
for x in generic_list_of_emails:
      if re_HMMY.match(x): HMMY_List.append(x)
      elif re_MECH.match(x): MECH_List.append(x)
      elif re_MSC.match(x): MSC_List.append(x)
      else UNKNOWN_List.append(x)
```

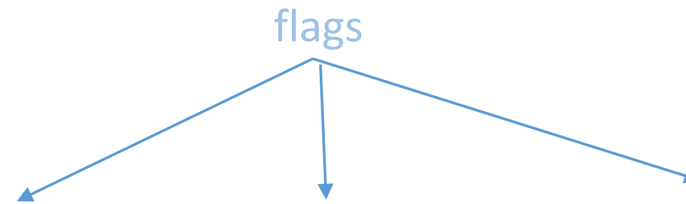
Αν θέλαμε οι λίστες μας να περιέχουν μόνο τους ΑΜ των φοιτητών ,οι Κ.Ε. θα έπρεπε να γραφούν ως εξής

```
re_HMMY_only_AM = re.compile('th(2[/d]{4})@edu.hmu.gr')
```

Τα flags στις κανονικές εκφράσεις

Οι περισσότερες μέθοδοι Κ.Ε. δέχονται στο τέλος ένα προαιρετικό όρισμα που ονομάζεται "flags".
Τα flags χρησιμοποιούνται για να τροποποιήσουν κάποιες από τις παραμέτρους ταιριάσματος

<u>*Παράδειγμα*</u>:




```
regex = r'...'
str = '...'
re.search(regex ,str, re.UNICODE | re.MULTILINE | re.IGNORECASE)
```

Μπορούμε να συνδυάζουμε πολλά flags, με το bitwise OR (|)

θα προσπαθήσει να ταιριάσει την Κ.Ε. `regex` στην συμβολοσειρά `str` θεωρώντας τις παρακάτω παραδοχές:

- η συμβολοσειρά `str` είναι κωδικοποιημένη σε Unicode (`re.UNICODE`)
- η συμβολοσειρά `str` εκτείνεται σε περισσότερες από μια γραμμές (`re.MULTILINE`)
- το ταίριασμα θα γίνει χωρίς διάκριση πεζών-κεφαλαίων (`re.IGNORECASE`)

Κυριότερα flags της re στην Python

`re.I` ή `re.IGNORECASE`	Ταίριασμα χωρίς διάκριση πεζών/κεφαλαίων
`re.M` ή `re.MULTILINE`	Επηρεάζει τα ^ και $ • το ^ ταιριάζει την αρχή κάθε γραμμής, όχι μόνο αρχή κειμένου • το $ ταιριάζει το τέλος κάθε γραμμής Χρήσιμο σε κείμενα με πολλές γραμμές
`re.S` ή `re.DOTALL`	Η τελεία (.) ταιριάζει και τα newlines Κανονικά η τελεία δεν ταιριάζει το \n. Με αυτό το flag ταιριάζει *τα πάντα.*
`re.A` ή `re.ASCII`	Λαμβάνει υπόψη μόνο ASCII για \w, \d, \s Π.χ. με αυτό *τα ελληνικά ΔΕΝ θεωρούνται word characters.*
`re.U` ή `re.UNICODE`	Τα \w και \W ερμηνεύονται σύμφωνα με την κωδικοποίηση Unicode
`re.L` ή `re.LOCALE`	Κάνει το \w, \b κ.λπ. να εξαρτώνται από το locale του συστήματος. Πλέον θεωρείται ξεπερασμένο
`re.X` ή `re.VERBOSE`	"Όμορφες" κανονικές εκφράσεις με σχόλια και κενά Επιτρέπει να γράφουμε Κ.Ε. σε πολλές γραμμές με σχόλια

Μπορούμε να συνδυάζουμε πολλά flags, με το bitwise OR (|)

Παραδείγματα στα flags

```
re.findall(r'cat', "Cat CAT cAt", re.I)  ⇨ ['Cat', 'CAT', 'cAt']
```

```
text = '''Hello
world'''
re.findall (r'H.*', text, re.S) ⇨ ['Hello\nworld']
```

Χωρίς το re.S, η τελεία θα σταματούσε στο newline, και θα μας έδινε `['Hello']`

```
text = """apple
banana
cherry"""
re.findall(r'^b\w+', text, re.M) ⇨ ['banana']
```

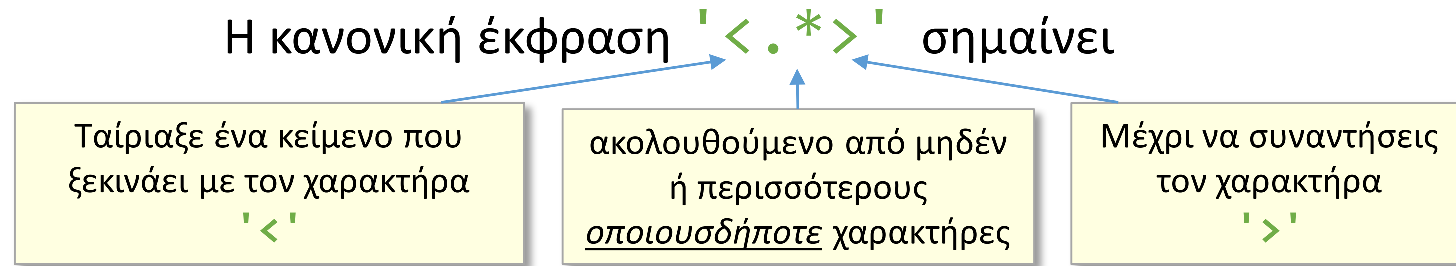
Χωρίς re.M, το ^b δεν θα ταίριαζε τίποτα, γιατί το ^ αναφέρεται μόνο στην αρχή ΟΛΟΥ του κειμένου

```
pattern = r'''
    \d+       # αριθμός
    \s*       # κενά
    [A-Z]+    # κεφαλαία γράμματα
'''

re.findall(pattern, text, re.X)
```

Ταίριασμα greedy / non greedy

greedy: *άπληστος, αχόρταγος, αυτός που τα θέλει όλα δικά του*



Τι θα πάρουμε άραγε αν εφαρμόσουμε την παραπάνω Κ.Ε. στη συμβολοσειρά

`'bla bla bla <H1>title</H1> more bla bla'`

Θα πάρουμε `'<H1>'` ή μήπως `'<H1>title</H1>'`;

Με άλλα λόγια θα σταματήσει στο *πρώτο* `'>'` που θα συναντήσει (*ολιγαρκής* / *non-greedy*) ,
ή θα συνεχίσει να ψάχνει και θα σταματήσει στο *τελευταίο δυνατό* `'>'` (*άπληστος* / *greedy*) ;

Η Python *εξ ορισμού* υποστηρίζει το *ταίριασμα greedy* οπότε θα πάρουμε `'<H1>title</H1>'`

Αν θελήσουμε *ταίριασμα non-greedy* (ώστε να πάρουμε μόνο το `'<H1>'`) πρέπει να το *δηλώσουμε*

Ταίριασμα greedy / non greedy

Σε μια Κ.Ε, οι τελεστές επανάληψης { `{a,b}` γενική μορφή, `*` ισοδύναμο του `{0,}`, `+` ισοδύναμο του `{1,}` } είναι εξ' ορισμού greedy

Αυτό σημαίνει ότι θα προσπαθήσουν να ταιριάξουν *όσους περισσότερους χαρακτήρες μπορούν*

Κάποιες φορές όμως είναι επιθυμητό το αντίθετο (δηλ. *να σταματήσουν μόλις επιτύχουν ένα ελάχιστο ταίριασμα*). Αυτό το ταίριασμα ονομάζεται *non-greedy* και δηλώνεται προσθέτοντας τον χαρακτήρα ? αμέσως μετά τον τελεστή επανάληψης. Επομένως ισχύει :

`{a,b}`	*Greedy matching.* Ο τελεστής επανάληψης θα προσπαθήσει να ταιριάσει *όσο περισσότερους χαρακτήρες μπορέσει*	`{a,b}?`	*Non greedy matching*. Ο τελεστής επανάληψης θα *σταματήσει να ψάχνει* μόλις επιτύχει το πρώτο ταίριασμα
`*`		`*?`	
`+`		`+?`	

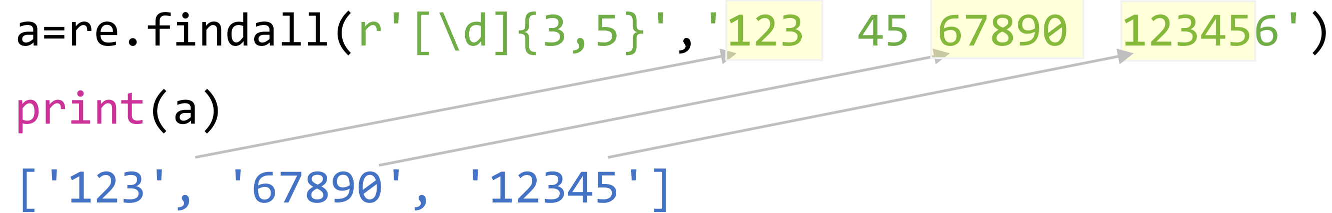
<u>Διευκρίνιση:</u> Το ? σηματοδοτεί ταίριασμα non-greedy *μόνο όταν ακολουθεί* έναν από τους 3 παραπάνω τελεστές επανάληψης. Όταν εμφανίζεται *μόνο του* σε μια Κ.Ε. ερμηνεύεται και αυτό ως ειδικός χαρακτήρας επανάληψης *είναι όπως έχουμε δει ισοδύναμο του* `{0,1}`

Ταίριασμα greedy / non greedy – Παράδειγμα 1

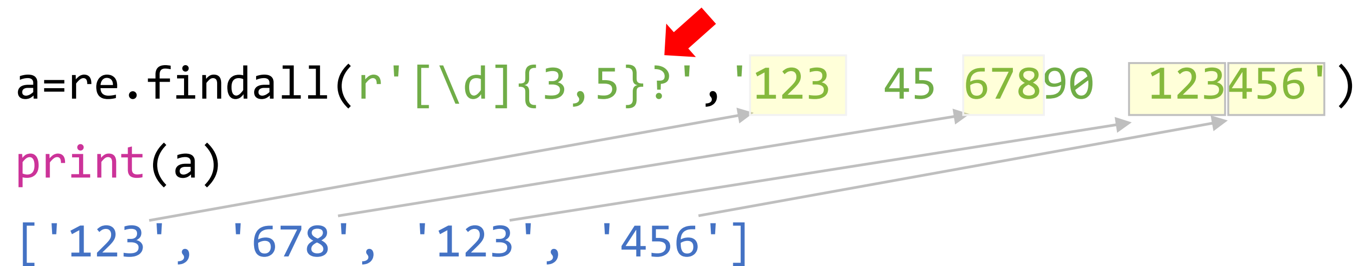
```
# Greedy matching. Returns the whole string as a single match
a = re.findall( r'<.*>',  '<H1> title </H1> <H1> subtitle </H1>')
print(a)
['<H1> title </H1> <H1> subtitle </H1>']
```
η λίστα περιέχει ένα στοιχείο

```
# Non-greedy matching. Returns four different matches
a = re.findall( r'<.*?>',  '<H1> title </H1> <H1> subtitle </H1>')
print(a)
['<H1>', '</H1>', '<H1>', '</H1>']
```
η λίστα περιέχει 4 στοιχεία

Ταίριασμα greedy / non greedy – Περισσότερα παραδείγματα

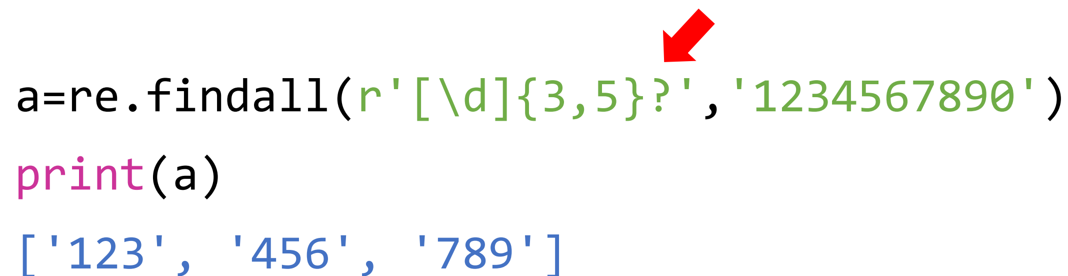


```
a=re.findall(r'[\d]{3,5}','123  45 67890  123456')
print(a)
['123', '67890', '12345']
```



```
a=re.findall(r'[\d]{3,5}?','123  45 67890  123456')
print(a)
['123', '678', '123', '456']
```

```
a=re.findall(r'[\d]{3,5}','1234567890')
print(a)
['12345', '67890']
```



```
a=re.findall(r'[\d]{3,5}?','1234567890')
print(a)
['123', '456', '789']
```

Τέλος Διάλεξης

Ερωτήσεις;

Τμήματα αυτής της διάλεξης περιέχουν στοιχεία από πηγές που είναι ελεύθερες στο διαδίκτυο όπως η Βικιπαιδεια και ανοιχτές σημειώσεις παρεμφερών διαλέξεων