# 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 [1]:
```
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 [2]:
```
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 [3]:
```
type(alex)
```

Out[3]:
```
__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 [4]:
```
alex.name = "Αλέξανδρος"
alex.age = 45
```

In [5]:
```
print (alex.name, alex.age)
```

```
Αλέξανδρος 45
```

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 [7]:
```
def is_adult(human):
    return human.age >= 18

is_adult(alex)
```

Out[7]:
```
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 προϋποθέσεις συνοψίζονται στο παρακάτω παράδειγμα:

In [8]:
```
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 [9]:
```
alex = Human()
alex.age = 40
alex.is_adult()
```

Out[9]:
```
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*. 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.:

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 [10]:
```
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 [11]:
```
kostas = Human()
kostas.name="Κώτσος"
kostas.is_adult()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-11-efae3772950a> in <module>
      1 kostas = Human()
      2 kostas.name="Κώτσος"
----> 3 kostas.is_adult()

<ipython-input-10-3b5fc2a2b980> 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 [12]:
```
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 [14]:
```
alex = Human()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-12a27f13d5ce> in <module>
----> 1 alex = Human()

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

Είμαστε υποχρεωμένοι να δηλώσουμε age!

In [15]:
```
alex = Human(age=45)
```

In [16]:
```
alex.is_adult()
```

Out[16]:
```
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 [17]:
```
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 [18]:
```
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[18]: 50

Μία άλλη μέθοδος είναι η `__getitem__` η οποία μας επιτρέπει να "πάρουμε" ένα στοιχείο από μία συλλογή μέσω του τελεστή: `[]` :

In [19]:
```
class Gene:
    def __init__(self, name, start, stop):
        self.name = name
        self.start = start
        self.stop = stop

    def __getitem__(self, i):
        ret = self.start + i
        if ret > self.stop:
            raise IndexError
        return ret
```

In [20]:
```
tpmt = Gene('TPMT', 150, 200)
tpmt[23]
```

Out[20]: 173

In [21]:
```
tpmt[60]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-21-39a580f61a36> in <module>
----> 1 tpmt[60]

<ipython-input-19-17b2584805cd> in __getitem__(self, i)
      8         ret = self.start + i
      9         if ret > self.stop:
---> 10             raise IndexError
     11         return ret
     12

IndexError:
```

Δύο άλλες μέθοδοι είναι οι `__iter__` και η `__next__` . Υλοποιώντας αυτές τις μεθόδους μπορούμε να κάνουμε iteration σε ένα αντικείμενο:

In [24]:
```
class Gene:
    def __init__(self, name, start, stop):
        self.name = name
        self.start = start
        self.stop = stop

    def __iter__(self,):
        self.i = self.start
        return self

    def __next__(self,):
        if self.i == self.stop:
            raise StopIteration

        ret = self.i
        self.i += 1
        return ret
```

In [25]:
```
tpmt = Gene('TPMT', 150, 200)
for x in tpmt:
    print (x)
```

```
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
```

Μπορούμε ακόμα και να αλλάξουμε τη συμπεριφορά των πράξεων, υλοποιώντας τις ανάλογες μεθόδους. Για παράδειγμα μπορούμε να "υποστηρίξουμε" τη πράξη `+` στη κλάση Gene:

In [35]:
```
class Gene:
    def __init__(self, name, start, stop):
        self.name = name
        self.start = start
        self.stop = stop

    def __add__(self, g):
        return Gene(
            name = f'{self.name}+{g.name}',
            start=self.start,
            stop = self.stop + g.stop-g.start, # Απλά προσθέτουμε το μήκος
        )

    def __str__(self,):
        return f'Gene: {self.name} Start: {self.start} End:{self.stop}'
```

In [36]:
```
tpmt = Gene('TPMT', 150, 200)
apoe = Gene('APOE', 400, 470)

new_gene = tpmt + apoe
print (new_gene)
```

```
Gene: TPMT+APOE Start: 150 End:270
```

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

In [37]:
```
dir(Gene)
```

Out[37]:
```
['__add__',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
```

```
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__']
```

# Κλάσεις (μέρος 2ο)

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

In [49]:
```
class Gene:
    def __len__(self,):
        return 50
```

In [50]:
```
b = Gene()
print (len(b))
```

```
50
```

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

In [51]:
```
a = [1,2,5]
print (a.__len__())
```

```
3
```

Ας φτιάξουμε μία κλάση που αντιπροσωπεύει έναν άνθρωπο:

In [52]:
```
class Human:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Και ας φτιάξουμε και ένα αντικέμενο:

In [53]:
```
mitsos = Human('Μήτσος', 50)
```

In [54]:
```
mitsos.name
```

Out[54]:
```
'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 [55]:
```
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 [56]:
```
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 [57]:
```
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 [58]:
```
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[58]:
```
True
```

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

In [59]:
```
mitsos = Human('Μήτσος', 50)
mitsos.is_adult(100)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-59-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 [61]:
```
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 [62]:
```
print (Human.is_adult(60))
mitsos = Human('Μήτσος', 50)
print (mitsos.is_adult(60))
```

```
True
True
```

Σημείωση: το staticmethod είναι ένας decorator, μία ιδιότητα της python που δεν έχουμε παρουσιάσει..

Μία καλύτερη υλοποίηση:

In [63]:
```
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
```

# Κλάσεις (Μέρος 3ο)

Μία κλάση μπορεί να "κληρονομήσει" μία άλλη κλάση: 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 [64]:
```
class Employee(Human):
    pass

manolis = Employee('Μανόλης', 40)
```

Η Employee περιέχει όλες τις μεθόδους της Human:

In [65]:
```
manolis.is_adult()
```

Out[65]: True

Ας βάλουμε μία νέα ιδιότητα στη Employee

In [66]:
```
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 [67]:
```
john = Employee('Γιάννης', 15, 10000) # παιδική εργασία!
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-67-220eac5da521> in <module>
----> 1 john = Employee('Γιάννης', 15, 10000) # παιδική εργασία!

<ipython-input-66-70c56380c507> 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 [68]:
```
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

In [69]:
```
class Resident:
    def __init__(self, address):
        self.address = address

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

In [70]:
```
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 [71]:
```
mitsos = Employee('Μήτσος', 50)
```

Εδώ παρατηρούμε ότι ο constructor της Resident δεν καλέστηκε!

In [72]:
```
mitsos.show_address()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-72-05b17d0e9a10> in <module>
----> 1 mitsos.show_address()

<ipython-input-69-3eaf54002b5a> 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 [73]:
```
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 [74]:
```
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 [75]:
```
str(stuff)
```

Out[75]:
```
'{0: <__main__.Employee object at 0x7ff76d574880>, 1: <__main__.Employee ob
ject at 0x7ff76d5747c0>}'
```

Ούτε μπορούμε να τα κάνουμε json:

In [76]:
```
import json
json.dumps(stuff)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-76-54caab5df3b5> in <module>
      1 import json
----> 2 json.dumps(stuff)

~/anaconda3/lib/python3.8/json/__init__.py in dumps(obj, skipkeys, ensure_a
scii, check_circular, allow_nan, cls, indent, separators, default, sort_key
s, **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/lib/python3.8/json/encoder.py in encode(self, o)
    197         # exceptions aren't as detailed.  The list call should be r
oughly
    198         # equivalent to the PySequence_Fast that ''.join() would d
o.
--> 199         chunks = self.iterencode(o, _one_shot=True)
    200         if not isinstance(chunks, (list, tuple)):
    201             chunks = list(chunks)

~/anaconda3/lib/python3.8/json/encoder.py in iterencode(self, o, _one_shot)
    255                 self.key_separator, self.item_separator, self.sort_
keys,
    256                 self.skipkeys, _one_shot)
--> 257         return _iterencode(o, 0)
    258
    259 def _make_iterencode(markers, _default, _encoder, _indent, _floatst
r,

~/anaconda3/lib/python3.8/json/encoder.py in default(self, o)
    177
    178         """
--> 179         raise TypeError(f'Object of type {o.__class__.__name__} '
    180                         f'is not JSON serializable')
    181

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

Για αυτό το σκοπό μπορούμε να χρησιμοποιήσουμε τη βιβλιοθήκη pickle:

In [78]:
```
import pickle
stuff_serialized = pickle.dumps(stuff)
```

In [79]:
```
stuff_serialized
```

Out[79]:
```
b'\x80\x04\x95\x97\x00\x00\x00\x00\x00\x00\x00}\x94(K\x00\x8c\x08__main__\x
```

```
94\x8c\x08Employee\x94\x93\x94)\x81\x94}\x94(\x8c\x04name\x94\x8c\x06Kosta
s\x94\x8c\x03age\x94K(\x8c\x07address\x94\x8c\x10\xce\x97\xcf\x81\xce\xac\x
ce\xba\xce\xbb\xce\xb5\xce\xb9\xce\xbf\x94\x8c\x06salary\x94M\xe8\x03ubK\x0
1h\x03)\x81\x94}\x94(h\x06\x8c\x07Andreas\x94h\x08K(h\t\x8c\n\xce\xa0\xce\x
ac\xcf\x84\xcf\x81\xce\xb1\x94h\x0bKdubu.'
```

In [80]:
```
type(stuff_serialized)
```

Out[80]:
```
bytes
```

Μπορούμε να φορτώσουμε pickle δεδομένα:

In [82]:
```
a = pickle.loads(stuff_serialized)
print (a)
```

```
{0: <__main__.Employee object at 0x7ff76d59b7c0>, 1: <__main__.Employee obj
ect at 0x7ff76d59b9a0>}
```

# Επιπλέον σημειώσεις

Σας προτρέπω να διαβάσετε τις έξοχες σημειώσεις από τον Σπύρο Χαυλή schavlis@imbb.forth.gr που είχε ετοιμάσει για το μάθημα του 2016 του μεταπτυχιακού προγράμματος της ιατρικής, σχετικά με τις κλάσεις και τον αντικειμενικοστραφή προγραμματισμό.