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

HMY01K06
Επιστημονικός Προγραμματισμός με Python

Διάλεξη Δωδέκατη
Αντικειμενοστρεφής Προγραμματισμός
Κλάσεις , Αντικείμενα, Κληρονομικότητα, Πολυμορφισμός

Φθινόπωρο 2025

Μοντέλα Προγραμματισμού

Τρόποι *αντίληψης* και έκφρασης ενός πλαισίου βασικών εννοιών οι οποίες *διαμορφώνουν* τον *σχεδιασμό* και την *υλοποίηση* ενός προγράμματος

Πρώτο ιστορικά μοντέλο:

Δηλωτικός / προστακτικός *προγραμματισμός* *(imperative programming)*

Προέκυψε από την φυσική περιγραφή και εξέλιξη των πρώτων γλωσσών προγραμματισμού στις δεκαετίες '50 και '60. *(Assembly, FORTRAN…)*

A = 12

ADD …

MULTIPLY …

WRITE …

GOTO …

Απλές εντολές που είτε *δηλώνουν* κάτι *(πχ. A=12)*, είτε *προστάζουν* να γίνει κάτι *(πχ. WRITE…)*

Μοντέλα Προγραμματισμού

Στα χρόνια που ακολούθησαν αναπτύχθηκαν και άλλα μοντέλα προγραμματισμού

- *Δομημένος / διαδικαστικός προγραμματισμός (structured / procedural programming)* Προέκυψε ως *εξέλιξη* του δηλωτικού / προστακτικού και αποτελεί το πιο διαδεδομένο μοντέλο στο οποίο βασίζονται οι πιο γνωστές γλώσσες προγραμματισμού γενικής χρήσης *(Pascal, C…)*

- *Αντικειμενοστρεφής προγραμματισμός (object-oriented programming)*

- *Λογικός προγραμματισμός (logic programming)*

- *Συμβολικός προγραμματισμός (symbolic programming)*

- …

Αντικειμενοστρεφής προγραμματισμός

Object Oriented Programming (OOP)

Τι είναι;

Τυπικός ορισμός

Η αντικειμενοστρέφεια (object orientation) είναι μια *προσέγγιση* στην ανάπτυξη λογισμικού σύμφωνα με την οποία το πρόβλημα κωδικοποιείται σε μια συλλογή από *διακριτά αντικείμενα* (*objects*) τα οποία *αλληλεπιδρούν* για την επίλυσή του *(προβλήματος)*

Κάθε αντικείμενο αποτελεί *εκπρόσωπο ή στιγμιότυπο* (*instance*) μιας γενικότερης οντότητας που ονομάζεται *κλάση* (*class*)

Ένα μήλο είναι <u>*εκπρόσωπος*</u> ή <u>*στιγμιότυπο*</u> της κλάσης "Φρούτα"

Διαφορά διαδικαστικού και αντικειμενοστρεφούς προγραμματισμού

Διαδικαστικός προγραμματισμός

Έμφαση στις διαδικασίες (συναρτήσεις)

Παράδειγμα: Συνάρτηση `find()`:
Ελέγχει αν μια συμβολοσειρά περιέχεται μέσα σε μια άλλη

Επιστρέφει έναν ακέραιο με την θέση της δεύτερης συμβολοσειράς μέσα στη πρώτη

Τρόπος σύνταξης : `n = find(S1,S2)`

Τρόπος σκέψης: *(έμφαση στη διαδικασία)*

"'Ελα εδώ `find()`. Πάρε τις συμβολοσειρές `S1` και `S2` και ψάξε να βρεις αν η `S1` περιέχει την `S2`. Ξέρεις πώς να το κάνεις"

Αντικειμενοστρεφής προγραμματισμός

Έμφαση στα αντικείμενα

Παράδειγμα: Μέθοδος `find()`:
Εφαρμόζεται πάνω σε μια συμβολοσειρά για να ελέγξει αν περιέχει μια άλλη

Επιστρέφει έναν ακέραιο με την θέση της δεύτερης συμβολοσειράς μέσα στη πρώτη

Τρόπος σύνταξης : `n = S1.find(S2)`

Τρόπος σκέψης: *(έμφαση στο αντικείμενο)*

"'Ελα εδώ `S1`. Πάρε την `S2` και ψάξε να δεις αν την περιέχεις κάπου. Ξέρεις πώς να το κάνεις"

Όλα τα αντικείμενα τύπου "συμβολοσειρά" ξέρουν πως να (έχουν μια μέθοδο χειρισμού για να) βρίσκουν αν ένα τμήμα τους ταυτίζεται ή όχι με μια άλλη δοσμένη (συνήθως μικρότερη) συμβολοσειρά

Αντικειμενοστρεφής προγραμματισμός

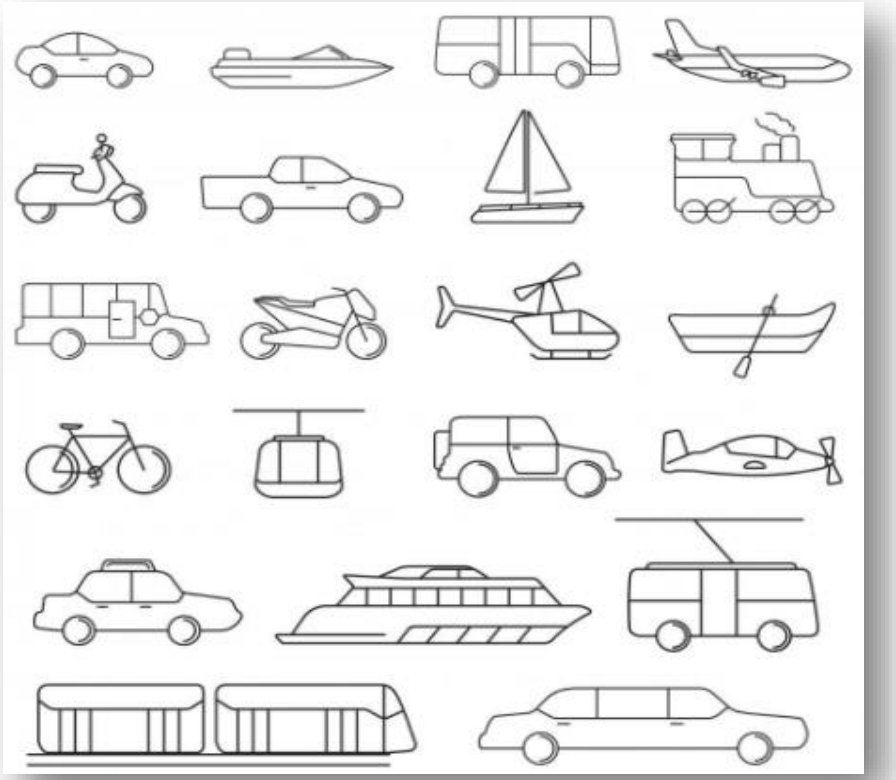
Ιστορία:

- 1967: **Simula67** (*Νορβηγία*) → Πρώτη αντικειμενοστρεφής γλώσσα
- 1970 **SmallTalk** (*Palo Alto CA*) → Κάθε στοιχείο και αντικείμενο
- 1980: **C++** (*Bjarne Stroustrup AT&T Bell Labs*) → Σοβαρή, αποδοτική γλώσσα, πρότυπο στη βιομηχανία
- 1990: **Python** (*Guido van Rossum, Ολλανδία*) Ισχυρή, ευέλικτη και εύκολη στη εκμάθηση, αποτελεί εναν "ελβετικό σουγιά" για προγραμματιστές
- 1995 **Java** (*James Gosling Sun Microsystems*) Σχεδιασμένη ώστε οι εφαρμογές της να έχουν τη λιγότερη δυνατή εξάρτηση από hardware της πλατφόρμας που εκτελούνται *(JVM – Java Virtual Machine)*

Κατάταξη γλωσσών προγραμματισμού ως προς τον βαθμό αντικειμενοστρέφειάς τους

- *Αμιγώς αντικειμενοστρεφείς* (pure object oriented). ***Όλα*** τα στοιχεία τους είναι *αντικείμενα (από χαρακτήρες και σημεία στίξης, μέχρι δομές δεδομένων κλάσεις, μπλοκ, δομοστοιχεία κλπ.)*
  - Smalltalk, Ruby, ***Python*** κ.α.
- Σχεδιασμένες *κυρίως για αντικειμενοστρέφεια*, αλλά *διαθέτουν* και *στοιχεία διαδικαστικών γλωσσών* (object oriented / procedural)
  - Java, C++, C#, Delphi/Object Pascal, VB.NET κ.α.
- *Ιστορικά διαδικαστικές* (procedural) οι οποίες έχουν **επεκταθεί** ώστε να **περιλαμβάνουν** κάποια **στοιχεία αντικειμενοστρέφειας**
  - Visual Basic, MATLAB, Fortran 2003, Pascal κ.α.

Αντικείμενα και Κλάσεις αντικειμένων στον φυσικό κόσμο



Η έννοια του αντικειμένου

Στην καθημερινή μας ζωή περιστοιχιζόμαστε από κάθε είδους *αντικείμενα*

Κάθε αντικείμενο που χρησιμοποιούμε έχει

- *Τα δικά του χαρακτηριστικά* (ιδιότητες)

  πχ. σχήμα, χρώμα, βάρος, διαστάσεις, υλικά κατασκευής

- *Τον δικό του τρόπο χρήσης* (μεθόδους χειρισμού)

  πχ. για μια τηλεόραση: άνοιγμα / κλείσιμο, επιλογή καναλιών, ρύθμιση έντασης, εγγραφή βίντεο, κλπ.

Η έννοια του αντικειμένου

Παράδειγμα: Ένα αυτοκίνητο

- *Χαρακτηριστικά*: Μάρκα, μοντέλο, χρώμα, αριθμός κυκλοφορίας, χιλιόμετρα που έχει διανύσει, είδος καυσίμου, τελική ταχύτητα, πλήθος θυρών…

- *Τρόποι χειρισμού:* Ξεκίνημα, επιτάχυνση, αλλαγή ταχύτητας, φρενάρισμα, στρίψιμο, άνοιγμα-κλείσιμο πόρτας/καπό / πορτ-μπαγκάζ…

Παρατηρείστε ότι κάποια χαρακτηριστικά ενός αυτοκινήτου μπορεί να είναι *κοινά με άλλα αυτοκίνητα (μάρκα, χρώμα, πλήθος θυρών…),* ενώ κάποια *άλλα αφορούν αποκλειστικά το συγκεκριμένο αυτοκίνητο (αριθμός πλαισίου, αριθμός κυκλοφορίας…)*

Η έννοια της κλάσης αντικειμένων

Αντικείμενα με *κοινά χαρακτηριστικά* και *τρόπους χειρισμού* ταξινομούνται σε *κλάσεις (κατηγορίες)*

Παράδειγμα: Η κλάση 'αυτοκίνητα'

Τα αυτοκίνητα μπορεί να *διαφέρουν από κατηγορία σε κατηγορία (οικογενειακά, σπορ, επαγγελματικά)*, και *από μάρκα σε μάρκα*, όλα όμως έχουν κάποια *γενικά χαρακτηριστικά* και *τρόπους χειρισμού*

- *Γενικά χαρακτηριστικά*: Μάρκα, μοντέλο, χρώμα, είδος καυσίμου, τελική ταχύτητα, πλήθος θυρών…
- *Τρόποι χειρισμού:* Ξεκίνημα, επιτάχυνση, αλλαγή ταχύτητας, φρενάρισμα, στρίψιμο, …

- Τα *χαρακτηριστικά* και η *υλοποίηση* των μηχανισμών ελέγχου ενός αυτοκινήτου *διαφέρουν από κατασκευαστή σε κατασκευαστή*, αλλά ο *τρόπος χειρισμού* (τιμόνι, πεντάλ…) είναι *κοινός για όλα*

- Χαρακτηριστικά που αφορούν *συγκεκριμένα μέλη της κλάσης (αριθμός πλαισίου, αριθμός κυκλοφορίας…) δεν περιλαμβάνονται* στα γενικά χαρακτηριστικά της κλάσης και θεωρούνται *ειδικά χαρακτηριστικά μέλους της κλάσης*

Επίσης *ο τρόπος υλοποίησης* των μηχανισμών και η *πολυπλοκότητά* τους *αποκρύπτεται* από τον οδηγό, πχ μπορούμε να *αλλάζουμε ταχύτητες* χωρίς να έχουμε *ιδέα* για το πώς λειτουργεί το *κιβώτιο ταχυτήτων (βλ. έννοια της ενθυλάκωσης παρακάτω)*

Γενικές και πιο εξειδικευμένες κλάσεις

Οι κλάσεις μπορεί να είναι

Πολύ γενικές
πχ '*αυτοκίνητα*'

Ιδιαίτερα
εξειδικευμένες
πχ '*Tesla Model 3*'

Οι *εξειδικευμένες* κλάσεις συνήθως προκύπτουν από *γενικότερες*,
από τις οποίες *κληρονομούν* κάποια *βασικά χαρακτηριστικά*

Μεταφορικά μέσα > Οχήματα ξηράς > Αυτοκίνητα > Αμιγώς ηλεκτρικά > ***Tesla Model 3***

Ζώα > Σπονδυλωτά > Θηλαστικά > Αιλουροειδή > ***Γάτες***

Όταν μια κλάση Β προκύπτει ως *εξειδίκευση* μιας κλάσης Α,
τότε η Β καλείται *θυγατρική* κλάση *(child class)* της Α,
και ταυτόχρονα η Α καλείται *γονεϊκή* κλάση *(parent class)* της Β

Ειδικά γνωρίσματα αντικειμενοστρέφειας

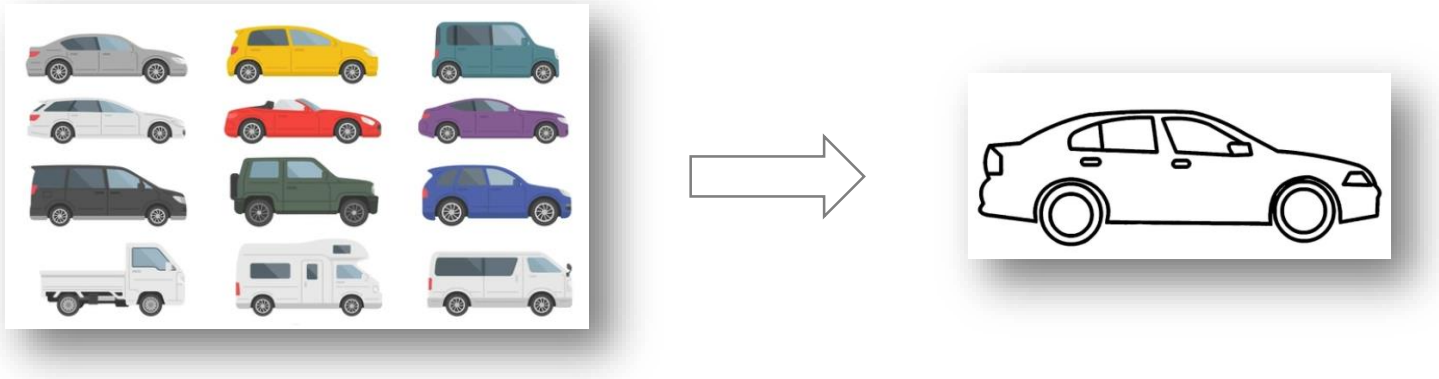
- ***Αφαίρεση*** *(abstraction)*
- ***Ενθυλάκωση*** *(encapsulation)*
- ***Κληρονομικότητα και σύνθεση*** *(inheritance and composition)*
- ***Πολυμορφισμός*** *(polymorphism)*

Ειδικά γνωρίσματα αντικειμενοστρέφειας

Αφαίρεση */ Ενθυλάκωση / Κληρονομικότητα και σύνθεση / Πολυμορφισμός*

Αφαίρεση (abstraction): Μια κλάση αποτελεί ένα *απλοποιημένο μοντέλο-άποψη* μιας *πολύπλοκης οντότητας* του φυσικού κόσμου, που διατηρεί μόνο τα *στοιχεία που κρίνονται σημαντικά* για την συγκεκριμένη μοντελοποίηση

Φυσικό παράδειγμα: Κατά την δημιουργία της κλάσης 'αυτοκίνητα' μοντελοποιούμε *μόνο τα χαρακτηριστικά και τις μεθόδους που μας ενδιαφέρουν (σασί, ρόδες, …. αλλαγή ταχύτητας, επιτάχυνση, φρενάρισμα)* και *αγνοούμε τις άσχετες λεπτομέρειες* όπως ~~*χρώμα και υλικό ταπετσαρίας καθισμάτων, διαστάσεις ελαστικών, ρυθμίσεις καθρέπτη, ενεργοποίηση αλάρμ*~~ *κλπ*.



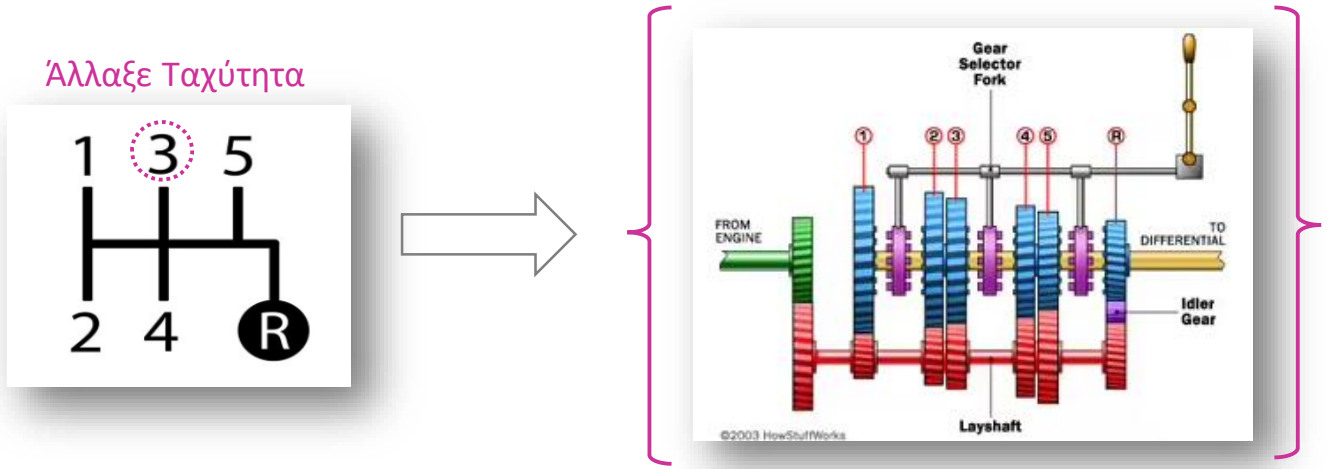
Ειδικά γνωρίσματα αντικειμενοστρέφειας

*Αφαίρεση / **Ενθυλάκωση** / Κληρονομικότητα και σύνθεση / Πολυμορφισμός*

Ενθυλάκωση (encapsulation). Οι *λεπτομέρειες υλοποίησης* μιας κλάσης *αποκρύπτονται* από το περιβάλλον χρήσης της

Η κλάση έχει μια *δημόσια διεπαφή* η οποία περιγράφει τις *ιδιότητες και μεθόδους χρήσης της* ενώ η *εσωτερική δομή* της παραμένει *ιδιωτική*. Έτσι προστατεύονται τα εσωτερικά δεδομένα από ανεπιθύμητη προσπέλαση / μεταβολή

Φυσικό παράδειγμα: Ο τρόπος *(μέθοδος)* αλλαγής ταχύτητας σε ένα αυτοκίνητο

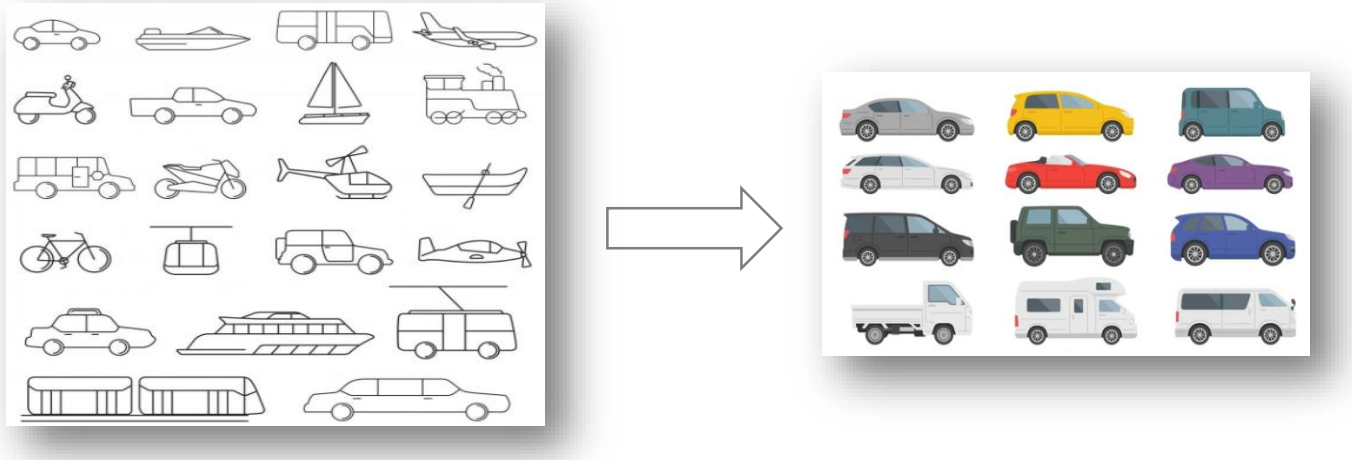


Ειδικά γνωρίσματα αντικειμενοστρέφειας

*Αφαίρεση / Ενθυλάκωση / **Κληρονομικότητα και σύνθεση** / Πολυμορφισμός*

Κληρονομικότητα και σύνθεση (inheritance and composition). Όταν δημιουργούνται *νέες κλάσεις* που βασίζονται και εξειδικεύουν προϋπάρχουσες, τότε αυτές *κληρονομούν δομικά στοιχεία (ιδιότητες και μεθόδους)* των αρχικών κλάσεων

Φυσικό παράδειγμα: Η κλάση '*αυτοκίνητα*' όταν δημιουργείται ως εξειδίκευση της κλάσης '*μεταφορικά μέσα*' κληρονομεί κάποια *δομικά* της στοιχεία *(διαστάσεις, βάρος, πλήθος επιβατών… επιτάχυνση, επιβράδυνση, στρίψιμο)* προσθέτοντας ταυτόχρονα άλλα πιο *εξειδικευμένα (πχ. πλήθος θυρών, πλήθος αξόνων/ελαστικών… αλλαγή ταχύτητας…)*

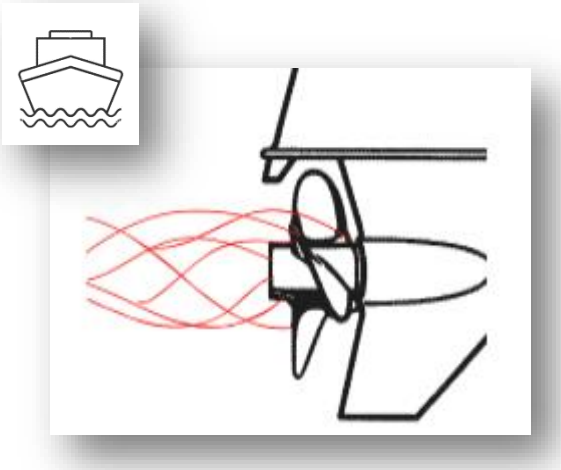


Ειδικά γνωρίσματα αντικειμενοστρέφειας

*Αφαίρεση / Ενθυλάκωση / Κληρονομικότητα και σύνθεση / **Πολυμορφισμός***

Πολυμορφισμός (polymorphism). Προκύπτει όταν *διαφορετικές κλάσεις* που έχουν δημιουργηθεί κληρονομώντας στοιχεία από την *ίδια βασική κλάση* έχουν κάποιες μεθόδους με *ίδιο όνομα* και *ίδια λειτουργικότητα* υλοποιημένες με *διαφορετικό τρόπο* ανάλογα με την περίσταση

Φυσικό παράδειγμα: Οι κλάσεις '*αυτοκίνητα*' και '*πλοία*' που δημιουργήθηκαν ως εξειδικεύσεις της κλάσης '*μεταφορικά μέσα*' έχουν και οι δυο κληρονομήσει την μέθοδο '*επιβράδυνση* ' *που έχει ίδιο όνομα και ίδια λειτουργικότητα αλλά υλοποιείται με εντελώς διαφορετικό τρόπο (αυτοκίνητα: άσκηση τριβής πάνω στις ρόδες, πλοία: μείωση στροφών έλικας ή/και αλλαγή φοράς στρέψης έλικας)*



Αντικείμενα και Κλάσεις αντικειμένων στην Python

```
class Student(Person): #Child class of Person
  def __init__(self, fname, lname, year):
      Person.__init__(self, fname, lname)
      self.graduation_year = year

  def welcome(self):
        print('Welcome', self.first
        self.lastname, 'to the clas
        self.graduation_year)

>>> y = Student('Mary', 'Jones', 20
>>> y.printname()
Mary Jones
>>> y.welcome()
Welcome Mary Jones to the class of
```

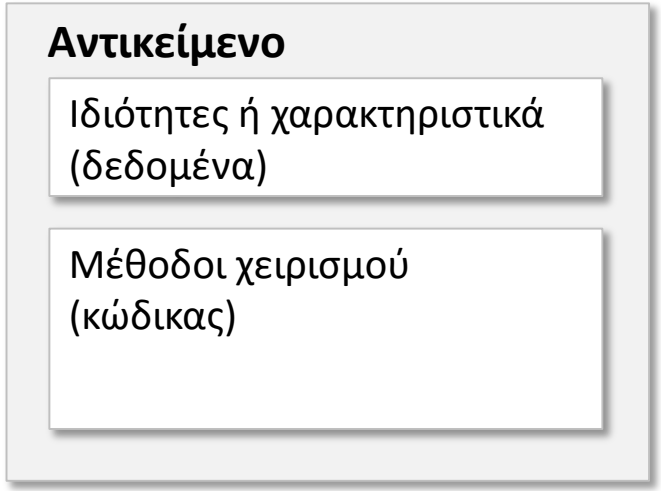
```
>>> class Dog:
        gene = 'Canis lupus'
        def __init__(self, name):
            self.name = name
            self.age = 0

        def get_age(self):
            return self.age

        def set_age(self, age):
            self.age = age

        def bark(self):
            print('Woof')
```

Η έννοια του αντικειμένου στον προγραμματισμό

Θυμόμαστε ότι: στον *αντικειμενοστρεφή προγραμματισμό* το πρόβλημα κωδικοποιείται σε μια συλλογή από *διακριτά αντικείμενα* τα οποία αλληλεπιδρούν για την επίλυση του

Κάθε αντικείμενο αποτελείται από

- ***δεδομένα*** που συνήθως ονομάζονται *ιδιότητες ή χαρακτηριστικά* (*attributes*)
- ***κώδικα*** υπό μορφή *διαδικασιών ή συναρτήσεων* που συνήθως ονομάζονται *μέθοδοι [χειρισμού]* (*methods*) και περιγράφουν τις ενέργειες που ορίζονται πάνω στα χαρακτηριστικά του αντικειμένου



Αντικείμενα υπάρχουν σε πολλές γλώσσες προγραμματισμού, όμως οι *αμιγώς αντικειμενοστρεφείς γλώσσες* (Python, Java…) υποστηρίζουν επιπλέον την έννοια της *κλάσης* (*class*) σύμφωνα με την οποία όλα τα αντικείμενα αποτελούν *στιγμιότυπα* (*instances*) κλάσεων οι οποίες *προσδιορίζουν τον τύπο τους*

Κλάσεις και αντικείμενα στην Python

- Τα πάντα στην Python είναι *αντικείμενα (σταθερές, μεταβλητές, λίστες, σύνολα, συναρτήσεις, αρχεία…)* και αποτελούν *εκπροσώπους -ή αλλιώς στιγμιότυπα- κάποιας προκαθορισμένης κλάσης*
- Για να δούμε σε ποια κλάση ανήκει ένα αντικείμενο χρησιμοποιούμε τη συνάρτηση `type`

```
>>> type(42)
<class 'int'>
```

```
>>> type(23.1)
<class 'float'>
```

```
>>> type([1,2])
<class 'list'>
```

```
>>> type({1,2})
<class 'set'>
```

```
>>> type(None)
<class 'NoneType'>
```

```
>>> type(type)
<class 'type'>
```

Ακόμα και η ίδια η συνάρτηση `type` είναι αντικείμενο και ανήκει σε δική της κλάση

```
>>> x = 42
>>> type(x)
<class 'int'>
```

```
>>> y = 23.1
>>> type(y)
<class 'float'>
```

```
>>> L = [1,2]
>>> type(L)
<class 'list'>
```

```
>>> a = {1,2}
>>> type(a)
<class 'set'>
```

```
>>> d = {1:2}
>>> type(d)
<class 'dict'>
```

```
>>> def f():
        pass
>>> type(f)
<class 'function'>
```

```
>>> f1= open('myfile.txt', 'wt')
>>> type(f1)
<class '_io.TextIOWrapper'>
```

```
>>> f2= open('myfile.dat', 'wb')
>>> type(f2)
<class '_io.BufferedWriter'>
```

```
>>> import math
>>> type(math)
<class 'module'>
```

```
>>> type(len)
<class 'builtin_function_or_method'>
```

Τα δυο βασικά δομικά στοιχεία των κλάσεων

- Ιδιότητες ή χαρακτηριστικά (attributes): Χρησιμοποιούνται για την περιγραφή ιδιοτήτων και καταστάσεων μιας κλάσης

```
>>> f.name
'C:/Users/MK/my_friends.txt'
>>> f.mode
'w'
>>> f.closed
True
```

Τα παραδείγματα αναφέρονται στην κλάση της Python `_io.TextIOWrapper` στην οποία ανήκουν τα αρχεία κειμένου

- Μέθοδοι χειρισμού (methods): *Συναρτήσεις* που ορίζονται μέσα στην κλάση και περιγράφουν συγκεκριμένες *ενέργειες* που ορίζονται *πάνω στα χαρακτηριστικά της*. Αποτελούν ουσιαστικό στοιχείο της ιδέας της *ενθυλάκωσης* (*encapsulation*) στο αντικειμενοστρεφές υπόδειγμα προγραμματισμού. Ξεχωρίζουν από τις ιδιότητες επειδή ακολουθούνται πάντα από παρενθέσεις

```
>>> f.write(names)
>>> f.close()
```

Οι *λεπτομερείς ενέργειες* που πραγματοποιούνται για την αποθήκευση δεδομένων στο αρχείο, είναι *κωδικοποιημένες στην συνάρτηση* (μέθοδο) *write*. Ο χρήστης εντούτοις δεν ενδιαφέρεται για αυτές τις λεπτομέρειες. Απλώς καλεί την *write*, η οποία και *αναλαμβάνει την αποθήκευση των δεδομένων* όπως εκείνη κρίνει καλύτερα (encapsulation of complexity)

Πώς μπορούμε να δούμε όλες τις *ιδιότητες* και *μεθόδους* μιας κλάσης

Χρησιμοποιούμε την συνάρτηση `dir()` για να πάρουμε τη λίστα με όλα τα "εσωτερικά" ονόματα των στοιχείων μιας κλάσης. Στην παρένθεση της `dir()` δίνουμε:

- είτε το όνομα της *ίδιας της κλάσης*
  πχ `dir('_io.TextIOWrapper')`
- είτε το όνομα ενός *στοιχείου (στιγμιότυπου)* της κλάσης
  πχ `dir(fileobj)` *αν προηγουμένως έχουμε ορίσει πχ* `fileobj = open('a.txt', 'w')`

```
>>> fileobj = open('a.txt', 'w')
>>> dir(fileobj)
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__',
'__eq__', '__exit__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '_checkClosed', '_checkReadable', '_checkSeekable', '_checkWritable',
'_finalizing', 'buffer', 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno', 'flush',
'isatty', 'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline', 'readlines',
'reconfigure', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write', 'write_through',
'writelines']
```

Οι *ιδιότητες* και οι *μέθοδοι* της κλάσης των *αρχείων κειμένου*, είναι όσα από τα παραπάνω ονόματα ***δεν αρχίζουν*** *από κάτω παύλα, ή διπλή κάτω παύλα* (αυτά που έχουν τονιστεί με μπλε χρώμα παραπάνω)

Πώς ξεχωρίζουμε τις *ιδιότητες* από τις *μεθόδους* μιας κλάσης

Οι *ιδιότητες* και οι *μέθοδοι* μιας κλάσης, είναι όπως είπαμε εκείνα τα ονόματα που επιστρέφονται από την `dir()` τα οποία ***δεν αρχίζουν*** *από κάτω παύλα, ή διπλή κάτω παύλα*

```
>>> dir(fileobj)
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__',
'__eq__', '__exit__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_checkClosed', '_checkReadable', '_checkSeekable', '_checkWritable', '_finalizing',
'buffer', 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno', 'flush', 'isatty',
'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline', 'readlines',
'reconfigure', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write', 'write_through',
'writelines']
```

ΙΔΙΟΤΗΤΕΣ ΚΑΙ ΜΕΘΟΔΟΙ

Για να ξεχωρίσουμε τις ιδιότητες από τις μεθόδους χρησιμοποιούμε την συνάρτηση `callable(object)` *που επιστρέφει* `True` *αν ένα αντικείμενο μπορεί να κληθεί μέσα από ένα πρόγραμμα (άρα είναι μέθοδος ή γενικότερα συνάρτηση) και* `False` *διαφορετικά.*

πχ `callable(getattr(fileobj, 'read'))` επιστρέφει `True`. Επομένως το `'read'` είναι μέθοδος

ενώ `callable(getattr(fileobj, 'mode'))` επιστρέφει `False` Επομένως το `'mode'` είναι ιδιότητα

Εδώ η συνάρτηση `getattr` χρησιμοποιείται για να επιστρέψει την ιδιότητα ή μέθοδο ενός αντικειμένου μιας κλάσης, από την συμβολοσειρά που χρησιμοποιείται για να το προσδιορίσει.

Εφαρμογή: εκτύπωση ιδιοτήτων και μεθόδων μιας κλάσης

```
#Δημιουργία ενός αντιπροσώπου της κλάσης αρχείων κειμένου
fileobj = open('test.txt', 'w')

attributes_and_methods = [x for x in dir(fileobj) if not x.startswith("_")]

attributes = [x for x in attributes_and_methods if not callable(getattr(fileobj, x))]
methods = [x for x in attributes_and_methods if callable(getattr(fileobj, x))]

print(type(obj))
print('Ιδιότητες\n', attributes)
print('Μέθοδοι\n', methods)
```

```
<class '_io.TextIOWrapper'>
Ιδιότητες
['buffer', 'closed', 'encoding', 'errors', 'line_buffering', 'mode', 'name', 'newlines',
'write_through']
Μέθοδοι
['close', 'detach', 'fileno', 'flush', 'isatty', 'read', 'readable', 'readline', 'readlines',
'reconfigure', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write', 'writelines']
```

Δημιουργία νέων κλάσεων στην Python

Στην Python μπορούμε να δημιουργήσουμε *και νέες δικές μας κλάσεις* πέρα από τις ήδη υπάρχουσες, και στη συνέχεια να *ορίσουμε στιγμιότυπα (instances) ή αλλιώς εκπροσώπους* πάνω σ' αυτές

- Ο ορισμός μιας κλάσης γίνεται με την με τη λέξη-κλειδί `class` ως εξής

```
class <όνομα κλάσης>:
 .... <μπλοκ εντολών κλάσης>
```

- Το μπλοκ εντολών που ακολουθεί περιγράφει τα *χαρακτηριστικά ή αλλιώς ιδιότητες της κλάσης (class attributes)* και τις *μεθόδους χειρισμού της κλάσης (class methods)*

  Παράδειγμα μιας πολύ απλής κλάσης που περιέχει μόνο μια ιδιότητα

```
>>> class Cat:
        gene = "Felis_catus" # ιδιότητα της κλάσης
```

  Εναλλακτικά μπορούν να υπάρχουν και παρενθέσεις δίπλα στο όνομα της κλάσης

```
>>> class Cat():
```

  Ειδική περίπτωση: Αν η κλάση ορίζεται ως *εξειδίκευση/επέκταση* μιας γενικότερης προϋπάρχουσας *κλάσης-γονέα* (*parent class*) τότε η παρένθεση περιέχει το όνομα αυτής της κλάσης γονέα. Στην περίπτωση αυτή, η νέα κλάση ονομάζεται *κλάση-παιδί (child class)* Παράδειγμα: `class Cat(Animals)`

Δημιουργία στιγμιότυπων (εκπροσώπων) κλάσεων

Συνεχίζοντας το προηγούμενο παράδειγμα, έστω ότι έχουμε ορίσει την εξής κλάση

```
>>> class Cat:
        gene = 'Felis_catus' # ιδιότητα της κλάσης που έχει την τιμή 'Felis_catus'
        name = ''            # ιδιότητα της κλάσης χωρίς κάποια τιμή προς το παρόν
```

Μπορούμε τώρα να ορίσουμε έναν *εκπρόσωπο* ή αλλιώς *στιγμιότυπο (instance)* της κλάσης `Cat`

```
>>> my_cat = Cat()
```

Για να αναφερθούμε σε μια ιδιότητα ενός στιγμιότυπου γράφουμε: `<στιγμιότυπο>.<ιδιότητα>`

```
>>> my_cat.gene
'Felis_catus'
>>> my_cat.name
''
>>> my_cat.name = 'Molly'
>>> my_cat.name
'Molly'
```

Μπορούμε να αναφερθούμε και απ' ευθείας στην ιδιότητα *της ίδιας της κλάσης*

```
>>> Cat.gene
'Felis_catus'
```

Δημιουργία στιγμιότυπων (εκπροσώπων) κλάσεων

Οι *γενικές ιδιότητες μιας κλάσης* (*class attributes*) περνάνε σε όλα τα στιγμιότυπά της.

Επιπλέον, μια κλάση μπορεί να περιέχει και ιδιότητες στις οποίες αντιστοιχούμε *ξεχωριστές τιμές για κάθε στιγμιότυπο.* Οι ιδιότητες αυτές καλούνται *ειδικές ιδιότητες δεδομένων του στιγμιότυπου* (*data attributes*).

```
>>> class Cat:
        gene = 'Felis_catus'
        name = ''
        color = ''
        age = 0
```

Γενικές ιδιότητες κλάσης (class attributes)

Ειδικές ιδιότητες δεδομένων στιγμιότυπου (data attributes)
Υπό κανονικές συνθήκες, ***δεν ορίζονται εδώ****.*
(θα δούμε που ακριβώς, σε επόμενη διαφάνεια)

```
>>> my_cat = Cat()
>>> my_cat.name = 'Molly'
>>> my_cat.color = 'white'
>>> my_cat.age = 2

>>> your_cat = Cat()
>>> your_cat.name = 'Jimmy'
>>> your_cat.color = 'red'
>>> your_cat.age = 3
```

- Οι *γενικές ιδιότητες της κλάσης* μεταφέρονται *αυτόματα* σε κάθε νέο στιγμιότυπο που δημιουργείται
- Οι *ειδικές ιδιότητες δεδομένων* αρχικοποιούνται *μετά* την δημιουργία του στιγμιότυπου (ή και *ταυτόχρονα* με την δημιουργία του όπως θα δούμε παρακάτω)

Αρχικοποίηση στιγμιότυπων: Η μέθοδος __init__

Οι ειδικές ιδιότητες δεδομένων (*data attributes*) *(όπως πχ. name, age)* ενός στιγμιότυπου μιας κλάσης μπορούν να καθορίζονται και *ταυτόχρονα με την δημιουργία του στιγμιότυπου* όπως δείχνει το επόμενο παράδειγμα:

```
>>> my_cat = Cat('Molly','white',2)
```

Για να γίνει αυτό, ορίζουμε μέσα στην κλάση την ειδική μέθοδο (συνάρτηση) `__init__` *(διπλές κάτω παύλες)*

```
>>> class Cat:
        gene = 'Felis_catus'
        def __init__(self, p1, p2, p3):
            self.name = p1
            self.color = p2
            self.age = p3
>>> my_cat = Cat('Molly','white',2)
>>> your_cat = Cat('Jimmy','red',3)
```

*Γενικές ιδιότητες κλάσης **κοινές για όλα τα στιγμιότυπα***

Ειδικές ιδιότητες δεδομένων
διαφορετικές για κάθε στιγμιότυπο
*Ορίζονται μέσω της **__init__***

Η *πρώτη παράμετρος κάθε μεθόδου* που ορίζουμε μέσα σε μια κλάση `(self)` αναφέρεται πάντα στο *όνομα του στιγμιότυπου* πάνω στο οποίο εφαρμόζεται η μέθοδος. Γι αυτή την παράμετρο *έχει επικρατήσει το όνομα* `self` παρόλο που θεωρητικά μπορεί να χρησιμοποιηθεί οποιοδήποτε άλλο όνομα *(κάτι εντούτοις που δεν συνιστάται)*

Αλλαγές τιμών σε ιδιότητες ενός στιγμιότυπου κλάσης

<στιγμιότυπο> . <ιδιότητα> = <νέα τιμή ιδιότητας>

```
>>> your_cat.age = 4  #Αλλάζουμε την τιμή της ιδιότητας  age στο στιγμιότυπο your_cat
>>> print(your_cat.age)
4
```

Προσθήκη επιπλέον ιδιοτήτων *σε συγκεκριμένα* στιγμιότυπα

<στιγμιότυπο> . <νέα ιδιότητα> = <τιμή νέας ιδιότητας>

```
>>> your_cat.breed = 'Siamese' #Προσθέτουμε μια νέα ιδιότητα breed (ράτσα) στο στιγμιότυπο your_cat
>>> print(your_cat.breed)
Siamese
```

Οι επιπλέον ιδιότητες που ορίζουμε σε ενα συγκεκριμένο στιγμιότυπο, δεν υπάρχουν –προφανώς- στα υπόλοιπα στιγμιότυπα της ίδιας κλάσης

```
>>> print(my_cat.breed) #Η ιδιότητα breed  δεν υπάρχει στο στιγμιότυπο my_cat

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
AttributeError: 'my_cat' object has no attribute 'breed'
```

breed: ράτσα

Αλλαγές τιμών στις γενικές ιδιότητες μιας κλάσης

```
>>> class Cat:
  gene = "Felis_catus" # Γενική ιδιότητα κλάσης (κοινή για όλα τα στιγμιότυπα)
  def __init__(self, p1, p2, p3):
    self.name = p1
    self.color = p2     # Ειδικές ιδιότητες δεδομένων (ξεχωριστές για κάθε στιγμιότυπο). Ορίζονται μέσω της __init__
    self.age = p3

>>> my_cat = Cat("Molly", "Black", 2)
>>> your_cat = Cat( "Jimmy", "White", 4)


>>> my_cat.gene = "Siamese"
>>> my_cat.gene
'Siamese'
>>> your_cat.gene
'Felis_catus'
```

Προσοχή: Παρόλο που εκ πρώτης όψεως φαίνεται ότι αλλάζω την τιμή της *γενικής* ιδιότητας gene της κλάσης `Cat`, δεν ισχύει κάτι τέτοιο.

Στην πραγματικότητα έχω προσθέσει μια ακόμα ειδική ιδιότητα *(στην ουσία μια τοπική μεταβλητή)* στο στιγμιότυπο `my_cat`, στην οποία έχω δώσει ίδιο όνομα με τη γενική ιδιότητα gene της κλάσης `Cat`.

Στην περίπτωση αυτή το στιγμιότυπο `my_cat` δεν έχει πλέον πρόσβαση στη *γενική ιδιότητα* gene της κλάσης `Cat` αλλά μόνο στην ομώνυμη *νέα ειδική ιδιότητα* *(ομώνυμες local & global variables σε μια συνάρτηση)*

```
>>> Cat.gene = "Κεραμιδόγατος"
>>> my_cat.gene, your_cat.gene
('Siamese','Κεραμιδόγατος')
```

Ο μόνος τρόπος για να αλλάξω μια γενική ιδιότητα μιας κλάσης είναι να το κάνω πάνω στην ίδια την κλάση: `Cat.gene = '...'`

Η ιδιότητα `my_cat.gene` *παραμένει* `"Siamese"`*. Γιατί;*

Συνοψίζοντας τις ιδιότητες μιας κλάσης (attributes)

Χρησιμοποιούνται για την περιγραφή *χαρακτηριστικών* ή/και *καταστάσεων* μιας κλάσης.

- *Γενικές ιδιότητες της κλάσης* (class attributes): Ιδιότητες οι οποίες ανήκουν *στην κλάση* και όχι στο κάθε αντικείμενο-στιγμιότυπο της κλάσης που δημιουργείται
  - Ορίζονται μέσα στο μπλοκ της κλάσης *(συνήθως στην αρχή του)* ***έξω*** από την συνάρτηση `__init__` και ***έξω*** από οποιαδήποτε άλλη συνάρτηση-μέθοδο της κλάσης Θεωρούνται *κοινές ιδιότητες* για *όλα τα στιγμιότυπα* της κλάσης
  - Αν αλλάξουν τιμή *στην ίδια την κλάση*, τότε η αλλαγή αυτή επηρεάζει *όλα τα αντικείμενα -στιγμιότυπα της κλάσης*, ενώ αν αλλάξουν τιμή *μόνο σε ένα αντικείμενο-στιγμιότυπο*, η αλλαγή αυτή *δεν επηρεάζει τα υπόλοιπα στιγμιότυπα*
- *Ειδικές ιδιότητες των στιγμιότυπων της κλάσης* (data attributes): Ιδιότητες οι οποίες ανήκουν μόνο στο αντικείμενο-στιγμιότυπο της κλάσης που θα δημιουργηθεί
  - *Ορίζονται* και *αρχικοποιούνται* μέσα στην ειδική συνάρτηση `__init__(self,..)` της κλάσης
  - Αν αλλάξουν τιμή σε ένα αντικείμενο-στιγμιότυπο, η αλλαγή *δεν επηρεάζει τα υπόλοιπα στιγμιότυπα*
  - *Μια καλή πρακτική* είναι οι ειδικές ιδιότητες των στιγμιότυπων να μεταβάλλονται *μόνο μέσω ειδικών μεθόδων (συναρτήσεων)* που ορίζονται μέσα στην κλάση. *(Η Python συνιστά μεν, αλλά δεν επιβάλλει αυτή την πρακτική σε αντίθεση με άλλες γλώσσες όπως Java και C++)*

Μέθοδοι κλάσης (methods)

Συναρτήσεις που ορίζονται μέσα στην κλάση και περιγράφουν *συγκεκριμένες ενέργειες πάνω στα χαρακτηριστικά της*

Η *πρώτη παράμετρος κάθε μεθόδου* αναφέρεται πάντα στο *όνομα του στιγμιότυπου* πάνω στο οποίο εφαρμόζεται η μέθοδος. Γι αυτή την παράμετρο *έχει επικρατήσει η χρήση του προσδιοριστή* `self`

Θεωρητικά μπορεί να χρησιμοποιηθεί *οποιοδήποτε άλλο όνομα* εκτός από `self`, όμως αυτό *δεν συνιστάται*.

```
>>> class Dog:
        gene = 'Canis lupus'
        def __init__(self, name):
            self.name = name
            self.age = 0

        def get_age(self):
            return self.age

        def set_age(self, age):
            self.age = age

        def bark(self):
            print('Woof')
```

```
>>> my_dog = Dog('Rex')
>>> my_dog.name
'Rex'

>>> my_dog.bark()
Woof

>>> my_dog.get_age()
0

>>> my_dog.set_age(5)
>>> my_dog.get_age()
5
```

Μέθοδοι κλάσης (methods)

Μια μέθοδος *μπορεί να κληθεί από άλλη μέθοδο μέσα στην ίδια κλάση*, με χρήση του `self`.

Πχ. η προηγούμενη κλάση `Dog` θα μπορούσε να έχει μια ακόμα μέθοδο:

```
def bark2times(self):
    self.bark()
    self.bark()
. . .
>>> my_dog.bark()
Woof
>>> my_dog.bark2times()
Woof
Woof
```

ορίζεται και αυτή μέσα στην κλάση Dog

Εναλλακτικές συντακτικές μορφές κλήσης μεθόδων

Αρχικά, η κλήση μιας μεθόδου από ένα στιγμιότυπο μιας κλάσης είχε την συντακτική μορφή:

`κλάση.μέθοδος(στιγμιότυπο, παράμετροι)` πχ. `Dogs.set_age(my_dog,5)`

Αρχικός τρόπος σύνταξης

Μεταγενέστερα όμως η μορφή αυτή απλουστεύθηκε ως εξής:

`στιγμιότυπο.μέθοδος(παράμετροι)` πχ. `my_dog.set_age(5)`

Νεότερος, απλουστευμένος τρόπος

Παρατηρούμε ότι παρόλο που *επιβάλλεται* να καθοριστεί η παράμετρος `self` (που ορίζει το στιγμιότυπο) *κατά τον ορισμό* της μεθόδου στην κλάση…

Ορισμός μεθόδου:
Το self είναι απαραίτητο

```
def set_age(self, age):
        self.age = age
```

…*όταν η μέθοδος καλείται* με τον απλουστευμένο τρόπο *(που πρακτικά μόνος αυτός χρησιμοποιείται)*, το όνομα του στιγμιότυπου ήδη υπάρχει μπροστά από το όνομα της μεθόδου, οπότε δεν χρειάζεται να μπει ως πρώτο όρισμα στην παρένθεση

```
>>> my_dog.set_age( 5)
```

Κλήση μεθόδου με τον απλουστευμένο τρόπο:
Η πρώτη παράμετρος (self) παραλείπεται καθώς το όνομα του στιγμιότυπου *(εδώ το my_dog)* ήδη υπάρχει στην σύνταξη της εντολής

Κατηγορίες μεθόδων/συναρτήσεων κλάσης

1. **Κανονικές** □□□□

   Όλες οι συναρτήσεις που ορίζονται μέσα σε μια κλάση, *εφόσον το όνομά τους δεν ξεκινάει με διπλό underscore (__)*. Οι συναρτήσεις αυτές αποτελούν τις ***μεθόδους της κλάσης*** και χρησιμοποιούνται από τα στιγμιότυπα της κλάσης για να μεταβάλλουν τα χαρακτηριστικά τους

2. **Ιδιωτικές** __□□□□

   Συναρτήσεις που ορίζονται μέσα σε μια κλάση, των οποίων το όνομα *ξεκινάει με διπλό underscore (__)* αλλά *δεν τερματίζει με διπλό underscore (__)* ονομάζονται ιδιωτικές και μπορούν να καλούνται ***μόνο από άλλες συναρτήσεις στο εσωτερικό της κλάσης*** *(δεν αναγνωρίζονται έξω από την κλάση - στην ουσία πρόκειται για τοπικές συναρτήσεις)*

3. **Ειδικές** __□□□□__

   Συναρτήσεις που ορίζονται μέσα σε μια κλάση και το όνομά τους ταυτίζεται με κάποιο από τα ειδικά χαρακτηριστικά, εκείνα δηλαδή που *ξεκινάνε και τερματίζουν με διπλό underscore (__)* Οι συναρτήσεις αυτές αποτελούν *τις ειδικές μεθόδους της κλάσης* και χρησιμοποιούνται για να *επανακαθορίσουν / εξειδικεύσουν* την τιμή κάποιων *ειδικών χαρακτηριστικών* σύμφωνα με τις απαιτήσεις του προγραμματιστή της κλάσης (πχ `__init__`)

Ιδιωτικές συναρτήσεις κλάσης (παράδειγμα)

```
>>> class Dog:
        def __init__(self, name):
            self.name = name
            self.age = 0

        def get_age(self):
            return self.age

        def set_age(self, age):
            self.age = age

        def __bark(self):
            print('Woof')

        def bark2times(self):
            self.__bark()
            self.__bark()
```

```
>>> d = Dog("Rex")
>>> d.bark2times()
Woof
Woof
>>> d.__bark()
Traceback (most recent call last):
File "<pyshell#53>", line 1, in <module> d.__sit()
AttributeError: 'Dog' object has no attribute '__bark'
```

Ιδιωτική συνάρτηση (μέθοδος).
Δεν είναι ορατή έξω από την κλάση

Ειδικές μέθοδοι κλάσης

Μέθοδοι που χρησιμοποιούνται για να *επανακαθορίσουν / εξειδικεύσουν* την τιμή κάποιων *ομώνυμων ειδικών χαρακτηριστικών* σύμφωνα με τις απαιτήσεις του προγραμματιστή της κλάσης

Ξεκινάνε και τερματίζουν με *διπλή κάτω παύλα* (double underscore -> dunder). Ιδού μερικές

`__init__`	Χρησιμοποιείται για να *αρχικοποιήσει τα ειδικά χαρακτηριστικά* κάθε νέου στιγμιότυπου της κλάσης
`__str__`	Χρησιμοποιείται για να αντικαταστήσει την *γενική και αόριστη* περιγραφή του στιγμιότυπου μιας κλάσης που εκτυπώνεται μέσα από το print με μια πιο *ξεκάθαρη και φιλική* περιγραφή *Απευθύνεται κυρίως σε χρήστες της κλάσης (παράδειγμα στην επόμενη διαφάνεια)* *Πχ*: Το `>>> print(my_cat)` αντί για `<__main__.Cat object at 0x0358F988>`, θα μπορούσε να εκτυπώνει `A 2 years old cat named Molly`
`__repr__`	Χρησιμοποιείται για να δημιουργήσει μια πιο *"τεχνική" περιγραφή* του στιγμιότυπου η οποία αντικαθιστά την γενική και αόριστη περιγραφή που επιστρέφει η Python. *Απευθύνεται κυρίως σε προγραμματιστές* *Πχ*: Το `>>> my_cat` αντί για `<__main__.Cat object at 0x0358F988>`, θα μπορούσε εκτυπώνει `Cat: name=Molly, color=white, age=2`

`__add__`
`__sub__`
`__mul__`
`__truediv__`
	Χρησιμοποιούνται για να *υπερφορτώσουν* τους τελεστές πρόσθεσης, αφαίρεσης, πολλαπλασιασμού και διαίρεσης *ώστε να χειρίζονται με ειδικό τρόπο* τα αντικείμενα της κλάσης
	Παράδειγμα: σε μια κλάση χειρισμού *διανυσμάτων* μπορούμε να υπερφορτώσουμε τον τελεστή του πολλαπλασιασμού ώστε να υπολογίζει το *εσωτερικό γινόμενο* των `d1` και `d2` γράφοντας απλά `d1 * d2`

Παραδείγματα στην επόμενη σελίδα

Παράδειγμα χρήσης __str__ και __repr__

```
>>> class Fruit:
        def __init__(self, nam
            self.name = name

>>> my_fruit = Fruit("Banana")

>>> my_fruit
<__main__.Fruit object at 0x7f0ece0e8d00>
>>> print(my_fruit)
<__main__.Fruit object at 0x7f0ece0e8d00>
```

Εντελώς τεχνικές περιγραφές καθώς δεν έχουμε ορίσει μέσα στην κλάση τις ειδικές μεθόδους __str__ και __repr__

Με την ειδική μέθοδο **__str__** καθορίζουμε το περιγραφικό κείμενο που θέλουμε να εμφανίζεται στην έξοδο του προγράμματος όταν ο χρήστης της κλάσης εκτυπώσει το όνομα ενός στιγμιότυπου της κλάσης *μέσα από το print* πχ. `print(my_fruit)`

Με την ειδική μέθοδο **__repr__** καθορίζουμε το κείμενο που θέλουμε να εμφανίζεται στην κονσόλα όταν ο χρήστης της κλάσης πληκτρολογήσει το όνομα ενός στιγμιότυπου της κλάσης πχ. `>>> my_fruit`

Στην δεύτερη περίπτωση, επειδή η *περιγραφή απευθύνεται κυρίως στον προγραμματιστή*, το κείμενο είναι κάπως πιο *"τεχνικό"*

```
>>> class Fruit:
        def __init__(self, name):
            self.name = name

        def __str__(self):
            return f'I am a {self.name}'

        def __repr__(self):
            return f'Fruit("{self.name}")'

>>> my_fruit = Fruit("Banana")

>>> my_fruit
Fruit("Banana")

>>> print(my_fruit)
I am a Banana
```

Φιλικότερες περιγραφές, καθώς έχουμε ορίσει μέσα στην κλάση τις ειδικές μεθόδους __str__ και __repr__

Κληρονομικότητα (inheritance)

- Η δυνατότητα δημιουργίας εξειδικευμένων κλάσεων (*θυγατρικές κλάσεις*- *child classes*) οι οποίες βασίζονται πάνω σε πιο γενικές κλάσεις (*γονεϊκές κλάσεις* - *parent classes*) τις οποίες εξειδικεύουν
- Οι γονεϊκές κλάσεις ονομάζονται αλλιώς και *υπερκλάσεις* (*superclasses*) και αντίστοιχα οι θυγατρικές κλάσεις ονομάζονται αλλιώς και *υποκλάσεις* (*subclasses*)
  - *Γονεϊκή κλάση* ή υπερκλάση *(parent class)*: η κλάση της οποίας τα χαρακτηριστικά κληρονομούνται
  - *Θυγατρική κλάση* ή υποκλάση *(child class)*: η κλάση η οποία κληρονομεί τα χαρακτηριστικά
- Οι θυγατρικές κλάσεις *κληρονομούν όλα τα βασικά χαρακτηριστικά και μεθόδους* των γονεϊκών κλάσεων από τις οποίες προέρχονται *προσθέτοντας επιπλέον* και τα δικά τους χαρακτηριστικά και μεθόδους
- Αποτέλεσμα: δημιουργία μιας *ιεραρχίας κλάσεων* ξεκινώντας από μία *γενική* η οποία *βρίσκεται στην κορυφή* και με την βοήθεια της *κληρονομικότητας* κατεβαίνουμε σε *όλο και πιο συγκεκριμένες υλοποιήσεις*

  Παράδειγμα: *Ζώα > Σπονδυλωτά > Θηλαστικά > Αιλουροειδή > Γάτες*
- Οι θυγατρικές κλάσεις *(υποκλάσεις)* μπορούν εφόσον χρειαστεί να ορίζουν εκ νέου με διαφορετικό τρόπο κάποιες από τις μεθόδους που κληρονόμησαν, *εξειδικεύοντάς τις κατάλληλα*

Κληρονομικότητα (Παράδειγμα)

```
class Person: #Father class
  def __init__(self, fname, lname):
     self.firstname = fname
     self.lastname = lname

  def printname(self):
     print(self.firstname, self.lastname)
```

```
>>> x = Person('John', 'Smith')
>>> x.printname()
John Smith
```

Στο παράδειγμα αυτό
οι κλάσεις `Person` και `Student`
είναι *μέχρι στιγμής* πανομοιότυπες

```
class Student(Person): #Child class of Person
     pass  # κληρονομεί όλες τις μεθόδους και τα χαρακτηριστικά της κλάσης Person
           # χωρίς να προστίθενται άλλες ιδιότητες ή μέθοδοι (προς το παρόν)
```

```
>>> y = Student ('Ted','Jones')
>>> y.printname()
Ted Jones
```

Κληρονομικότητα (συνέχεια του παραδείγματος)

```
class Student(Person): #Child class of Person
  def __init__(self, fname, lname, year):
      self.firstname = fname
      self.lastname = lname
      self.graduation_year = year

  def welcome(self):
        print('Welcome', self.firstname, self.lastname, 'to the class of', self.graduation_year)
```

Στη θυγατρική κλάση Student:

- η νέα μέθοδος `__init__` αντικατέστησε την αντίστοιχη γονεϊκή, *εξειδικεύοντας την για φοιτητές*
- προστέθηκε η μέθοδος `welcome`

```
>>> y = Student('Mary', 'Jones', 2025)
>>> y.printname() # η μέθοδος printname έχει κληρονομηθεί από την γονεϊκή κλάση Person
Mary Jones
>>> y.welcome() # η μέθοδος welcome έχει οριστεί στη θυγατρική κλάση Student
Welcome Mary Jones to the class of 2025
```

Κληρονομικότητα (συνέχεια του παραδείγματος)

Η συνάρτηση __init__ στη γονεϊκή κλάση Person, ήταν αρκετά απλή *(είχε μόνο τις δυο γραμμές κώδικα* `self.firstname = fname` *και* `self.lastname = lname`*)* οπότε απλώς τις αντιγράψαμε μέσα στην θυγατρική __init__ όταν την ορίζαμε *(και έτσι η θυγατρική κληρονόμησε τη συμπεριφορά αρχικοποίησης της γονεϊκής)*

Πιο σωστά όμως, αν θέλουμε η θυγατρική __init___ να αποτελεί *επέκταση* της γονεϊκής __init___ τότε *καλούμε* αρχικά *την γονεϊκή __init__ μέσα από την θυγατρική __init__* και μετά συνεχίζουμε να προσθέτουμε τις επιπλέον ιδιότητες που εξειδικεύουν την θυγρατρική κλάση *(εδώ πχ το graduation_year)*

```
def __init__(self, fname, lname, year):
    Person.__init__(self, fname, lname)
    self.graduation_year = year
```

ή εναλλακτικά ακόμα πιο σωστά

```
def __init__(self, fname, lname, year):
    super().__ init__(self, fname, lname)
    self.graduation_year = year
```

Η συνάρτηση `super()`

Αν χρησιμοποιηθεί μέσα σε μια θυγατρική κλάση, επιστρέφει *το όνομα της γονεϊκής της κλάσης*

Αυτό μας δίνει την δυνατότητα να *παραμετροποιούμε* το όνομα της γονεϊκής κλάσης μέσα στον κώδικα της θυγατρικής, ώστε *να μπορούμε εύκολα να αντικαθιστούμε μια γονεϊκή κλάση,* χωρίς να χρειάζεται να ψάχνουμε τον κώδικα της θυγατρικής και να αλλάζουμε το όνομα της παλιάς με την καινούρια παντού όπου υπάρχει

Πολυμορφισμός

Όταν *η ίδια συνάρτηση (συνάρτηση με το ίδιο όνομα)* χρησιμοποιείται με *διαφορετικό τρόπο* ανάλογα με τα δεδομένα της

Ή λίγο πιο "τεχνικά": Όταν *διαφορετικές κλάσεις* έχουν από μια μέθοδο/συνάρτηση με *ίδιο όνομα* και *ίδια λειτουργικότητα* υλοποιημένη με *ίδιο* ή *διαφορετικό τρόπο* ανάλογα με την περίσταση

Παράδειγμα η συνάρτηση `len()`

```
len('Hello')
```

```
len([1, 'a', 55, 8])
```

```
len({'a':12, 'b':15, 'c':23, 'd':8})
```

Πολυμορφισμός και κληρονομικότητα

Οι *θυγατρικές* κλάσεις *κληρονομούν* όλες τις μεθόδους των *γονεϊκών* κλάσεων. Αυτό καθιστά ***όλες τις κοινές μεθόδους τους πολυμορφικές***

Στην περίπτωση αυτή όλες οι πολυμορφικές μέθοδοι εκτός από *κοινό όνομα* έχουν και *κοινό κώδικα*

Εναλλακτικά μια θυγατρική κλάση μπορεί να *εξειδικεύσει* κάποιες από τις μεθόδους που κληρονόμησε, *διατηρώντας το όνομα τους*, επομένως και την *πολυμορφία* τους

Πολυμορφισμός και κληρονομικότητα

Οι θυγατρικές κλάσεις μπορούν να *μεταβάλλουν* κάποιες από τις μεθόδους που κληρονόμησαν, *εξειδικεύοντάς τις*. Στην ουσία πρόκειται για *νέες μεθόδους* με *ίδιο όνομα* και *διαφορετική (αν και παρόμοια)* λειτουργικότητα

Παράδειγμα:

```
class Animal:
    # Είδος
    # Γένος
    # ...

    def speak():
        print("I don't know what to say...")
```

```
class Cat(Animal): # Θυγατρική κλάσης της Animal
    def speak():  # Νέα μέθοδος speak. αντικαθιστά την γονεϊκή
        print("Meow!")

class Dog(Animal):  # Θυγατρική κλάσης της Animal
    def speak():  # Νέα μέθοδος speak. αντικαθιστά την γονεϊκή
        print("Woof!")
```

```
# Δημιουργία Στιγμιότυπων
>>> just_an_animal = Animal()
>>> my_dog = Dog()
>>> my_cat = Cat()

>>> my_dog.speak()
Woof!

>>> my_cat.speak()
Meow!

>>> just_an_animal.speak()
I don't know what to say...
```

Πολυμορφισμός και κληρονομικότητα στην πράξη

Ένα 'τρικ' που βοηθάει να γράφουμε καλύτερο κώδικα

Κάποιες φορές υπάρχει ανάγκη να δημιουργήσουμε *παρόμοιες (συναφείς)* κλάσεις που όλες έχουν κάποιες μεθόδους με *ίδιο όνομα* και *ίδια λειτουργικότητα*.

Πχ σε ένα πρόγραμμα ζωγραφικής, κάθε κατηγορία σχημάτων *(ορθογώνια, κύκλοι, ελλείψεις, βέλη κλπ.)* πρέπει να οριστεί σαν μια ξεχωριστή κλάση καθώς έχει τις δικές του ιδιαιτερότητες χειρισμού

Όμως παρά τις διαφορές τους, όλα τα σχήματα έχουν κάποιες κοινές ιδιότητες και μεθόδους χειρισμού που υλοποιούνται με τον ίδιο ακριβώς τρόπο.

Πχ ο,τιδήποτε έχει να κάνει με το χρώμα τους *(get_fillcolor set_fillcolor, getborder_color, setborder_color κλπ)* ή την γεωμετρία τους *(rotate, flip κλπ.)*

Ένα έξυπνο 'τρικ' για να μην χρειαστεί να γράφουμε τον ίδιο κώδικα *(κάνοντας στην ουσία copy-paste)* είναι να δημιουργήσουμε μια *κοινή γονεϊκή κλάση (σχήματα)* και να ορίσουμε εκεί όλες τις κοινές μεθόδους και ιδιότητες. Στη συνέχεια μπορούμε να ορίζουμε κάθε ξεχωριστό σχήμα ως θυγατρική κλάση με αποτέλεσμα *να κληρονομήσει όλες τις κοινές μεθόδους της γονεϊκής κλάσης*

Πολυμορφισμός και κληρονομικότητα στην πράξη

Εφαρμογή:

Σε ένα πρόγραμμα ζωγραφικής, για τις κλάσεις ορισμού σχημάτων `Circle, Rectangle, Square` κλπ.

α) Δημιουργούμε μια γονεϊκή κλάση `Shape` με *όλες τις κοινές ιδιότητες και μεθόδους*

Κοινές ιδιότητες

Κοινές μέθοδοι χειρισμού

```
def Shape:
    fillcolor = 0
    backcolor = 0

    def get_fillcolor(self,…):
      ...
    def set_fillcolor(self,… ):
      ...
    def get_backcolor(self,… ):
      ...
    def set_backcolor(self,… ):
      ...
    def rotate(self,… ):
      ...
    def flip(self,… ):
      ...
```

β) Ορίζουμε κάθε ξεχωριστό σχήμα ς ως *θυγατρική κλάση* της `Shape` ώστε να *κληρονομήσει αυτόματα όλες τις κοινές ιδιότητες και μεθόδους*

(στη συνέχεια φυσικά προσθέτουμε τις εξειδικευμένες μεθόδους κάθε κλάσης ξεχωριστά)

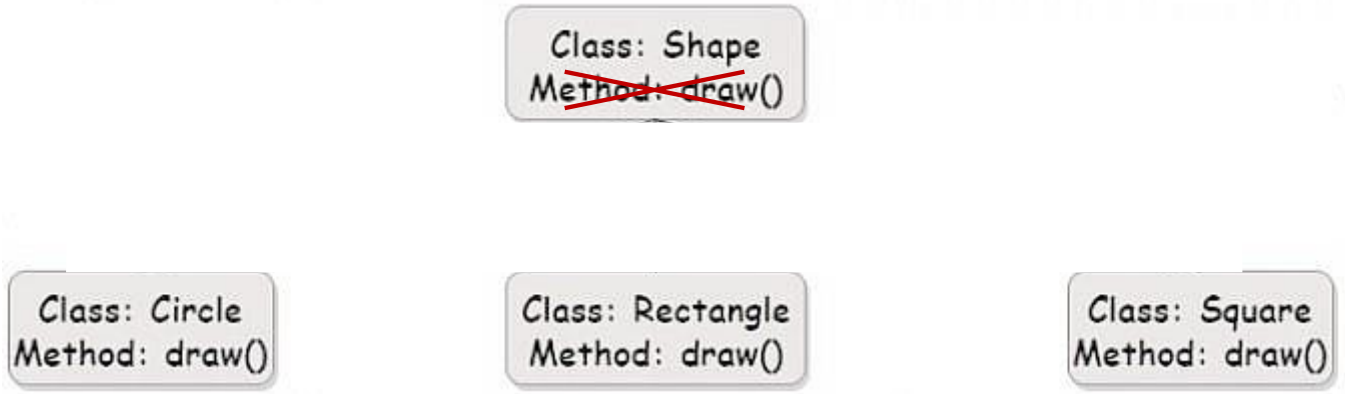
```
def Circle(Shape):
    ...

def Rectangle(Shape):
    ...

def Square(Shape):
    ...
```

Πολυμορφισμός και κληρονομικότητα στην πράξη

(συνέχεια του προηγούμενου παραδείγματος)

Οι θυγατρικές κλάσεις `Circle, Rectangle, Square` κλπ. πιθανόν να έχουν και άλλες πολυμορφικές μεθόδους όπως πχ. `draw(), get_area()` ή `get_perimeter()`

Οι πολυμορφικές αυτές μέθοδοι έχουν *ίδιο όνομα* και *ίδια λειτουργικότητα σε* όλες τις κλάσεις, όμως *δεν μπορούν να οριστούν από κοινού με τις υπόλοιπες* στην γονεϊκή κλάση καθώς *υλοποιούνται με διαφορετικό τρόπο σε κάθε κλάση (διαφορετικοί τύποι υπολογισμών ανάλογα με το σχήμα)*



Τέλος Διάλεξης

Ερωτήσεις;

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