Προγραμματισμός με τη γλώσσα python

[Alexandros Kanterakis (mailto:kantale@ics.forth.gr)](mailto:kantale@ics.forth.gr) **kantale@ics.forth.gr**

Διάλεξη 7η, Τρίτη 26 Νοεμβρίου 2019

Κάποιες σημειώσεις για τις λίστες και τα dictionaries

Πολλές φορές υπάρχει το ζήτημα τι να χρησιμοποιήσω: λίστες ή dictionaries. Τα περισσότερα πράγματα μπορούν να γίνουν και με τα δύο. Σε περίπτωση που υπάρχει σύγχυση οι παρακάτω σημειώσεις μπορούν να βοηθήσουν:

Για να εντοπίσει η python αν ένα στοιχεία υπάρχει (ή όχι) σε μία λίστα κάνει ότι θα κάναμε σε έναν αταξινόμητο κατάλογο: Ψάχνει μία-μία όλες τις εγγραφές. Αντίθετα στο dictionary ξέρει "που" θα βρει το κάθε κλειδί. Αυτό το επιτυγχάνει με ειδικές δομές δεδομένων και αλγόριθμους([hash tables (https://en.wikipedia.org/wiki/Hash_table)](https://en.wikipedia.org/wiki/Hash_table)). Ένα παράδειγμα. Μία λίστα με όλους τους αριθμούς από 1 μέχρι το 1.000.000:

In [139]:
```
a = list(range(1000000))
```

Η παρακάτω εντολή `%%timeit` μετράει πόσο "γρήγορα τρέχει" μία εντολή:

In [152]:
```
%%timeit
234234 in a # 0.00314 seconds
```

```
3.14 ms ± 197 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Ας κάνουμε το ίδιο με dictionary:

In [142]:
```
b = {x:5 for x in range(1000000)}
```

In [153]:
```
%%timeit
234234 in b #0.000062 seconds
```

```
62 ns ± 1.12 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Βλέπουμε δηλαδή ότι η αναζήτηση στο dictionary είναι 0.00314/0.000062 ~= 50 φορές πιο γρήγορη. Για σύγκριση μπορούμε να δούμε πόσο κάνει η python για να μη κάνει... Τίποτα:

In [149]:
```
%%timeit
pass
```

```
9.37 ns ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
```

Βέβαια πληρώσαμε ένα τίμημα για αυτή τη ταχύτητα. Η συνάρτηση `getsizeof` επιστρέφει τη μνήμη που καταναλώνει μία μεταβλητή:

In [9]:
```
from sys import getsizeof
```

In [10]:
```
getsizeof(a)
```

Out[10]:
```
9000112
```

In [11]:
```
getsizeof(b)
```

Out[11]:
```
41943144
```

Δηλαδή το dictionary θέλει: 41943144/9000112 ~= 4.5 φορές παραπάνω μνήμη. **ΠΡΟΣΟΧΗ!** Αυτές οι μετρήσεις είναι ενδεικτικές μη βασίζεται τις υλοποιήσεις σας σε αυτές. Υπάρχουν πολλές μετρήσεις με πολύ πιο αντικειμενικό τρόπο για το ποια δομή είναι καλύτερη.

Πως κάνουμε join μία λίστα που έχει (και) μη-string μεταβλητες;

In [54]:
```
'-'.join(map(str,['a','b',4,'c']))
```

Out[54]:
```
'a-b-4-c'
```

Πως "στέλνουμε" μία πολύπλοκη δομή δεδομένων;

Ας υποθέσουμε ότι έχουμε τη παρακάτω "πολύπλοκη" δομή:

In [155]:
```
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 [157]:
```
import json
a_json = json.dumps(a)
print (a_json)
```

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

In [159]:
```
type(a_json)
```

Out[159]:
```
str
```

Οπότε μπορούμε να σώσουμε το a_json σε ένα αρχείο:

In [160]:
```
with open('results.txt', 'w') as f:
    f.write(a_json + '\n')
```

Ή αλλιώς:

In [163]:
```
with open('results.txt', 'w') as f:
    json.dump(a, f)
```

In [165]:
```
!cat results.txt
```

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

Αφού στείλουμε το αρχείο μπορεί ο παραλήπτης να το ανοίξει:

In [168]:
```
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 (https://en.wikipedia.org/wiki/Serialization)

Exceptions

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

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

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-1-023a503edd86> 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:

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 [2]:
```
try:
    a=1/0
except:
    print ("Something bad happened")
```

```
Something bad happened
```

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

In [3]:
```
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 [4]:
```
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 (https://docs.python.org/3/library/exceptions.html). Αντί να χρησιμοποιήσουμε το try..except μπορούμε να επιλέξουμε ποια ακριβώς exception θέλουμε να πιάσουμε:

In [5]:
```
try:
    a=1/0
except ZeroDivisionError:
    print ('Διαίρεση με το 0')
```

```
Διαίρεση με το 0
```

Μπορούμε έτσι να "πιάσουμε" πολλών ειδών λάθη:

In [6]:
```
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 [7]:
```
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 [8]:
```
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 [9]:
```
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 [10]:
```
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.txt'
```

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 [14]:
```
divisor=0
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed..')
else:
    print ('Division succeeded!!')
```

```
Division failed..
```

In [15]:
```
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 [13]:
```
divisor=0
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed..')
finally:
    print ('I always run')
```

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

In [16]:
```
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` ; Χρησιμοποιείτε όταν θέλουμε να "συμαζέψουμε" (clean-up) την ανακατοσούρα που κάναμε μέσα στο try. Π.χ. να κλείσουμε τα αρχεία που ανοίξαμε.

Μπορούμε να χρησιμοποιήσουμε και το try..except..else..finally :

In [17]:
```
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 [18]:
```
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-18-0457af044c52> 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 "ρίξουμε" και το δικό μας exception:

In [19]:
```
raise Exception('Χάλασε το δεξί φιλάντζι') # https://www.youtube.com/watch?v=DrwVB4vMx-Q&feature=youtu.be&t=30
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-19-30beae3ba6e0> in <module>()
----> 1 raise Exception('Χάλασε το δεξί φιλάντζι') # https://www.youtube.com/watch?v=DrwVB4vMx-Q&feature=youtu.be&t=30

Exception: Χάλασε το δεξί φιλάντζι
```

In [20]:
```
raise NameError("Stacey? That's not my name") # https://www.youtube.com/watch?v=v1c2OfAzDTI
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-20-ca4b8bf54983> in <module>()
----> 1 raise NameError("Stacey? That's not my name") # https://www.youtube.com/watch?v=v1c2OfAzDTI

NameError: Stacey? 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 [21]:
```
def f():
    raise Exception('Άσε με να κάνω λάθος!')

def g():
    f()

def h():
    g()

try:
    h()
except Exception as e:
    print ('Error: {}'.format(e))
```

```
Error: Άσε με να κάνω λάθος!
```

Προχωρημένο: Μπορούμε να φτιάξουμε τη δική μας exception:

In [22]:
```
# Θα εξηγήσουμε αργότερα στις κλάσεις τι σημαίνουν όλα αυτά
class MyFabulousException(Exception):
    pass
```

In [23]:
```
raise MyFabulousException
```

```
---------------------------------------------------------------------------
MyFabulousException                       Traceback (most recent call last)
<ipython-input-23-cb966f875a08> in <module>()
----> 1 raise MyFabulousException

MyFabulousException:
```

Κλάσεις (μέρος 1ο)

Αλήθεια δεν έχετε βαρεθεί με όλα αυτά τα lists, dictionaries, sets κτλ και με όλες τις ιδιοτροπίες τους; Γιατί πρέπει να βάζω τα γονίδια σε λίστες, τα χρωμοσώματα σε dictionaries κτλ; Δεν θα ήταν ωραίο να έφτιαχνα τον δικό μου τρόπο που διαχειρίζομαι και αποθηκεύω τα δεδομένα μου;

Αυτό ακριβώς κάνουν οι κλάσεις!

Με τις κλάσεις ορίζετε εσείς τι πράξεις μπορώ να κάνω στα δεδομένα, πως αποθηκεύονται, πως τυπώνονται, πως προσθέτω νέα, πως σβήνω παλιά, πως...

Αλλά αρκετά με τα λόγια. Ας δούμε την πιο απλή κλάση που μπορεί να υπάρχει:

In [24]:
```
class Human:
    pass
```

Μόλις έχω φτιάξει μία νέα κλάση. Τα lists, dictionaries, sets, exceptions είναι και αυτά κλάσεις. Ας φτιάξουμε τώρα ένα *αντικείμενο* από αυτή τη κλάση:

In [25]:
```
alex = Human()
```

Το alex είναι μία νέα μεταβλητή τύπου Human.

Θυμηθείτε λίγο το:

```
a = [1,2,3]
```

Όπως ακριβώς το a ήταν μία μεταβλητή τύπου list, έτσι και το alex είναι μία μεταβλητή τύπου Human. Δηλαδή έχουμε φτιάξει ένα νέο τύπου μεταβλητής.

In [26]:
```
type(alex)
```

Out[26]:
```
__main__.Human
```

Εδώ πρέπει να κάνουμε ένα διάλειμα και να μιλήσουμε λίγο για την ορολογία. Όταν κάνουμε:

```
a=3
```

τότε λέμε ότι το a είναι μία μεταβλητή τύπου int και η τιμή της είναι 3.

Όταν όμως λέμε:

```
alex=Human()
```

Τότε τι τιμή έχει το alex;

Ακριβώς επειδή δεν μπορούμε να απαντήσουμε με ακρίβεια (και συντομία) σε αυτήν την ερώτηση, λέμε ότι το alex είναι ένα ξεχωριστό είδος μεταβλητής, το οποίο ονομάζουμε **αντικείμενο** (object). Ή για να είμαστε πιο ακριβείς, ένα αντικείμενο τύπου Human.

Όταν φτιάχνουμε έναν δικό μας τύπο μεταβλητής (δηλαδή όταν φτιάχνουμε ένα αντικείμενο μίας κλάσης), τότε μπορούμε να την κάνουμε να έχει ό,τι ιδιότητα θέλουμε:

In [27]:
```
alex.name = "Αλέξανδρος"
alex.age = 40
```

In [28]:
```
print (alex.name, alex.age)
```

```
Αλέξανδρος 40
```

Έχουμε ξαναδεί αυτό το `μεταβλητή.ιδιότητα` και πιο παλιά.

όταν κάναμε:

```
a = [1,2,4]
a.append(5)
```

Τότε χρησιμοποιούσαμε την ιδιότητα `append` της κλάσης `list`

Ας φτιάξω τώρα στον alex μία ιδιότητα που υπολογίζει αν είναι ενήλικας ή όχι. Πως θα το κάνω αυτό; Ένας τρόπος είναι αυτός:

In [29]:
```
def is_adult(human):
    return human.age >= 18

is_adult(alex)
```

Out[29]:
```
True
```

Αν και το παραπάνω μας έδωσε το αποτέλεσμα που θέλαμε, η συνάρτηση is_adult ΔΕΝ είναι ιδιότητα του alex (όπως π.χ είναι το name και το age).

Δηλαδή αντί για:

```
is_adult(alex)
```

Εμείς θέλουμε να κάνουμε:

```
alex.is_adult()
```

Προσέξτε ότι η is_adult() τώρα ΔΕΝ παίρνει κάποιο όρισμα! Τότε πως θα μπορέσω εγώ να πάρω το age για να δω αν είναι adult ή όχι;

Η python μας δίνει τη δυνατότητα να φτιάξουμε συναρτήσεις οι οποίες μπορούν να είναι ιδιότητες σε ένα αντικείμενο. Για να γίνει αυτό, πρέπει να ικανοποιηθούν 2 προϋποθέσεις:

- Η πρώτη προϋπόθεση είναι ότι το πρώτο όρισμα της συνάρτησης πρέπει να είναι μία μεταβλητή που ονομάζεται: `self`. Η `self` περιέχει το αντικείμενο για το οποίο "καλείται" η συνάρτηση αυτή. Δηλαδή όταν λέμε: `αντικείμενο.μέθοδος()`, το `self` είναι το `αντικείμενο`. Στη περίπτωσή μας, όταν λέμε `alex.is_adult()`, το `self` περιέχει το αντικείμενο `alex`.
- Η συνάρτηση πρέπει να ορίζεται μέσα στη κλάση και όχι μέσα στο αντικείμενο.

Αυτές οι δύο προυποθέσεις συνοψίζονται στο παρακάτω παράδειγμα:

In [31]:
```
class Human:
    # Η Συνάρτηση ορίζεται μέσα στη κλάση
    def is_adult(self,): # Το πρώτο όρισμα είναι το self
        return self.age >= 18
```

Τώρα μπορώ να κάνω:

In [32]:
```
alex = Human()
alex.age = 40
alex.is_adult()
```

Out[32]:
```
True
```

Το παραπάνω κρύβει πολύ "φιλοσοφία" μέσα του. Ας το ξαναγράψουμε:

```
alex = Human()
alex.age = 40
alex.is_adult()
```

Προσέξτε το εξής: Έχω ορίσει έναν νέο τύπο μεταβλητής (Human) το οποίο αν του βάλω ένα πεδίο με το όνομα age τότε μπορεί να υπολογίσει αν αυτό το Human είναι adult ή όχι. Δηλαδή ο νέος τύπος (Human) δημιουργεί μεταβλητές που έχουν μια *συμπεριφορά*: αυτή του να υπολογίζουν αν η μεταβλητή αναφέρεται σε ενήλικο ή όχι. Επίσης παρατηρούμε ότι κάποιος δεν χρειάζεται να έχει ιδέα για το πως έχει υλοποιηθεί η is_adult, αρκεί να ξέρει ότι είναι *μέθοδος* της κλάσης. Όπως ακριβώς δεν έχουμε ιδέα πως έχει υλοποιηθεί η append στις λίστες. Μέθοδος μίας κλάσης είναι μία ιδιότητα η οποία είναι συνάρτηση.

Το `alex.is_adult()` με το `is_adult(alex)` έχουν μια τεράστια διαφορά: Αν διαβάσουμε το πρώτο (`alex.is_adult()`) από αριστερά προς τα δεξιά τότε πάμε από το γενικό (`alex`) προς το ειδικό (`is_adult`). Ενώ αν διαβάσουμε το δεύτερο τότε πάμε από το ειδικό προς το γενικό.

Στη "πραγματική" ζωή τι είναι πιο πιθανό να ρωτάγαμε: "Είναι ο Alex ενήλικας;" ή "Είναι ενήλικας ο Alex;" (δοκιμάστε το και στα Αγγλικά που δεν επιτρέπουν τόσο ποικιλία στη θέση των λέξεων όσο τα Ελληνικά).

Και λίγο ιστορία

Στη προσπάθεια λοιπόν να γίνει ο προγραμματισμός πιο "φυσιολογικός" και πιο κοντά στην ανθρώπινη αντίληψη, εισήχθει τη δεκαετία του '50 η έννοια του *αντικειμενοστραφή* προγραμματισμού (https://en.wikipedia.org/wiki/Object-oriented_programming). Όπως και όλες σχεδόν οι προγραμματιστικές έννοιες έχει περάσει απο "40 κύματα", δηλαδή με πολλές υλοποιήσεις και ορισμούς.

Αυτό που πρέπει να κρατήσουμε είναι ότι μέσω του ΑΣΠ (ΑντικειμενοΣτραφής Προγραμματισμός) (OOP στα Αγγλικά) μπορούμε να χρησιμοποιήσουμε το συντακτικό της python για να γράψουμε πράγματα που βγάζουν πιο έυκολα νόημα. π.χ.:

Δηλαδή ένα κομμάτι ΑΣΠ προγραμματισμού τύπικα έχει εντολές όπως αυτές:

```
if geneA.is_in_between(geneB)...
if debt.is_paid()....
alex.hire()
if circle_A > circle_B ...
```

Οι ίδιες εντολές αν υποθέσουμε ότι χρησιμοποιούμε μόνο lists, dictionaries, sets κτλ και δεν χρησιμοποιούμε ΑΣΠ θα ήταν κάπως έτσι:

```
if geneA['start'] > geneB['start'] and geneA['start'] < geneB['end']...
if debt['capital']['balance'] == 0.0...
alex['job_status'] = statuses['hired']
if circle_A['radius'] > circle_B['radius']
```

Παρατηρούμε πόσο πιο κοντά στην ανθρώπινη αντίληψη είναι το πρώτο σετ εντολών.

Αν έχετε μπερδευτεί είναι φυσιολογικό. Ας συνοψίσουμε λίγο την ορολογία:

- **Κλάση** (class): Ένας τύπος δεδομένων κατάλληλος για συγκεκριμένες έννοιες (άνθρωπος, εταιρία, γονίδιο, ασθένεια)
- **Αντικείμενο** (object): Μία μεταβλητή που ο τύπος της είναι μία κλάση. (σε αντιστοιχία με τα παραπάνω: Μίτσος, public, TPMT, Δαλτονισμός)
- **Ιδιότητα** (attribute): Μία ιδιότητα της κλάσσης (σε αντιστοιχία με τα παραπάνω: όνομα, πλήθος εργαζομένων, μήκος, γενετικός παράγοντας)
- **Μέθοδος** (method): Μία ιδιότητα που είναι συνάρτηση (σε αντιστοιχία με τα παραπάνω: περπατάει, προσλαμβάνει, εκφράζεται, αντιμετωπίζεται)

Ας επιστρέψουμε στη κλάση που έχουμε φτιάξει:

In [36]:
```
class Human:
    # Η Συνάρτηση ορίζεται μέσα στη κλάση
    def is_adult(self,): # Το πρώτο όρισμα είναι το self
        return self.age >= 18
```

Αν φτιάξουμε έναν άνθρωπο και δεν του δώσουμε ηλικία, προφανώς δεν μπορούμε να τρέξουμε την is_adult:

In [37]:
```
kostas = Human()
kostas.name="Κώτσος"
kostas.is_adult()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-37-efae3772950a> in <module>()
      1 kostas = Human()
      2 kostas.name="Κώτσος"
----> 3 kostas.is_adult()

<ipython-input-36-3b5fc2a2b980> in is_adult(self)
      2     # Η Συνάρτηση ορίζεται μέσα στη κλάση
      3     def is_adult(self,): # Το πρώτο όρισμα είναι το self
----> 4         return self.age >= 18

AttributeError: 'Human' object has no attribute 'age'
```

Πως μπορούμε να υποχρεώσουμε όταν δηλώνουμε ένα αντικείμενο τύπου Human, να δηλώνουμε και το age; Αυτό μπορεί να γίνε με την μέθοδο `__init__()` :

In [38]:
```
class Human:
    def __init__(self, age):
        self.age = age
    # Η Συνάρτηση ορίζεται μέσα στη κλάση
    def is_adult(self,): # Το πρώτο όρισμα είναι το self
        return self.age >= 18
```

Τι είναι αυτό το:

```
self.age=age
```

Εδώ όταν αρχικοποιούμε ένα αντικείμενο, δημιουργούμε μία ιδιότητα (self.age) και την αρχικοποιούμε με την τιμη της παραμετρου age της `__init__` :

In [39]:
```
alex = Human()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-12a27f13d5ce> in <module>()
----> 1 alex = Human()

TypeError: __init__() missing 1 required positional argument: 'age'
```

Είμαστε υποχρεωμένοι να δηλώσουμε age!

In [40]:
```
alex = Human(age=40)
```

In [41]:
```
alex.is_adult()
```

Out[41]:
```
True
```

Οι κλάσεις έχουν κάποιες "ειδικές" μεθόδους οι οποίες καλούνται μέσα από της built-in συναρτήσεις (`print` , `len` , ...) της python. Μία από αυτές είναι η `___str___` . Αυτή η συνάρτηση καλείται όποτε χρειαζόμαστε μια αναπαράσταση της κλάσης σε string (συνήθως μέσω της `print`):

In [42]:
```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
    def __str__(self):
        return '-->{} {}<--'.format(self.name, self.surname)

mitsos = Person("Δημήτρης", "Τραμπάκουλας")
print (mitsos)
```

```
-->Δημήτρης Τραμπάκουλας<--
```

Μία άλλη μέθοδος είναι η `__len__` η οποία καλείται όταν εφαρμόζουμε στο αντικέμενο μας την `len()` :

In [43]:
```
class Gene:
    def __init__(self, name, start, stop):
        self.name = name
        self.start = start
        self.stop = stop

    def __len__(self,):
        return self.stop-self.start

tpmt = Gene('TPMT', 150, 200)
len(tpmt) # 200-150
```

Out[43]:
```
50
```

Μπορούμε να δούμε όλες τις "ειδικές" μεθόδους που έχει μία κλάση:

In [44]:
```
dir(Gene)
```

Out[44]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__len__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__']
```

Ένα ολοκληρωμένο παράδειγμα

Αρκετά με τα λόγια. Ας κάνουμε τη παρακάτω άσκηση αποκλειστικά με κλάσεις.

Σε αυτό το link (https://s3.eu-central-1.amazonaws.com/kantalepythonlessons/data.txt) υπάρχει ένα αρχείο το οποίο περιέχει βασικές πληροφορίες για 1000 γονίδια. Πρέπει να κατεβάσετε το αρχείο αυτό και να το αποθηκεύσετε στον υπολογιστή σας με το όνομα `data.txt` . Ας το κάνουμε:

In [46]:
```
!wget -O data.txt https://s3.eu-central-1.amazonaws.com/kantalepythonlessons/da
ta.txt
```

```
--2019-11-22 10:52:28--  https://s3.eu-central-1.amazonaws.com/kantalepythonle
ssons/data.txt
Resolving s3.eu-central-1.amazonaws.com (s3.eu-central-1.amazonaws.com)... 52.
219.74.139
Connecting to s3.eu-central-1.amazonaws.com (s3.eu-central-1.amazonaws.com)|5
2.219.74.139|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 35335 (35K) [text/plain]
Saving to: 'data.txt'

data.txt            100%[===================>]  34.51K  --.-KB/s    in 0.05s

2019-11-22 10:52:29 (677 KB/s) - 'data.txt' saved [35335/35335]
```

Οι πρώτες 10 γραμμές του αρχείου είναι:

In [47]:
```
!head data.txt
```

```
Gene Symbol;Entrez ID;Chr;Start;Stop;Strand
DCBLD1;285761;6;117803803;117891021;+
MEIS1;4211;2;66662257;66799891;+
MED27;9442;9;134735497;134955274;-
STAU1;6780;20;47729876;47804907;-
IPO9;55705;1;201798288;201853422;+
DNAJC4;3338;11;63997594;64001753;+
DUSP10;11221;1;221874764;221916204;-
GABRP;2568;5;170210723;170241051;+
LIN28B;389421;6;105384874;105531207;+
```

Η πρώτη γραμμή του αρχείου είναι η επικεφαλίδα (header). Κάθε μία από τις υπόλοιπες 1000 γραμμές περιέχουν πληροφορίες για ένα γονίδιο. Το αρχείο περιέχει 6 στήλες. Οι στήλες διαχωρίζονται με τον χαρακτήρα: `;` (ελληνικό ερωτηματικό). Αυτές οι στήλες είναι:

- Στήλη: `Gene Symbol` περιέχει το όνομα του γονιδίου.
- Στήλη: `Entrez ID` περιέχει έναν μοναδικό κωδικό για αυτό το γονίδιο (μπορούμε να το αγνοήσουμε αυτό)
- Στήλη: `Chr` περέχει σε ποιο χρωμόσωμα ανοίκει το γονίδιο (από 1 μέχρι 22)
- Στήλη: `Start` περιέχει τη θέση στο χρωμόσωμα που αρχίζει το γονίδιο (ή αλλιώς start position).
- Στήλη: `Stop` περιέχει τη θέση στο χρωμόσωμα που τελειώνει το γονίδιο (ή αλλιώς end position).
- Στήλη: `Strand` περιέχει αν το γονίδιο βρίσκεται στο positive (+) ή negative strand (-). Δεν χρειάζεται να ψάξετε τι ακριβώς σημαίνει αυτό για να λύσετε τις ασκήσεις!

Απαντήστε με python στις παρακάτω ερωτήσεις:

- Ερ 1: Ποιο χρωμόσωμα έχει τα περισσότερα γονίδια;
- Ερ 2: Ποιο είναι το μεγαλύτερο γονίδιο; (Το μήκος του γονιδίου είναι Stop-Start)
- Ερ 3: Τυπώστε όλα τα γονίδια που ανοίκουν στον χρωμόσωμα 6 ταξινομημένα ανάλογα με τη θέση που αρχίζουν.
- Ερ 4: Φτιάξτε μια συνάρτηση που θα παίρνει το όνομα ενός γονιδίου και θα βρίσκει ποιο γονίδιο είναι ποιο κοντά. Η απόσταση μεταξύ 2 γονιδίων (για αυτή την άσκηση) ορίζεται μόνο για γονίδια που ανοίκουν στο ίδιο χρωμόσωμα και είναι η απόλυτη διαφορά του start position τους.
- Ερ 5: Πόσος είναι ο μέσος όρος του μήκους των γονιδίων που ανοίκουν σε ζυγό χρωμόσωμα (2,4,6,...) και βρίσκονται στο negative strand position;
- Ερ 6: Υπάρχουν κάποια γονίδια με το ίδιο `Entrez ID` ;
- **Προαιρετική**: Υπάρχουν γονίδια με Start position ανάμεσα στα Start και End position άλλων γονιδίων (που να ανοίκουν στο ίδιο χρωμόσωμα); Αν ναι, ποια ζευγάρια γονιδίων είναι αυτά;

Σημείωση: Τα δεδομένα προέρχονται από το paper:

> Hammerschlag, Anke R., et al. \"Genome-wide association analysis of insomnia complaints identifies risk genes and genetic overlap with psychiatric and metabolic traits.\" Nature Genetics (2017). https://www.nature.com/articles/ng.3888 (https://www.nature.com/articles/ng.3888) . Supplementary Table 6 (https://images.nature.com/original/nature-assets/ng/journal/v49/n11/extref/ng.3888-S3.xlsx)"

In [52]:
```
class CSV:
    '''
    Αυτή η κλάση διαχειρίζεται csv αρχεία.
    '''
    def __init__(self, filename, sep=',', header=True, preprocess_fields=None):
        '''
        sep: separator
        header: Does it have header?
        preprocess_fields: Optional apply a function to the data of a column.
                    It is a dictionary
        '''
        self.filename = filename
        self.sep = sep
        self.header_exists = header
        self.preprocess_fields = preprocess_fields

        #Parse the file
        self.parse_file()

    def parse_file(self,):
        '''
        parses the CSV file
        '''
        with open(self.filename) as f:
            data = f.read()

        data = data.split('\n')
        data = [x.split(self.sep) for x in data if x]

        if self.header_exists:
            self.header = data[0]
            self.data = data[1:]
        else:
            self.header = None # Δεν έχουμε header
            self.data = data

        if (self.preprocess_fields):
            self.preprocess()

    def preprocess(self,):
        '''
        Preprocesses the data
        '''

        # We need the header to preprocess
        if self.header is None:
            raise Exception('Dataset needs to have a header to preprocess')

        for key, preprocess_function in self.preprocess_fields.items():
            if not key in self.header:
                continue

            index = self.header.index(key)
            for data_row in self.data:
                data_row[index] = preprocess_function(data_row[index])

    def dict_iterator(self,):
        '''
        Returns an iterator
        Each item that it yields is a dictionary
        '''

        if self.header is None:
            raise Exception('Dataset needs to have a header in order to create
an iterator')

        for d in self.data:
            yield dict(zip(self.header, d))
```

In [53]:
```
# Er.1. Ποιο χρωμόσωμα έχει τα περισσότερα γονίδια;
my_list.count_chromosomes()
max(my_list.chromosome_counts)
```

Out[53]:
```
(107, '11')
```

In [54]:
```
#Er.2. Ποιο είναι το μεγαλύτερο γονίδιο; (Το μήκος του γονιδίου είναι Stop-Start)
print(max(my_list.genes, key=len))
```

```
DPP10 57628 chr:2:115199899-116603328/+
```

In [55]:
```
#Ερ. 3. Τυπώστε όλα τα γονίδια που ανοίκουν στον χρωμόσωμα 6 ταξινομημένα ανάλο
γα με τη θέση που αρχίζου
print (my_list.filter(lambda gene:gene.chromosome=='6').sorted(key=lambda gene:
gene.start))
```

```
MYLK4 340156 chr:6:2663863-2765615/-
C6orf201 404220 chr:6:4079440-4131000/+
ECI2 10455 chr:6:4115927-4135831/-
DTNBP1 84062 chr:6:15523032-15663289/-
HIST1H4C 8364 chr:6:26104176-26104565/+
HIST1H2BL 8340 chr:6:27775257-27775709/-
HIST1H2AI 8329 chr:6:27775977-27776445/+
HIST1H2AJ 8331 chr:6:27782080-27782518/-
HIST1H2BM 8342 chr:6:27782822-27783267/+
HIST1H2AK 8330 chr:6:27805658-27806117/-
HIST1H1B 3009 chr:6:27834570-27835359/-
MCCD1 401250 chr:6:31496739-31498008/+
DDX39B 7919 chr:6:31497996-31510252/-
LY6G5C 80741 chr:6:31644461-31648150/-
BAK1 578 chr:6:33540323-33548072/-
ITPR3 3710 chr:6:33587951-33664351/+
IP6K3 117283 chr:6:33689415-33714762/-
LEMD2 221496 chr:6:33738990-33756906/-
MLN 4295 chr:6:33762449-33771793/-
PI16 221476 chr:6:36916039-36932613/+
RRP36 88745 chr:6:42989385-42997337/+
PTK7 5754 chr:6:43044006-43129458/+
SRF 6722 chr:6:43138920-43149244/+
CUL9 23113 chr:6:43149913-43192325/+
DNPH1 10591 chr:6:43193367-43197211/-
VEGFA 7422 chr:6:43737946-43754224/+
TMEM14A 28978 chr:6:52535884-52551386/+
GSTA2 2939 chr:6:52614885-52628361/-
DST 667 chr:6:56322785-56819426/-
KHDC1 80759 chr:6:73951037-74019938/-
DPPA5 340168 chr:6:74062785-74063999/-
OOEP 441161 chr:6:74078278-74079515/-
MB21D1 115004 chr:6:74134856-74162043/-
SLC17A5 26503 chr:6:74303102-74363737/-
LOC101928870 101928870 chr:6:87861525-87865307/+
GJB7 375519 chr:6:87992696-88038996/-
SMIM8 57150 chr:6:88032306-88052043/+
C6orf163 206412 chr:6:88054571-88075381/+
HACE1 57531 chr:6:105175968-105307794/-
LIN28B 389421 chr:6:105384874-105531207/+
DCBLD1 285761 chr:6:117803803-117891021/+
GOPC 57120 chr:6:117881432-117923705/-
NKAIN2 154215 chr:6:124124991-125146786/+
BCLAF1 9774 chr:6:136578001-136611783/-
MAP7 9053 chr:6:136663419-136871957/-
IL20RA 53832 chr:6:137321108-137366317/-
IFNGR1 3459 chr:6:137518621-137540981/-
AKAP12 9590 chr:6:151561134-151679694/+
ZBTB2 57621 chr:6:151685250-151712677/-
RGS17 26575 chr:6:153332026-153452389/-
ARID1B 57492 chr:6:157098980-157531913/+
GTF2H5 404672 chr:6:158589379-158620376/+
PLG 5340 chr:6:161123225-161175086/+
RPS6KA2 6196 chr:6:166822854-167275948/-
PSMB1 5689 chr:6:170844204-170862417/-
TBP 6908 chr:6:170863384-170881958/+
PDCD2 5134 chr:6:170884660-170893780/-
```

In [56]:
```
# Ερ. 4 Φτιάξτε μια συνάρτηση που θα παίρνει το όνομα ενός γονιδίου και θα βρίσ
κει ποιο γονίδιο είναι ποιο κοντά.
# Η απόσταση μεταξύ 2 γονιδίων (για αυτή την άσκηση) ορίζεται μόνο για γονίδια
που ανοίκουν στο
# ίδιο χρωμόσωμα και είναι η απόλυτη διαφορά του start position τους.

my_list.closer(name='PDCD2')
```

Out[56]:
```
(21276, 'TBP 6908 chr:6:170863384-170881958/+')
```

In [57]:
```
# Ερ. 5. Πόσος είναι ο μέσος όρος του μήκους των γονιδίων που ανοίκουν σε ζυγό
χρωμόσωμα (2,4,6,...)
# και βρίσκονται στο negative strand position;

my_list.filter(lambda x:int(x.chromosome)%2==0 and x.strand=='-').average_lengt
h()
```

Out[57]:
```
83351.24867724867
```

In [58]:
```
# Ερ. 5 (β τρόπος)
my_list.filter(lambda x:int(x.chromosome)%2==0).filter(lambda x:x.strand=='-').
average_length()
```

Out[58]:
```
83351.24867724867
```

In [59]:
```
#Ερ 6: Υπάρχουν κάποια γονίδια με το ίδιο Entrez ID;
my_list.check_entrez()
```

Out[59]:
```
True
```

In [62]:
```
my_list.in_between(my_list)[:10] #  Τυπώνω μόνο τα πρώτα 10
```

```
THIS GENE:
   GOPC 57120 chr:6:117881432-117923705/-
IS IN BETWEEN THIS GENE:
   DCBLD1 285761 chr:6:117803803-117891021/+

THIS GENE:
   POLG 5428 chr:15:89859536-89878026/-
IS IN BETWEEN THIS GENE:
   FANCI 55215 chr:15:89785634-89860362/+

THIS GENE:
   LOC101929519 101929519 chr:9:96433303-96435032/-
IS IN BETWEEN THIS GENE:
   PHF2 5253 chr:9:96338909-96441869/+

THIS GENE:
   NMT1 4836 chr:17:43138322-43186384/+
IS IN BETWEEN THIS GENE:
   DCAKD 79877 chr:17:43100706-43138473/-

THIS GENE:
   FLRT1 23769 chr:11:63803442-63886655/+
IS IN BETWEEN THIS GENE:
   MACROD1 28992 chr:11:63766006-63933602/-

THIS GENE:
   CAMK2N2 94032 chr:3:183977003-183979251/-
IS IN BETWEEN THIS GENE:
   ECE2 9718 chr:3:183967445-184010819/+

THIS GENE:
   KRT81 3887 chr:12:52679697-52685299/-
IS IN BETWEEN THIS GENE:
   KRT86 3892 chr:12:52642892-52702947/+

THIS GENE:
   ERN2 10595 chr:16:23701625-23724821/-
IS IN BETWEEN THIS GENE:
   PLK1 5347 chr:16:23690201-23701688/+

THIS GENE:
   TRPT1 83707 chr:11:63991271-63993726/-
IS IN BETWEEN THIS GENE:
   FERMT3 83706 chr:11:63974152-63991363/+

THIS GENE:
   TIMP4 7079 chr:3:12194568-12200851/-
IS IN BETWEEN THIS GENE:
   SYN2 6854 chr:3:12182150-12233532/+

THIS GENE:
   TGS1 96764 chr:8:56685791-56738007/+
IS IN BETWEEN THIS GENE:
   TMEM68 137695 chr:8:56651317-56685955/-

THIS GENE:
   LURAP1 541468 chr:1:46669006-46686928/+
IS IN BETWEEN THIS GENE:
   POMGNT1 55624 chr:1:46654353-46685977/-

THIS GENE:
   LIMD2 80774 chr:17:61773249-61778527/-
IS IN BETWEEN THIS GENE:
   MAP3K3 4215 chr:17:61699404-61773670/+

THIS GENE:
   C14orf178 283579 chr:14:78227173-78236085/+
IS IN BETWEEN THIS GENE:
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-62-71496bbbd8f2> in <module>()
----> 1 my_list.in_between(my_list)[:10] #  Τυπώνω μόνο τα πρώτα 10

TypeError: 'NoneType' object is not subscriptable
```

Παρατηρούμε ότι η υλοποίηση με κλάσεις είναι πολύ πιο μεγάλη, δηλαδή έχει πολύ περισσότερες γραμμές κώδικα. Παρόλα αυτά ο κώδικας είναι καλύτερα οργανομένος και οι εντολές που απαντούν στις ερωτήσεις της άσκησεις είναι πιο κατανοητές (π.χ. `my_list.closer(name='PDCD2')`).

Οι κλάσεις έχουν τα θετικά και τα αρνητικά (https://en.wikipedia.org/wiki/Object-oriented_programming#Criticism) τους. Ένας κλασσικός κανόνας που εφαρμόζουμε είναι: Υλοποιούμε το πρόγραμμά μας σε κλάσεις όταν υπάρχει ένας σαφής εννοιολογικός διαχωρισμός των οντοτήτων που θέλουμε να διαχειριστούμε (π.χ. πρόσωπα, γονίδια, εταιρία, βάση δεδομένων, κτλ.)

Κλάσεις (μέρος 2ο)

Είχαμε πει ότι μπορεί μία κλάση να αποκτήσει "μέγεθος" (length) αν υλοποιήσει τη `__len__` μέθοδο:

In [72]:
```
class Gene:
    def __len__(self,):
        return 50
```

In [74]:
```
b = Gene()
print (len(b))
```

```
50
```

Άλλωστε όταν εφαρμόζουμε τη `len` σε μία λίστα στη πραγματικότητα εκτελείται η `__len__()` :

In [76]:
```
a = [1,2,5]
print (a.__len__())
```

```
3
```

Ας φτιάξουμε μία κλάση που αντιπροσωπεύει έναν άνθρωπο:

In [176]:
```
class Human:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Και ας φτιάξουμε και ένα αντικέμενο:

In [177]:
```
mitsos = Human('Μήτσος', 50)
```

In [178]:
```
mitsos.name
```

Out[178]:
```
'Μήτσος'
```

Όπως έχουμε δει το name και το age είναι "πεδία" της κλάσης Human. Πολλές φορές χρειάζεται μία κλάση να έχει κάποιες μεταβλητές (συνήθως σταθερές τιμές) η οποία να είναι ίδιες για ΟΛΑ τα αντικείμενα αυτής της κλασής. Να πως γίνεται αυτό:

In [181]:
```
class Human:
    max_age = 100

    def __init__(self, name, age):
        self.name = name
        self.age = age

mitsos = Human('Μήτσος', 50)
kwstas = Human('Κώστας', 10)
```

In [182]:
```
print (mitsos.max_age)
```

```
100
```

Παρατηρήστε οτι το `max_age` δεν το έχουμε θέσει ούτε στον `mitsos` ούτε στο `kwstas` . Παρόλα αυτά αν κάνουμε `print (mitsos.max_age)` βγάζει τη τιμή που έχουμε θέσει στη `Human` . Αυτό γίνεται γιατί το max_age υπάρχει σε ΟΛΑ τα αντικείμενα της κλάσης `Human` . Επίσης αν αλλάξουμε τη τιμή αυτού του πεδίου στη Human τότε θα αλλάξει σε όλα τα αντικείμενα αυτής της κλάσης:

In [183]:
```
Human.max_age = 200
print (mitsos.max_age)
```

```
200
```

Ορισμός: οι ιδιότητες (πεδία ή μέθοδοι) μίας κλάσης που είναι προσπελάσιμες από την κλάση χωρίς να χρειάζεται η δημιουργία ενός αντικειμένου τους ονομάζονται static

Μπορεί και μία μέθοδος να είναι static:

In [186]:
```
class Human:
    max_age = 100

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def is_adult(age):
        return age>18

Human.is_adult(30)
```

Out[186]:
```
True
```

Το μειονέκτημα της παραπάνω υλοποίησης είναι ότι ΔΕΝ μπορούμε να τρέξουμε την `is_adult` από ένα αντικείμενο:

In [189]:
```
mitsos = Human('Μήτσος', 50)
mitsos.is_adult(100)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-189-a0e45862f27e> in <module>()
      1 mitsos = Human('Μήτσος', 50)
----> 2 mitsos.is_adult(100)

TypeError: is_adult() takes 1 positional argument but 2 were given
```

Για να μπορεί να τρέξει μία μέθοδος και από τη κλάση αλλά και από το αντικείμενο πρέπει να χρησιμοποιήσουμε το `@staticmethod` :

In [190]:
```
class Human:
    max_age = 100

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @staticmethod
    def is_adult(age):
        return age>18
```

In [192]:
```
print (Human.is_adult(60))
mitsos = Human('Μήτσος', 50)
print (mitsos.is_adult(60))
```

```
True
True
```

Μία καλύτερη υλοποίηση:

In [199]:
```
class Human:
    max_age = 100
    adult_age = 18

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def is_adult(self,):
        return Human.is_adult_2(self.age)

    @staticmethod
    def is_adult_2(age):
        return age >= Human.adult_age



mitsos = Human('Μήτσος', 50)
print (mitsos.is_adult()) # True
kwstas = Human('Κώστας', 10)
print (kwstas.is_adult()) # False
print (Human.is_adult_2(20)) # True
```

```
True
False
True
```

Μα τι είναι αυτό το `@staticmethod` ;; Αυτό ονομάζεται decorator (https://www.learnpython.org/en/Decorators) και θα το αναλύσουμε άλλη φορά.

Κλάσεις (Μέρος 3ο)

Μία κλάση μπορεί να "κληρονομήσει" μία άλλη κλάση: https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming (https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))

Όταν γίνεται αυτό, τότε η νέα κλάση περιέχει όλες τις ιδιότητες (πεδία + μέθοδοι) της παλιάς. Κλασσικά παραδείγματα είναι:

- η κλάση φορτηγό έχει κληρονομήσει τη κλάση όχημα
- η κλάση DNA και η κλάση RNA έχει κληρονομήσει τη κλάση sequence
- η κλάση Employee έχει κληρονιμήσει τη κλάση Human

Στη python αυτό γίνεται ως εξής:

In [201]:
```
class Employee(Human):
    pass

manolis = Employee('Μανόλης', 40)
```

Η Employee περιέχει όλες τις μεθόδους της Human:

In [202]:
```
manolis.is_adult()
```

Out[202]:
```
True
```

Ας βάλουμε μία νέα ιδιότητα στη Employee

In [206]:
```
class Employee(Human):
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

        if not self.is_adult():
            raise Exception('THIS IS ILLEGAL!')

manolis = Employee('Μανόλης', 40, 10000)
```

In [208]:
```
john = Employee('Γιάννης', 17, 10000) # παιδική εργασία!
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-208-f7869cbe3fb9> in <module>()
----> 1 john = Employee('Γιάννης', 17, 10000) # παιδική εργασία!

<ipython-input-206-70c56380c507> in __init__(self, name, age, salary)
      6
      7         if not self.is_adult():
----> 8             raise Exception('THIS IS ILLEGAL!')
      9
     10 manolis = Employee('Μανόλης', 40, 10000)

Exception: THIS IS ILLEGAL!
```

Παρατηρείστε ότι το κομμάτι

```
    self.name = name
            self.age = age
```

Το έχουμε στη Human αλλά και στην Employee. Δεν μπορούμε να το αποφύγουμε αυτό; Γίνεται με την εντολή `super` η οποία καλεί την parent class.

In [210]:
```
class Employee(Human):
    def __init__(self, name, age, salary):
        super().__init__(name, age)
        self.salary = salary

        if not self.is_adult():
            raise Exception('THIS IS ILLEGAL!')

manolis = Employee('Μανόλης', 40, 10000)
```

Η `super().__init__()` καλεί `__init__()` της parent class.

Πολλαπλή κληρονομικότητα

Μία κλάση μπορεί να κληρονομήσει περισσότερες από μία κλάσεις. π.χ:

In [226]:
```
class Resident:
    def __init__(self, address):
        self.address = address

    def show_address(self):
        print (self.address)
```

In [217]:
```
class Employee(Human, Resident):
    pass
```

Εδώ πρέπει να τονίσουμε κάτι πολύ σημαντικό: Η σειρά με την οποία δηλώνουμε τις parent classes είναι πολύ σημαντική. Η νέα κλάση κληρονομεί τον constructor (`__init__()`) ΜΟΝΟ της πρώτης κλάσης:

In [219]:
```
mitsos = Employee('Μήτσος', 50)
```

Εδώ παρατηρούμε ότι ο constructor της Resident δεν καλέστηκε!

In [221]:
```
mitsos.show_address()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-221-05b17d0e9a10> in <module>()
----> 1 mitsos.show_address()

<ipython-input-212-260abcb16edb> in show_address(self)
      4
      5     def show_address(self):
----> 6         print (self.address)

AttributeError: 'Employee' object has no attribute 'address'
```

Αυτό μπορούμε να το διορθώσουμε καλώντας τους constructors με όποια σειρά θέλουμε από την `__init__` της νέας κλάσης:

In [229]:
```
class Employee(Human, Resident):
    def __init__(self, name, age, address, salary):
        Human.__init__(self, name, age)
        Resident.__init__(self, address)
        self.salary = salary

mitsos = Employee('Μήτσος', 50, "Ηράκλειο", 10000)
print (mitsos.is_adult())
mitsos.show_address()
```

```
True
Ηράκλειο
```

Μπορούμε να βάλουμε αντικείμενα μέσα σε λίστες dictionaries..

In [231]:
```
stuff = {
    0: Employee('Kostas', 40, 'Ηράκλειο', 1000),
    1: Employee('Andreas', 40, 'Πάτρα', 100),
}
```

Τι γίνεται όταν θέλουμε να σώσουμε μία λίστα/dictionary που έχει αντικείμενα σε ένα αρχείο; Δεν μπορούμε να τα μετατρέψουμε άμεσα σε string:

In [232]:
```
str(stuff)
```

Out[232]:
```
'{0: <__main__.Employee object at 0x1106bd5f8>, 1: <__main__.Employee object a
t 0x1106bd550>}'
```

Ούτε μπορούμε να τα κάνουμε json:

In [234]:
```
import json
json.dumps(stuff)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-234-54caab5df3b5> in <module>()
      1 import json
----> 2 json.dumps(stuff)

~/anaconda3/envs/arkalos/lib/python3.6/json/__init__.py in dumps(obj, skipkey
s, ensure_ascii, check_circular, allow_nan, cls, indent, separators, default,
sort_keys, **kw)
    229         cls is None and indent is None and separators is None and
    230         default is None and not sort_keys and not kw):
--> 231         return _default_encoder.encode(obj)
    232     if cls is None:
    233         cls = JSONEncoder

~/anaconda3/envs/arkalos/lib/python3.6/json/encoder.py in encode(self, o)
    197         # exceptions aren't as detailed.  The list call should be roug
hly
    198         # equivalent to the PySequence_Fast that ''.join() would do.
--> 199         chunks = self.iterencode(o, _one_shot=True)
    200         if not isinstance(chunks, (list, tuple)):
    201             chunks = list(chunks)

~/anaconda3/envs/arkalos/lib/python3.6/json/encoder.py in iterencode(self, o,
_one_shot)
    255                 self.key_separator, self.item_separator, self.sort_key
s,
    256                 self.skipkeys, _one_shot)
--> 257         return _iterencode(o, 0)
    258
    259 def _make_iterencode(markers, _default, _encoder, _indent, _floatstr,

~/anaconda3/envs/arkalos/lib/python3.6/json/encoder.py in default(self, o)
    178         """
    179         raise TypeError("Object of type '%s' is not JSON serializable"
%
--> 180                         o.__class__.__name__)
    181
    182     def encode(self, o):

TypeError: Object of type 'Employee' is not JSON serializable
```

Για αυτό το σκοπό μπορούμε να χρησιμοποιήσουμε τη βιβλιοθήκη pickle (https://docs.python.org/3.1/library/pickle.html):

In [235]:
```
import pickle
stuff_serialized = pickle.dumps(stuff)
```

In [238]:
```
stuff_serialized
```

Out[238]:
```
b'\x80\x03}q\x00(K\x00c__main__\nEmployee\nq\x01)\x81q\x02}q\x03(X\x04\x00\x0
0\x00nameq\x04X\x06\x00\x00\x00Kostasq\x05X\x03\x00\x00\x00ageq\x06K(X\x07\x0
0\x00\x00addressq\x07X\x10\x00\x00\x00\xce\x97\xcf\x81\xce\xac\xce\xba\xce\xb
b\xce\xb5\xce\xb9\xce\xbfq\x08X\x06\x00\x00\x00salaryq\tM\xe8\x03ubK\x01h\x0
1)\x81q\n}q\x0b(h\x04X\x07\x00\x00\x00Andreasq\x0ch\x06K(h\x07X\n\x00\x00\x00\
xce\xa0\xce\xac\xcf\x84\xcf\x81\xce\xb1q\rh\tKdubu.'
```

In [239]:
```
type(stuff_serialized)
```

Out[239]:
```
bytes
```

Μπορούμε να φορτώσουμε pickle δεδομένα:

In [237]:
```
a = pickle.loads(stuff_serialized)
print (a)
```

```
{0: <__main__.Employee object at 0x110665470>, 1: <__main__.Employee object at
0x110665e80>}
```

Επιπλέον σημειώσεις

Σας προτρέπω να διαβάσετε τις έξοχες σημειώσεις (https://gist.github.com/spirosChv/6f22d6dc5cf79eec78069ddf36749fa4) από τον Σπύρο Χαυλή schavlis@imbb.forth.gr (mailto:schavlis@imbb.forth.gr) που είχε ετοιμάσει για το μάθημα του 2016 σχετικά με τις κλάσεις και τον αντικειμενικοστραφή προγραμματισμό.