Generators

Οι generators (γεννήτριες) είναι δομές παρόμοιες με τις λίστες με τη διαφορά, ότι στη μνήμη του υπολογιστή δεν κρατάει όλες τις τιμές της λίστας, αλλά τον κώδικα που χρειάζεται για την αναπαραγωγή τους.

Παρόλο που δεν το έχουμε τονίσει, οι συναρτήσεις `range`, `enumerate`, `items`, `map` που έχουμε παρουσιάσει επιτρέφουν generator.

Για να καταλάβουμε καλύτερα τους generators ας χρησιμοποιήσουμε ένα ανθρώπινο παράδειγμα. Ας υποθέσουμε οτι κάποιος σας λέει: απομνημόνευσε τους παρακάτω 100 αριθμούς τηλεφώνου με τη σειρά που στους δίνω. Αφού το κάνεις αυτό, πες τον πρώτο αριθμό, μετά τον δεύτερο, κτλ..

Αυτό θα απαιτούσε τεράστια προσπάθεια (και μνήμη) από εσάς.

Ας υποθέσουμε τώρα ότι κάποιος σας λέει: απομνημόνευσε όλους τους ζυγούς αριθμούς από το 2 μέχρι το 1000. Αφού το κάνεις αυτό πες μου τον πρώτο αριθμό, μετά τον δεύτερο κτλ..

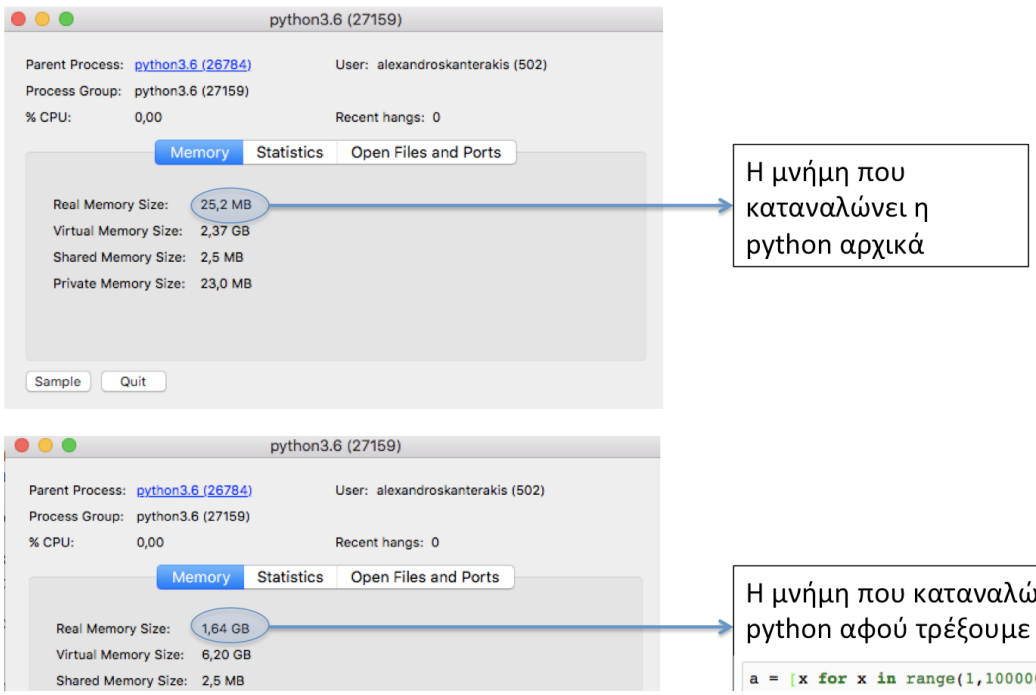
Δεν χρειάζεται ιδιαίτερος κόπος για αυτό. Απλά πρέπει να θυμάσαι που πρέπει να αρχίσεις (το 2), να τελειώσεις (το 1000), σε ποιο σημείο είσαι τώρα και με ποιο τρόπο βρίσκεις το επόμενο. Αυτό ακριβώς κάνουν οι generators. Είναι χρήσιμες σε περιπτώσεις που το επόμενο στοιχείο μιας λίστας μπορεί να υπολογιστεί και όχι να ανακληθεί από τη μνήμη.

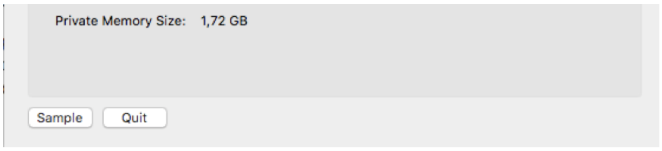
Έτσι όταν λέμε:

```
range(1,1000000)
```

Δεν υπάρχει λόγος να αποθηκεύσουμε 1.000.000 τιμές στη μνήμη του υπολογιστή. Ας το δούμε στη πράξη. Αν τρέξουμε:

```
a = [x for x in range(1,100_000_000)]
```





Η εντολή αυτή απαιτεί 1.5GB μνήμης!!

βλέπουμε ότι η python χρειάζεται 1.5GB μνήμης. Αν χρειάζεται να παίρνουμε αυθαίρετα τιμές από τη λίστα `a` τότε δεν μπορούμε να κάνουμε αλλιώς. Διαφορετικά αν χρειάζεται απλά να παίρνουμε τη μία τιμή μετά την άλλη τότε μπορούμε να γλυτώσουμε αυτή τη μνήμη γράφοντας:

```
for x in range(1, 1000000):
    ...
```

Μπορούμε να φτιάξουμε τον δικό μας generator με δύο τρόπους: Ο πρώτος είναι μέσω generator comprehension και ο δεύτερος είναι μέσω συναρτήσεων που αντί να κάνουν `return` κάνουν `yield` .

Τα generator comprehension είναι ακριβώς όπως και τα list comprehensions (και τα dictionary, set comprehension) απλά χρησημοποιούμε τις παρενθέσεις:

In [1]:
```
my_generator = (x for x in range(1,10))
type(my_generator)
```

Out[1]:
```
generator
```

Ας πάρουμε τη πρώτη τιμή του generator:

In [2]:
```
next(my_generator)
```

Out[2]:
```
1
```

Ας πάρουμε την επόμενη:

In [3]:
```
next(my_generator)
```

Out[3]:
```
2
```

... κτλ.. Μπορούμε να πάρουμε τις υπόλοιπες τιμές του generator:

In [4]:
```
for x in my_generator:
    print (x)
```

```
3
4
5
6
7
8
9
```

Βλέπουμε ότι ο generator "κρατάει" σε ποια τιμή ήμασταν και υπολογίζει την επόμενη. Όταν τελειώσουν οι τιμές του generator τότε δεν μπορούμε να πάρουμε άλλη:

In [5]:
```
next(my_generator)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-5-beb7403f481d> in <module>
----> 1 next(my_generator)

StopIteration:
```

Ο δεύτερος τρόπος να φτιάξουμε generator είναι να δηλώσουμε μία συνάρτηση με τη `def` και αντί για `return` να κάνουμε `yield` :

In [6]:
```
def f():
    yield "first"
    yield "second"
    yield "third"
```

Προσοχή η `f` είναι μία συνάρτηση που φτιάχνει έναν generator! Η `f` δεν είναι generator!

In [7]:
```
type(f)
```

Out[7]:
```
function
```

In [8]:
```
my_generator = f()
type(my_generator)
```

Out[8]:
```
generator
```

Το my_generator είναι generator. Ας τον "τρέξουμε":

In [9]:
```
next(my_generator)
```

Out[9]:
```
'first'
```

In [10]:
```
next(my_generator)
```

Out[10]:
```
'second'
```

In [11]:
```
next(my_generator)
```

Out[11]:
```
'third'
```

Ή αλλιώς:

In [12]:
```
for x in f():
    print (x)
```

```
first
second
third
```

Ένα άλλο παράδειγμα. Ένας generator που κάνει generate πρώτους αριθμούς:

In [13]:
```
def prime_gen():
    yield 1
    n=2
    while True:
        for d in range(2, n-1):
            if n%d==0:
                break
        else:
            yield n
        n+=1
```

In [14]:
```
g = prime_gen()
```

In [15]:
```
next(g)
```

```
Out[15]: 1

In [16]: next(g)

Out[16]: 2

In [17]: next(g)

Out[17]: 3

In [18]: next(g)

Out[18]: 5

In [19]: g = prime_gen()
         for i in range(10):
             print (next(g))

1
2
3
5
7
11
13
17
19
23
```

Όταν "τελειώνει" ένας generator πετάει ένα exception (θα τα δούμε αργότερα) το οποίο μπορούμε να πιάσουμε:

```
In [21]: gen = (x*2 for x in range(10))

         while True:
             try:
                 n = next(gen)
             except StopIteration:
                 break
             print (n)

0
2
4
6
8
10
12
14
16
18
```

import

Με την `import` μπορούμε να "βάλουμε" στο περιβάλλον που δουλεύουμε κώδικα από ένα άλλο αρχείο. Ας υποθέσουμε ότι έχουμε το αρχείο `a.py` το οποίο έχει τον παρακάτω κώδικα:

```
# File: a.py

def f():
    print ("hello")
```

```
def g():
    print ("world")
```

Αυτό μπορούμε να το κάνουμε "τρέχοντας" το παρακάτω κελί:

In [22]:
```
%%writefile a.py
def f():
    print ("hello")

def g():
    print ("world")

name="mitsos"
```

```
Writing a.py
```

Τότε μπορούμε να κάνουμε `import` τις συναρτήσεις αυτές σε ένα άλλο αρχείο:

In [23]:
```
from a import f # Κάνω import μόνο την f
f()
```

```
hello
```

In [24]:
```
from a import f,g # Κάνω import την f και τη g
f()
g()
```

```
hello
world
```

In [25]:
```
from a import * # Κάνω import όλες τις συναρτήσεις / μεταβλητές / ... που υ
f()
g()
```

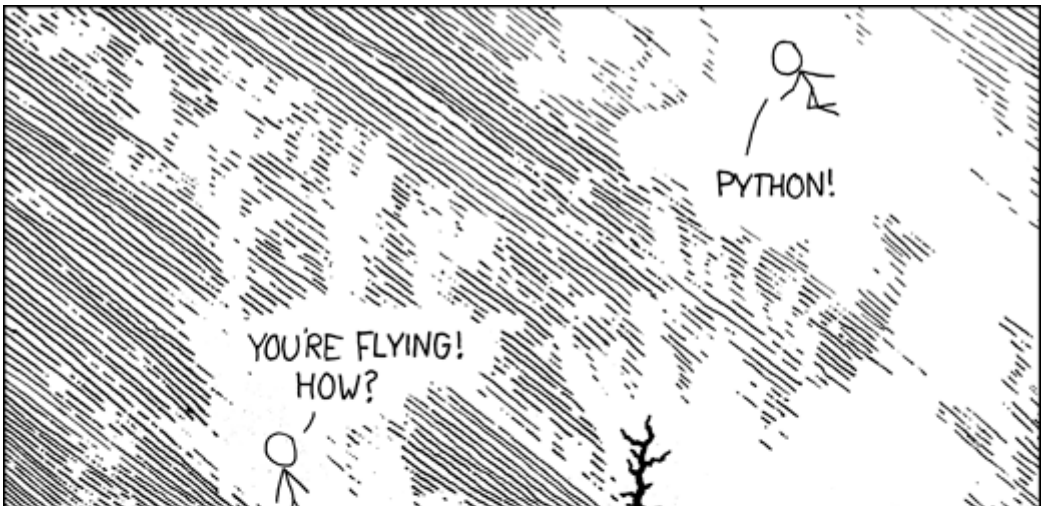
```
hello
world
```

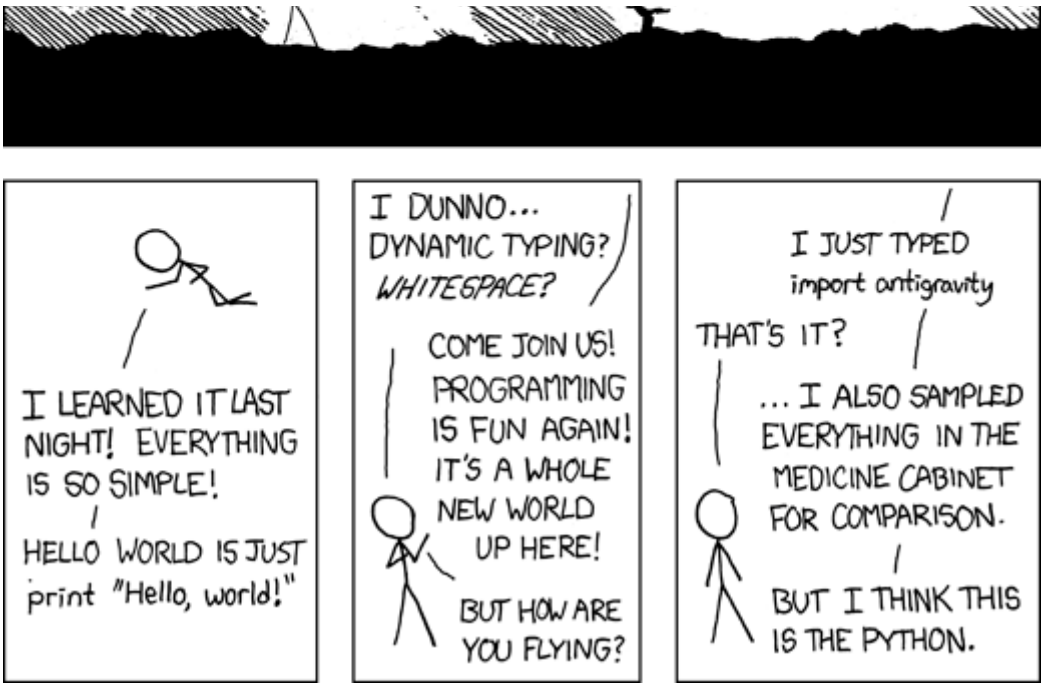
Μπορώ να κάνω `import` το `a` και να χρησιμοποιώ το `a.f()` όταν θέλω να καλέσω μία συνάρτηση:

In [26]:
```
import a
a.f()
a.g()
```

```
hello
world
```

Η python είναι μία γλώσσα "batteries included". Το οποίο σημαίνει ότι έχει πάρα πολλές βιβλιοθήκες (libraries) με χρήσιμες συναρτήσεις. Τις συναρτήσεις αυτές τις βάζουμε με την import





πηγή

Δοκιμάστε:

```
import antigravity
```

Η βιβλιοθήκη random

In [52]:
```
import random
```

In [28]:
```
random.random() # τυχαίοι αριθμοί από το 0 μέχρι το 1
```

Out[28]:
```
0.7875710628628508
```

In [29]:
```
random.randint(1,10) # τυχαίοι ακέραιοι από το 1 μέχρι το 10
```

Out[29]:
```
9
```

In [53]:
```
# https://www.youtube.com/watch?v=INa6b0yS3uk
random.choice(['Λονδίνο', 'Άμστερνταμ', 'Βερολίνο']) # Διαλέγει ένα στη τύχ
```

Out[53]:
```
'Βερολίνο'
```

In [31]:
```
random.sample(['Λονδίνο', 'Άμστερνταμ', 'Βερολίνο'],2) # Διαλέγει 2 στη τύχ
```

Out[31]:
```
['Ηράκλειο', 'Άμστερνταμ']
```

Εκτελώντας προγράμματα από τη γραμμή εντολών

Μέχρι στιγμής έχουμε μάθει πως να φτιάχνουμε απίστευτα πράγματα στο Jupyter. Το Jupyter έχει προκύψει τώρα τελευταία σαν μέσο συγγραφής κώδικα αλλά και σαν μέθοδο να διαμοιράζουμε τα προγράμματά μας. Το Jupyter **δεν** είναι ο κανόνας για τη δημιουργία προγραμμάτων με python αλλά η εξαίρεση. Τα περισσότερα προγράμματα γράφονται σε ένα αρχείο το οποίο έχει τη κατάληξη `.py` . Για παράδειγμα `program.py` . Αυτό το πρόγραμμα το τρέχουμε από τη γραμμή εντολών με την εντολή:

```
python program.py
```

Τι είναι όμως η γραμμή εντολών;

Κάθε υπολογιστής, ανεξάρτητα από το λειτουργικό του σύστημα (windows, osx, Linux) διαθέτει ένα πρόγραμμα που ονομάζεται γραμμή εντολών, ή command line. Όταν ανοίγετε αυτο το πρόγραμμα εμφανίζεται μία κονσόλα όπου ο χρήστης μπορεί να τυπώσει εντολές (commands) και πατώντας το κουμπί Enter, ο υπολογιστής τις εκτελεί (ή αλλιώς τις "τρέχει"). Μπορεί σήμερα να ακούγεται περίεργο αλλά αυτός είναι και ο κύριος τρόπος με τον οποίο τρέχουν τα περισσότερα προγράμματα στον υπολογιστής σας. Όταν κάνετε διπλό κλικ για να ανοίξετε ένα πρόγραμμα σε ένα γραφικό περιβάλλον, ο υπολογιστής εσωτερικά απλά εκτελεί την αντίστοιχη εντολή του προγράμματος.

Σκεφτείτε λίγο το εξής: Τα γραφικά περιβάλλοντα (π.χ. windows) υπάρχουν από τη δεκαετία του '90 και μετά. Πιο πριν πως δούλευε ο κόσμος στους υπολογιστές; Γράφοντας εντολές στη γραμμή εντολών.

Σκεφτείτε λίγο το εξής: Το Jupyter υπάρχει από το 2014 (και το IPython στο οποίο βασίζεται από το 2001). Το RStudio υπάρχει από το 2010. Πιο πριν πως δούλευε ο κόσμος σε αυτές τις δημοφιλείς γλώσσες; Μα με τη γραμμή εντολών.

Σκεφτείτε λίγο το εξής: Ένα πρόγραμμα που απαιτεί έναν υπερυπολογιστή για να τρέξει σε ένα απομακρυσμένο μηχάνημα και θέλει μήνες για να ολοκληρωθεί η επεξεργασία που κάνει, πως το στέλνουμε για εκτέλεση; Σίγουρα δεν μπορούμε να έχουμε ένα jupyter ανοικτό για μήνες! Η απάντηση είναι το εκτελούμε μέσα από τη γραμμή εντολών.

Σκεφτείτε λίγο το εξής: Ένα πρόγραμμα για να κάνει τους υπολογισμούς του απαιτεί από τον χρήστη να του δώσει κάποιους παραμέτρους. Για παράδειγμα σε ποιο αρχείο είναι τα δεδομένα που πρέπει να επεξεργαστεί. Πως "λέμε" στο πρόγραμμα τρέξε την επεξεργασία πάνω σε "αυτό" το αρχείο; Μα με τη γραμμή εντολών.

Η γραμμή εντολών είναι η "κλασσική" και διαχρονική διεπαφή μέσα από την οποία μιλάμε με τον υπολογιστή. Οι υπολογιστές και τα λειτουργικά συστήματα έχουν φτιαχτεί με αυτή τη φιλοσοφία: Ο χρήστης θα μου στέλνει εντολές και εγώ θα τις εκτελώ. Τα γραφικά συστήματα ήρθαν μετά και δουλεύουν "πάνω" από αυτή τη λογική.

Τότε εμείς γιατί κάνουμε jupyter;

Η χρήση της γραμμής εντολών απαιτεί κάποια γνώση και εμπειρία η οποία πολλές φορές δρα αποτρεπτικά σε αυτούς που θέλουν να μάθουν προγραμματισμό "τώρα!". Ειδικά στις επιστήμες στις οποίες ο προγραμματισμός είναι βοηθητικός (βιολογία, χημεία, μηχανική) οι φοιτητές βρίσκουν τη γραμμή εντολών σαν ένα έξτρα εμπόδιο.

Το jupyter και γενικότερα η λογική των notebooks ήρθε να αντικαταστήσουν τη γραμμή εντολών με τη λογική των "κελιών" που έχουμε ήδη μάθει. Με αυτόν τον τρόπο οι φοιτητές μπορούν κατευθείαν να εξασκηθούν στον προγραμματισμό σε ένα γνώριμο για αυτούς περιβάλλον όπως είναι ο browser. Όπως είδαμε όμως το jupyter παρέχε ένα υποσύνολο από τις δυνατότητες ενός σύγχρονου προγραμματιστικού περιβάλλοντος. Πλήρες προγραμματισμό μπορούμε να κάνουμε στο περιβάλλον που δίνει η γραμμή εντολών.

Στον παρακάτω πίνακα υπάρχει μία σύνοψη από τα πλεονεκτήματα και μειονεκτήματα των jupyter / γραμμής εντολών (credit: Ιωάννης-Ραφαήλ Τζονευράκης)

	Jupyter Notebook	Σενάριο γραμμής εντολών .py
Ευχρηστία κατά την συγγραφή κώδικα	**Μεγάλη**	**Μικρότερη**
Φορητότητα (Ευκολία εκτέλεσης του προγράμματος σε διαφορετικό περιβάλλον εργασίας από αυτό που συγγράφηκε)	**Ευνοείται λιγότερο** Πέρα από την ύπαρξη λειτουργικού διερμηνέα της python και την εγκατάσταση των απαραίτητων βιβλιοθηκών, απαιτείται το περιβάλλον εργασίας να είναι συμβατό με το jupyter notebook	**Ευνοείται περισσότερο** Απαιτείται η ύπαρξη λειτουργικού διερμηνέα της python και η εγκατάσταση των απαραίτητων βιβλιοθηκών
Διαχωρισμός κώδικα-δεδομένων	**Πρακτικά ανύπαρκτος** Δεδομένα και κώδικας αναμειγνύονται στον μέγιστο δυνατό βαθμό	**Μεγάλος** Πρέπει ο χρήστης να υποδείξει την θέση των δεδομένων ως παράμετρο
Καθολικές μεταβλητές	Ευνοείται η δημιουργία τους	Δεν ευνοείται η δημιουργία τους
Φιλικότητα προς μακρόχρονες εργασίες	**Μικρή** Η έξοδος του προγράμματος, που θα μπορούσε να περιλαμβάνει πιθανά warnings που να αφορούν την αξιοπιστία της ανάλυσης, δεν συνεχίζει να αποθηκεύεται αν κλείσει ο browser που είναι συνδεδεμένος με αυτό κατά την εκτέλεσή του	**Μεγάλη** Η έξοδος μπορεί να αποθηκευτεί σε αρχεία καταγραφής, τα οποία είναι ανεξάρτητα από την συνεχόμενη σύνδεση του τερματικού από το οποίο ξεκίνησε η εκτέλεση του προγράμματος
Ευκολία εκτέλεσης από τρίτους χρήστες	**Μικρή** Πρέπει οι τρίτοι χρήστες να εντοπίσουν χειροκίνητα κάθε μεταβλητή που αφορά ρυθμίσεις και παραμέτρους της ανάλυσης, ώστε να την προσαρμόσουν στις ανάγκες τους	**Μεγάλη** Όλες οι ρυθμίσεις και οι παράμετροι εκτίθενται ως επιλογές της γραμμής εντολών
Ευκολία αποσφαλμάτωσης	**Μεγάλη** Σε περίπτωση σφάλματος, δεν χάνεται ολόκληρη η κατάσταση του σημειωματάριου, παρά μόνο του κελιού όπου προέκυψε το σφάλμα, επιτρέποντας γρήγορες αλλαγές και διορθώσεις	**Μικρότερη** Σε περίπτωση σφάλματος, χάνεται η κατάσταση όλου του προγράμματος, εκτός και αν έχει υπάρξει μέριμνα για το αντίθετο (πχ αποθήκευση ενδιαμέσων αποτελεσμάτων / checkpoints)

Πως ξεκινάμε με τη γραμμή εντολών;

Υπάρχουν πολλά tutorials online. Για παράδειγμα:

- https://www.youtube.com/watch?v=MBBWVgE0ewk&list=PL6gx4Cwl9DGDV6SnbINlVUd0o2xT4JbMu

Είναι πολύ σημαντικό να παρακολουθήσετε ένα παρόμοιο tutorial και να εξοικειωθείτε με τη γραμμή εντολών. Σε αυτά τα tutorials θα μάθετε επίσης πολλά γύρω από το πως δουλεύουν οι υπολογιστές και πως οργανώνονται τα διάφορα αρχεία.

Όσοι έχετε python anaconda μπορείτε να γράψετε `anaconda prompt` στο search τως windows στο start menu. Δείτε και εδώ https://www.youtube.com/watch?v=dgjEUcccRwM πως γίνεται στο 1:45. Αν το κάνετε αυτό θα ανοίξει ένα περιβάλλον γραμμής εντολών όπου η python θα είναι "στημένη" και ρυθμισμένη σωστά. Αν έχετε κάποιο πρόβλημα στείλτε DM στο slack.

Πως γράφουμε ένα πρόγραμμα για να το τρέξουμε στη γραμμή εντολών;

Για να το κάνετε αυτό, πρέπει να εγκαταστήσετε κάποιον επεξεργαστή κειμένου (text editor) για προγράμματα. **Προσοχή!** το word δεν κάνει για αυτό! Ούτε και το notepad (με πολύ προσπάθεια ίσως τα καταφέρετε στο notepad). Χρειάζεστε ένα ειδικό πρόγραμμα το οποίο βοηθάει τον χρήστη να γράψει python κώδικα. Μερικές επιλογές που έχετε είναι το Notepad++ και το Sublime. Μέσα από τα προγράμματα δοκιμάστε να δημιουργήσετε ένα αρχείο με το όνομα: `program.py` το οποίο να έχει το εξής περιεχόμενο:

```
print ("Hello World!")
```

Που σώζω το αρχείο αυτό;

Το αρχείο θα πρέπει να το σώσετε στον ίδιο κατάλογο από όπου θα εκτελέσετε την εντολή `python program.py` . Μπορείτε μέσα από το command line να δείτε σε ποιο κατάλογο είσαστε. Ο κατάλογος που βρίσκεστε υπάρχει συνήθως πάνω στη γραμμή εντολών με τη μορφή: `C:\Users\Alex` ή κάτι τέτοιο. Αν γράψετε `cd` και πατήσετε enter θα εμφανιστεί επίσης αυτός ο κατάλογος. Στη συνέχεια πρέπει να σώσετε το αρχείο `program.py` σε αυτόν τον κατάλογο από το πρόγραμμα επεξεργασίας κειμένου. Τέλος μπορείτε να "τρέξετε" τον κώδικα που υπάρχει στο αρχείο αυτό με:

```
python program.py
```

Αν δεν συμβεί κάποιο λάθος θα δείτε στο περιβάλλον γραμμής εντολών να εμφανίζεται το κείμενο: "Hello World!".

Τι είναι αυτό το περίεργο `>>>` ;

Αν αντί για `python program.my` πατήσετε `python` και μετά enter, θα μπείτε στο πρόγραμμα γραμμής εντολών της python το οποίο είναι διαφορετικό από αυτό του υπολογιστή σας. Σε αυτό το περιβάλλον απλά γράφετε python. Για να "βγείτε" από αυτό πατήστε `exit()` .

Γράφοντας προγράμματα για τη γραμμή εντολών

Η δημιουργία προγραμμάτων για τη γραμμή εντολής δεν διαφέρει πολύ από το jupyter. Η κύρια διαφορά είναι ότι στα προγράμματα οι εντολές εκτελούνται με τη σειρά που γράφονται. Ενώ στο jupyter μπορεί ένα κελί που βρίσκεται πιο κάτω να εκτελεστεί πριν από ένα κελί που βρίσκεται πιο πάνω. Για παράδειγμα: πρώτα ορίζω το a δύο κελιά πιο κάτω και μετά το τυπώνω ένα κελί πιο κάτω:

In [2]:
```
print (a)
```

```
3
```

In [1]:
```
a=3
```

Σε αντίθεση το παρακάτω πρόγραμμα δεν θα έτρεχε στη γραμμή εντολών:

In [4]:
```
%%writefile program.py

print (a)
a=3
```

```
Writing program.py
```

In [5]:
```
!python program.py
```

```
Traceback (most recent call last):
  File "program.py", line 2, in <module>
    print (a)
NameError: name 'a' is not defined
```

Η μεταβλητή ___name___

Είδαμε πιο πάνω πως να φτιάχνουμε τη δική μας βιβλιοθήκη και πως να κάνουμε import τις συναρτήσεις και τις μεταβλητές που περιέχει. Εδώ όμως παρουσιάσαμε πως να φτιάχνουμε ένα πρόγραμμα το οποίο το τρέχουμε από τη γραμμή εντολών. Τελικά τι από τα δύο θα κάνουμε; Δηλαδή αν πρέπει να "δώσουμε" κάπου τον κώδικά μας, με τι μορφή θα τον δώσουμε; Ως μία βιβλιοθήκη που θα κάνει import ή ως ένα πρόγραμμα που θα τρέξει από τη γραμμή εντολών; Η απάντηση είναι: εσείς διαλέγετε! είτε σανν βιβλιοθήκη είτε από τη γραμμή εντολών ή ΚΑΙ μετα δύο. Ας εξερευνήσουμε λίγο αυτή τη 3η επιλογή.

Για αρχή ας φτιάξουμε μια απλή βιβλιοθήκη:

In [6]:
```
%%writefile program.py

def amazing_function(x):
    return x*2
```

```
Overwriting program.py
```

Και ας τη κάνουμε import:

In [7]:
```
from program import amazing_function
amazing_function(10)
```

Out[7]:
```
20
```

Παρατηρούμε ότι αν "τρέξουμε" το πρόγραμμα program.py δεν συμβαίνει τίποτα, αφού πουθενά δεν τυπώνουμε κάτι μέσα στο program.py:

In [8]:
```
!python program.py
```

Γίνεται άραγε να ξέρω αν με κάλεσε κάποιος μέσα από μία βιβλιοθήκη ή απευθείας από τη γραμμή εντολών; Καταρχήν γιατί αυτό είναι σημαντικό; Όταν δίνουμε ένα πρόγραμμα σε κάποιον άλλο (ή απλά όταν το διαθέτουμε ελεύθερο στο Internet), θέλουμε να δώσουμε τη δυνατητα να μας χρησιμοποιήσουν με τους περισσότερους δυνατούς τρόπους. Κάποιοι προγραμματιστές θα προτιμήσουν να χρησιμοποιήσουν τον κώδικά μας σαν βιβλιοθήκη και κάποιοι να τον τρέξουν από τη γραμμή εντολών.

Όταν τρέχουν τον κώδικά μας σαν βιβλιοθήκη (μέσω της import) τότε αυτός που μας χρησιμοποιεί "περνά" τις παραμέτρους απευθείας στις συναρτήσεις μας. Για παράδειγμα περάσαμε τη τιμή 10 στη συνάρτηση amazing_function. Πως όμως περνάμε τιμές στις συναρτήσεις όταν τρέχουμε ένα πρόγραμμα από τη γραμμή εντολών; Αυτό σηκώνει διερεύνυση!

Για αρχή, ας δούμε τι περιέχει η ειδική μεταβλητή __name__ :

In [9]:
```
%%writefile program.py

def amazing_function(x):
    return x*2

print (__name__)
```

```
Overwriting program.py
```

Τι θα τυπώσει αν κάνουμε import το program;

In [1]:
```
import program
```

```
program
```

Η μεταβλητή περιέχει το όνομα της βιβλιοθήκης στην οποία ανήκει! Για μια στιγμή, όταν το τρέχω από τη γραμμή εντολών δεν υπάρχει κάποια βιβλιοθήκη, τι θα τυπώσει τότε;

In [5]:
```
!python program.py
```

```
__main__
```

Τύπωσε `__main__` ! Ή αλλιώς όταν ένα πρόγραμμα φορτώνεται μέσω import τότε η `__name__` περιέχει το όνομα της βιβλιοθήκης που την περιέχει. Όταν όμως την καλούμε απο τη γραμμή εντολών, τότε περιέχει τη τιμή: `___main__` .

οκ, και πως μπορώ να το χρησιμοποιήσω αυτό;

Μπορώ να γράψω κώδικα ο οποίος εκτελείται μόνο όταν το πρόγραμμα εκτελείται από τη γραμμή εντολών και όχι σαν import και το αντίθετο:

In [6]:
```
%%writefile program.py

def amazing_function(x):
    return x*2

if __name__ == '__main__':
    print ('Με κάλεσαν από τη γραμμή εντολών!')
else:
    print ('Με φόρτωσαν μέσω κάποιας import!')
```

```
Overwriting program.py
```

In [1]:
```
import program
```

```
Με φόρτωσαν μέσω κάποιας import!
```

In [2]:
```
!python program.py
```

```
Με κάλεσαν από τη γραμμή εντολών!
```

Ωραία, είπαμε ότι όταν μας κάνουν import τότε αυτοί που μας φόρτωσαν έχουν την ευθύνη να μας περάσουν και τις παραμέτρους στις συναρτήσεις μας.

Αν όμως ΔΕΝ μας κάναν import και μας χρησιμοποιούν από τη γραμμή εντολών, πως περνάνε παραμέτρους στις συναρτήσεις μας;

Περνώντας παραμέτρους από τη γραμμή εντολών

Το να περνάν παράμετροι από τη γραμμή εντολών μέσα στα προγράμματα που καλούνται απο αυτήν είναι μία από τις πιο έξυπνες ιδέες αλλά και βασική λειτουργία τους. Αν το σκεφτείτε θα δείτε ότι αυτό το κάνετε ήδη όταν γράφετε: `python program.py` . Όταν

η γραμμή εντολών "δει" το `python program.py` , το σπάει σε λέξεις. Η πρώτη λέξη είναι το πρόγραμμα που θα καλέσει. Όλες οι υπόλοιπες λέξεις μετά το πρώτο είναι παράμετροι που θα περάσει στο πρόγραμμα. Στην ουσία δηλαδή αυτό που καταλαβαίνει είναι: "Τρέξε το πρόγραμμα python με την παράμετρο program.py". Η γραμμή εντολών δεν έχει ιδέα ότι το program.py είναι ένα αρχείο. Το καταλαβαίνει ως string το οποίο πρέπει να περάσει σαν παράμετρο στο python.

Για να περάσετε παράμετρους στο πρόγραμμά σας λοιπόν δεν έχετε παρά να προσθέσετε τις παραμέτρους αυτούς στη γραμμή εντολών. Για παράδειγμα:

In [3]:
```
!python program.py 10
```

```
Με κάλεσαν από τη γραμμή εντολών!
```

Ωραίο αλλά πως διαβάζουμε τη τιμή "10" μέσα από τη python; Υπάρχουν 2 τρόποι για να γίνει αυτό. Ο πρώτος είναι χρησιμοποιώντας τη βιβλιοθήκη `sys` η οποία περιέχει τη λίστα `args` (arguments) με όλες τις παραμέτρους από τη γραμμή εντολών:

In [5]:
```
%%writefile program.py
import sys

def amazing_function(x):
    return x*2

if __name__ == '__main__':
    print (sys.argv)
```

```
Overwriting program.py
```

In [6]:
```
!python program.py 10
```

```
['program.py', '10']
```

Παρατηρούμε ότι:

- οι παράμετροι "πέρασαν" στο πρόγραμμά μας σαν μία λίστα.
- Όλοι οι παράμετροι πέρασαν σαν string. Η γραμμή εντολών δεν διακρίνει διαφορετικών τύπου μεταβλητές. Μόνο strings
- Η πρώτη παράμετρος είναι το `program.py` . Αυτό είναι λογικό. Θυμάστε ότι είπαμε ότι η πρώτη λέξη είναι το πρόγραμμα (python) και οι υπόλοιπες είναι οι παράμετροι που περνάμε στο πρόγραμμα.

Τώρα μπορούμε να κάνουμε ό,τι θέλουμε με αυτή τη λίστα. Για παράδειγμα:

In [7]:
```
%%writefile program.py
import sys

def amazing_function(x):
    return x*2

if __name__ == '__main__':
    parameter = int(sys.argv[1])
    print (amazing_function(parameter))
```

```
Overwriting program.py
```

In [8]:
```
!python program.py 10
```

```
20
```

Επίσης το πρόγραμμά μας τρέχει και με τους δύο τρόπους!

In [9]:
```
from program import amazing_function

print (amazing_function(10))
```

```
20
```

Τα σύγχρονα προγράμματα κυρίως στη γενετική και στη βιοπληροφορική έχουν ένα τεράστιο πλήθος παραμέτρων. Για την ακρίβεια προσφέρουν τρομερή εκφραστικότητα. Χρησιμοποιώντας τη γραμμή εντολών μπορείτε να κάνετε πολύπλοκες αναλύσεις χωρίς να γράψετε ούτε μία γραμμή προγραμματισμού. Για παράδειγμα δείτε εδώ μία σύνοψη(!) των παραμέτρων που δέχεται το samtools ένα πρόγραμμα γραμμής εντολών για τη διαχείρηση αρχείων που προέρχονται από πειράματα αλληλούχισης. Άλλα παρόμοια είναι το GATK και το plink2 με πάνω από 100 παραμέτρους.

Αν το πρόγραμμά σας παίρνει το πολύ 3 παραμέτρους το sys.argv ίσως να είναι αρκετό. Για περισσότερους παραμέτρους η λίστα που δίνει το sys.argv δεν είναι αρκετή. Η python διαθέτει τη βιβλιοθήκη argparse για να σας βοηθήσει να "παρσάρετε" τις παραμέτρους από τη γραμμή εντολών και να τη περάσετε "ανώδυνα" στο πρόγραμμά σας.

Προτείνω να διαβάσετε τη πολύ καλή παρουσίαση και ανάλυση της argparse από τη realpython.

Ακολουθεί ένα παράδειγμα. Δηλώνουμε δύο παράμετρους μία υποχρεωτική και μία προαιρετική:

In [18]:
```
%%writefile program.py
import argparse

def amazing_function(x):
    return x*2

if __name__ == '__main__':

    parser = argparse.ArgumentParser(description="Amazing program")
    parser.add_argument('value', help="Number of desired iterations")     #
    parser.add_argument('--par', help="Value of the converge cutoff")     #

    args = parser.parse_args()

    value = int(args.value)
    par = args.par

    print (amazing_function(value))
    if not par is None:
        print ('The value of par={}'.format(par))
```

```
Overwriting program.py
```

Καλώντας με τη παράμετρο -h το πρόγραμμά μας εμφανίζει ένα ενημερωτικό μήνυμα με τον τρόπο που μπορεί κάποιος να το χρησιμοποιείσει:

In [11]:
```
!python program.py -h
```

```
usage: program.py [-h] [--par PAR] value

Amazing program
```

```
positional arguments:
  value       Number of desired iterations

optional arguments:
  -h, --help  show this help message and exit
```

Ας το καλέσουμε:

In [14]:
```
# Ας μη βάλουμε καμία παράμετρο.. Θα πετάξει μήνυμα λάθους
# Γιατί υπάρχει υποχρεωτική παράμετρο που δεν έχουμε βάλει!
!python program.py
```

```
usage: program.py [-h] [--par PAR] value
program.py: error: the following arguments are required: value
```

In [19]:
```
# Ας βάλουμε την υποχρεωτική παράμετρο:
!python program.py 10
```

```
20
```

In [20]:
```
# Ας βάλουμε την υποχρεωτική και την προαιρετική παράμετρο:
!python program.py 10 --par hello
```

```
20
The value of par=hello
```

Επαναλαμβάνω ότι όλος ο κώδικας που παρσάρει και επεξεργάζεται τις παραμέτρους είναι ανενεργός αν το κάνουμε import:

In [21]:
```
from program import amazing_function
```

Exceptions

Θα έχετε προσέξει ότι όταν συμβεί κάποιο λάθος η python πετάει ένα μήνυμα:

In [22]:
```
a=1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-22-023a503edd86> in <module>
----> 1 a=1/0

ZeroDivisionError: division by zero
```

Όλα τα λάθη που μπορούν να συμβούν στη python ανήκουν σε μία νέα κατηγορία αντικειμένων: τα exceptions. Ένα exception όταν συμβεί ΤΕΡΜΑΤΙΖΕΙ το πρόγραμμα:

In [26]:
```
a=1/0
print ("hello")
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-26-310b257491c9> in <module>
----> 1 a=1/0
      2 print ("hello")

ZeroDivisionError: division by zero
```

Παρατηρήστε ότι το πρόγραμμα δεν τύπωσε ποτέ "hello" γιατί απλά "κράσαρε" ή αλλιώς τερμάτισε απρόοπτα όταν συναίνει ένα λάθος (διαίρεση με το 0)

Τι γίνεται όταν θέλουμε να συνεχιστεί η εκτέλεση του προγράμματος ακόμα και όταν έχει συμβεί ένα exception; Μπορούμε να τρέξουμε το κομμάτι κώδικα μέσα σε ένα try..except μπλοκ:

In [25]:
```
try:
    a=1/0
except:
    print ("Something bad happened")
```

```
Something bad happened
```

Οι εντολές μετά το σημείο που έγινε το exception ΔΕΝ ΤΡΕΧΟΥΝ!

In [27]:
```
try:
    a=1/0
    print ("Αυτό δεν θα τρέξει ποτέ..")
except:
    print ("Something bad happened")
```

```
Something bad happened
```

Οι εντολές όμως μετά το try..except μπλοκ, τρέχουν κανονικά:

In [28]:
```
try:
    a=1/0
    print ("Αυτό δεν θα τρέξει ποτέ..")
except:
    print ("Something bad happened")

print ("Αυτό θα τρέξει κανικά!")
```

```
Something bad happened
Αυτό θα τρέξει κανικά!
```

Υπάρχουν πολλά ειδή exceptions. Εδώ υπάρχει μία λίστα με όλες αυτές. Αντί να χρησιμοποιήσουμε το try..except μπορούμε να επιλέξουμε ποια ακριβώς exception θέλουμε να πιάσουμε:

In [29]:
```
try:
    a=1/0
except ZeroDivisionError:
    print ('Διαίρεση με το 0')
```

```
Διαίρεση με το 0
```

Μπορούμε έτσι να "πιάσουμε" πολλών ειδών λάθη:

In [30]:
```
divisor = 0
try:
    a=1/divisor
    open('I_DO_NOT_EXIST.txt')
except ZeroDivisionError:
    print ('Διαίρεση με το 0')
except FileNotFoundError:
    print ('Δεν βρέθηκε κάποιο αρχείο')
except:
    print ('Κάτι άλλο περιέργο συναίβει')
```

```
Διαίρεση με το 0
```

In [31]:
```
divisor = 1
try:
    a=1/divisor
    open('I_DO_NOT_EXIST.txt')
except ZeroDivisionError:
    print ('Διαίρεση με το 0')
except FileNotFoundError:
    print ('Δεν βρέθηκε κάποιο αρχείο')
except:
    print ('Κάτι άλλο περιέργο συναίβει')
```

```
Δεν βρέθηκε κάποιο αρχείο
```

In [32]:
```
divisor = 1
b = [1,2]
try:
    a=1/divisor
    b[3] = 5
    open('I_DO_NOT_EXIST.txt')
except ZeroDivisionError:
    print ('Διαίρεση με το 0')
except FileNotFoundError:
    print ('Δεν βρέθηκε κάποιο αρχείο')
except:
    print ('Κάτι άλλο περιέργο συνέβει')
```

```
Κάτι άλλο περιέργο συνέβει
```

Αν χρησιμοποιήσουμε:

```
try:
    ...
except:
    ...
```

ή

```
try:
    ...
except Exception
    ....
```

Τότε έχουμε πιάσει ΟΛΑ τα δυνατά exceptions.

Μπορούμε να αποθηκεύσουμε το exception που συνέβει σε μία εντολή και να πάρουμε περισσότερες πληροφορίες για το λάθος που το προκάλεσε:

In [33]:
```
try:
    a=1/0
except Exception as e:
    print ('This error happened: {}'.format(e))
```

```
This error happened: division by zero
```

Μπορούμε να πιάσουμε πολλά ειδών exceptions και να την αποθηκεύσουμε σε μία μεταβλητή:

In [35]:
```
divisor = 1
try:
    a=1/divisor
    open('I_DO_NOT_EXIST.txt')
except (ZeroDivisionError, FileNotFoundError) as e:
    print ('This just happened: {}'.format(e))
```

```
This just happened: [Errno 2] No such file or directory: 'I_DO_NOT_EXIST.tx
t'
```

Επίσης αντί για το try..except μπορούμε να χρησιμοποιήσουμε το try..except..else. Ο κώδικα που βρίσκεται μέσα στο `else` εκτελείτε μόνο αν ΔΕΝ έχει συμβεί κάποιο exception.

In [36]:
```
divisor=0
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed..')
else:
    print ('Division succeeded!!')
```

```
Division failed..
```

In [37]:
```
divisor=1
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed..')
else:
    print ('Division succeeded!!')
```

```
Division succeeded!!
```

Επίσης μπορούμε να χρησιμοποιήσουμε το try..except..finally. Ο κώδικας που βρίσκεταί μέσα στο `finally` τρέχει είτε συμβεί είτε δεν συμβεί exception:

In [38]:
```
divisor=0
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed..')
finally:
    print ('I always run')
```

```
Division failed..
I always run
```

In [39]:
```
divisor=1
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed..')
finally:
    print ('I always run')
```

```
I always run
```

Τότε τι νόημα έχει το `finally` ; Αν συμβεί exception ο κώδικας του finally θα τρέχει **μετά** το exception. Αυτό είναι πολύ χρήσιμο όταν θέλουμε να "συμαζέψουμε" (clean-up) την ανακατοσούρα που κάναμε μέσα στο try. Π.χ. να κλείσουμε τα αρχεία που ανοίξαμε. Για παράδειγμα:

In [42]:
```
def save_weird_calculation_to_file(file, a,b):
    value = a/b
    file.write("{}\n".format(value))

f = open("amazing_results.txt", 'w')
a=4
b=1
try:
    save_weird_calculation_to_file(f, a, b)
finally:
    print ('Closing file!')
    f.close()
```

```
Closing file!
```

Στο παραπάνω παρατηρούμε ότι δεν συνέβη κανένα λάθος και έκλεισε κανονικά το αρχείο. Ας βάλουμε στο b τη τιμή 0:

In [43]:
```
def save_weird_calculation_to_file(file, a,b):
    value = a/b
    file.write("{}\n".format(value))

f = open("amazing_results.txt", 'w')
a=4
b=0
try:
    save_weird_calculation_to_file(f, a, b)
finally:
    print ('Closing file!')
    f.close()
```

```
Closing file!
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-43-287591b9dc58> in <module>
      7 b=0
      8 try:
----> 9     save_weird_calculation_to_file(f, a, b)
     10 finally:
     11     print ('Closing file!')

<ipython-input-43-287591b9dc58> in save_weird_calculation_to_file(file, a,
b)
      1 def save_weird_calculation_to_file(file, a,b):
----> 2     value = a/b
      3     file.write("{}\n".format(value))
      4
      5 f = open("amazing_results.txt", 'w')

ZeroDivisionError: division by zero
```

Παρατηρούμε το εξής: Έγινε το λάθος, πέταξε exception και παρόλα αυτά έκλεισε το αρχείο. Αν δεν είχαμε βάλει το try..finally το αρχείο δεν θα έκλεινε.

Ένα παράδειγμα που χρησιμοποιεί το try..except..else..finally:

In [44]:
```
divisor = 0
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed')
else:
    print ('Division succeeded')
finally:
    print ('I always run')
```

```
Division failed
I always run
```

ΠΡΟΣΟΧΗ! Το ότι έχουμε χρησιμοποιήσει try..except..else..finally κτλ δεν σημαίνει ότι έχουμε "καλυφθεί" από πιθανά exceptions:

In [45]:
```
try:
    mpaklavas
except ZeroDivisionError:
    print ('Division with zero')
else:
    print ('No exception happened')
finally:
    print ('I always run')
```

```
I always run
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-45-0457af044c52> in <module>
      1 try:
----> 2     mpaklavas
      3 except ZeroDivisionError:
      4     print ('Division with zero')
      5 else:

NameError: name 'mpaklavas' is not defined
```

Στο παραπάνω παράδειγμα συνέβει ένα `NameError` exception το οποίο δεν το πιάσαμε πουθενά.

Μπορούμε να "πετάξουμε" ή "σηκώσουμε" (raise) και το δικό μας exception:

In [47]:
```
raise Exception('Χάλασε το αριστερό φιλάντζι') # https://www.youtube.com/wa
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-47-d41eda379597> in <module>
----> 1 raise Exception('Χάλασε το αριστερό φιλάντζι') # https://www.youtub
e.com/watch?v=DrwVB4vMx-Q&feature=youtu.be&t=30

Exception: Χάλασε το αριστερό φιλάντζι
```

In [54]:
```
def print_my_name(name):
    if name in ['hell', 'her', 'Jane', 'Stacey', 'quiet']:
        raise NameError("That's not my name") # https://www.youtube.com/wat

    print ('My name is:', name)
```

In [55]:
```
print_my_name('Alex')
```

```
My name is: Alex
```

In [56]:
```
print_my_name('Stacey')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-56-5347c2f26f4c> in <module>
----> 1 print_my_name('Stacey')

<ipython-input-54-bc05d5512eaa> in print_my_name(name)
      1 def print_my_name(name):
      2     if name in ['hell', 'her', 'Jane', 'Stacey', 'quiet']:
----> 3         raise NameError("That's not my name") # https://www.youtub
e.com/watch?v=v1c2OfAzDTI
      4
      5     print ('My name is:', name)

NameError: That's not my name
```

Η πρωτοπορία των exceptions είναι ότι μπορούμε να τα "πιάσουμε" όχι μόνο ακριβώς στο σημείο που συνέβησαν αλλά και έξω ακόμα από τη συνάρτηση που τα προκάλεσε:

In [57]:
```
def f():
    raise Exception('Άσε με να κάνω λάθος!')

def g():
    f()

def h():
    g()

try:
    h() # Και κάπου εδώ ίσως υπάρχει ένα μικρό exception
except Exception as e:
    print ('Error: {}'.format(e))
```

```
Error: Άσε με να κάνω λάθος!
```

Προχωρημένο: Μπορούμε να φτιάξουμε τη δική μας exception:

In [58]:
```
# Θα εξηγήσουμε αργότερα στις κλάσεις τι σημαίνουν όλα αυτά
class MyFabulousException(Exception):
    pass
```

In [59]:
```
raise MyFabulousException
```

```
---------------------------------------------------------------------------
MyFabulousException                       Traceback (most recent call last)
<ipython-input-59-cb966f875a08> in <module>
----> 1 raise MyFabulousException

MyFabulousException:
```

Collections

Η βιβλιοθήκη collections περιέχει κάποιες επιπλέον δομές (πέρα από τις λίστες, λεξικά και σετ) για τη διαχείρηση δεδομένων. Δύο από τις πιο σημαντικές είναι η Counter και η defaultdict. Η Counter επιτρέφει μία μέτρηση των αντικειμένων που περιέχει μία οποιαδήποτε δομή:

In [60]:
```
from collections import Counter
```

In [61]:
```
Counter('hello world')
```

Out[61]:
```
Counter({'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

In [71]:
```
Counter([5,4,3,5,6,7,6,5,6,4,3,2,1,2,8])
```

Out[71]:
```
Counter({5: 3, 4: 2, 3: 2, 6: 3, 7: 1, 2: 2, 1: 1, 8: 1})
```

Το πιο κοινό στοιχείο:

In [73]:
```
a = Counter('helloworld')
a.most_common(1)[0][0]
```

Out[73]:
```
'l'
```

Μπορούμε να προσθέσουμε δύο Counter μεταξύ τους:

In [74]:
```
a = Counter('Mitsos')
b = Counter('Kostas')
print (a+b)
```

```
Counter({'s': 4, 't': 2, 'o': 2, 'M': 1, 'i': 1, 'K': 1, 'a': 1})
```

Το defaultdict είναι ένα λεξικό (dictionary) στο οποίο οποίο μπορούμε να ορίσουμε μία προκαθορισμένη τιμή:

In [75]:
```
from collections import defaultdict

d = {}

a = defaultdict(int) # Βάλε 0 όταν γίνει πρόσβαση σε ένα στοιχείο που δεν υ
```

In [76]:
```
print (a['mitsos'])
```

```
0
```

In [77]:
```
print (d['mitsos']) # πετάει λάθος!
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-77-20faa7276cea> in <module>
----> 1 print (d['mitsos']) # πετάει λάθος!

KeyError: 'mitsos'
```

In [78]:
```
a = defaultdict(list)

print (a['mitsos'])
```

```
[]
```

Που είναι χρήσιμο αυτό; Πολλές φορές όταν κάνουμε μία επανάληψη, θέλουμε να κάνουμε μία πράξη με τα στοιχεία της επανάληψης μέσα σε ένα dictionary. Κανονικά κάθε φορά θα πρέπει να ελέγχουμε αν υπάρχει το κλειδί μέσα στο dictionary και αν δεν υπάρχει να το δημιουργούμε. Μπορούμε να το αποφύγουμε αυτό με τα defaultdict. Για παράδειγμα, σου δίνω μία λίστα με strings. Φτιάξε ένα λεξικό όπου κάθε κλειδί θα έχει ένα αριθμό που θα είναι το μήκος του string. Οι τιμές θα είναι μία λίστα με τα strings της λίστας που έχουν αυτό το μέγεθος.

In [82]:
```
l = '''ἄνδρα μοι ἔννεπε, Μοῦσα, πολύτροπον, ὃς μάλα πολλὰ
πλάγχθη, ἐπεὶ Τροίης ἱερὸν πτολίεθρον ἔπερσε·
πολλῶν δ' ἀνθρώπων ἴδεν ἄστεα καὶ νόον ἔγνω,
πολλὰ δ' ὅ γ' ἐν πόντῳ πάθεν ἄλγεα ὃν κατὰ θυμόν,
ἀρνύμενος ἥν τε ψυχὴν καὶ νόστον ἑταίρων.
ἀλλ' οὐδ' ὣς ἑτάρους ἐρρύσατο, ἱέμενός περ·
αὐτῶν γὰρ σφετέρῃσιν ἀτασθαλίῃσιν ὄλοντο,
νήπιοι, οἳ κατὰ βοῦς Ὑπερίονος Ἠελίοιο
ἤσθιον· αὐτὰρ ὁ τοῖσιν ἀφείλετο νόστιμον ἦμαρ.
τῶν ἀμόθεν γε, θεά, θύγατερ Διός, εἰπὲ καὶ ἡμῖν.'''.split()
```

Χωρίς defaultdict:

In [83]:
```
d = {}
for word in l:
    length = len(word)
    if not length in d:
        d[length] = []

    d[length].append(word)

print (d)
```

```
{5: ['ἄνδρα', 'πολλὰ', 'ἱερὸν', 'ἄστεα', 'ἔγνω,', 'πολλὰ', 'πόντῳ', 'πάθεν
', 'ἄλγεα', 'ψυχὴν', 'αὐτῶν', 'αὐτὰρ', 'ἦμαρ.', 'Διός,', 'ἡμῖν.'], 3: ['μοι
', 'καὶ', 'καὶ', 'γὰρ', 'τῶν', 'γε,', 'καὶ'], 7: ['ἔννεπε,', 'ἔπερσε·', 'ἐτ
άρους', 'ἱέμενός', 'ὅλοντο,', 'νήπιοι,', 'Ἠελίοιο', 'ἤσθιον·', 'θύγατερ'],
6: ['Μοῦσα,', 'Τροίης', 'πολλῶν', 'θυμόν,', 'νόστον', 'τοῖσιν', 'ἀμόθεν'],
11: ['πολύτροπον,'], 2: ['ὃς', 'δ᾽', 'δ᾽', 'γ᾽', 'ἐν', 'ὃν', 'ἦν', 'τε', 'ὣ
ς', 'οἳ'], 4: ['μάλα', 'ἐπεὶ', 'ἴδεν', 'νόον', 'κατὰ', 'ἀλλ᾽', 'οὐδ᾽', 'περ
·', 'κατὰ', 'βοῦς', 'θεά,', 'εἰπὲ'], 8: ['πλάγχθη,', 'ἀνθρώπων', 'ἑταίρω
ν.', 'ἀφείλετο', 'νόστιμον'], 10: ['πτολίεθρον', 'σφετέρῃσιν'], 1: ['ὅ', 'ὁ
'], 9: ['ἀρνύμενος', 'ἐρρύσατο,', 'Ὑπερίονος'], 12: ['ἀτασθαλίῃσιν']}
```

Με defaultdict:

In [84]:
```
d = defaultdict(list)
for word in l:
    d[len(word)].append(word)
print(d)
```

```
defaultdict(<class 'list'>, {5: ['ἄνδρα', 'πολλὰ', 'ἱερὸν', 'ἄστεα', 'ἔγν
ω,', 'πολλὰ', 'πόντῳ', 'πάθεν', 'ἄλγεα', 'ψυχὴν', 'αὐτῶν', 'αὐτὰρ', 'ἦμα
ρ.', 'Διός,', 'ἡμῖν.'], 3: ['μοι', 'καὶ', 'καὶ', 'γὰρ', 'τῶν', 'γε,', 'καὶ
'], 7: ['ἔννεπε,', 'ἔπερσε·', 'ἐτάρους', 'ἱέμενός', 'ὅλοντο,', 'νήπιοι,', '
Ἠελίοιο', 'ἤσθιον·', 'θύγατερ'], 6: ['Μοῦσα,', 'Τροίης', 'πολλῶν', 'θυμό
ν,', 'νόστον', 'τοῖσιν', 'ἀμόθεν'], 11: ['πολύτροπον,'], 2: ['ὃς', 'δ᾽', 'δ
᾽', 'γ᾽', 'ἐν', 'ὃν', 'ἦν', 'τε', 'ὣς', 'οἳ'], 4: ['μάλα', 'ἐπεὶ', 'ἴδεν',
'νόον', 'κατὰ', 'ἀλλ᾽', 'οὐδ᾽', 'περ·', 'κατὰ', 'βοῦς', 'θεά,', 'εἰπὲ'], 8:
['πλάγχθη,', 'ἀνθρώπων', 'ἑταίρων.', 'ἀφείλετο', 'νόστιμον'], 10: ['πτολίεθ
ρον', 'σφετέρῃσιν'], 1: ['ὅ', 'ὁ'], 9: ['ἀρνύμενος', 'ἐρρύσατο,', 'Ὑπερίονο
ς'], 12: ['ἀτασθαλίῃσιν']})
```

In [ ]: