Τμήμα Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Μάθημα: **Τεχνικές Προγραμματισμού Υπολογιστών. (Εργαστηριακό μάθημα)**
Καθηγητής : Πεφάνης Ευάγγελος

4) Εργαστηριακές σημειώσεις στην γλώσσα προγραμματισμού Python.

Αντικειμενοστραφής Προγραμματισμός (Object-Oriented Programming)

Ο Αντικειμενοστραφής Προγραμματισμός (Object-Oriented Programming ή OOP για συντομία), όπως λέει και το όνομά του συνδέεται άμεσα με τα αντικείμενα και κατά συνέπεια, όπως θα δούμε και στη συνέχεια, και με τις κλάσεις. Τι είναι όμως ο Αντικειμενοστραφής Προγραμματισμός;

Κατ' ουσίαν, ο OOP είναι ένας τρόπος οργάνωσης των προγραμμάτων που γράφουμε. Ο τρόπος αυτός οργάνωσης, δεν είναι φυσικά ούτε μοναδικός, ούτε καν ο βέλτιστος. Άλλοι τρόποι οργάνωσης είναι ο *διαδικαστικός* (procedural ή imperative) και ο *συναρτησιακός* (functional). Συνηθίζεται, αυτοί οι τρόποι οργάνωσης ή τεχνικές οργάνωσης των προγραμμάτων να ονομάζονται προγραμματιστικά παραδείγματα (programming paradigms).

Η επιλογή του προγραμματιστικού παραδείγματος το οποίο θα χρησιμοποιήσουμε εξαρτάται από το πρόβλημα το οποίο καλούμαστε να επιλύσουμε, αλλά και από τη γλώσσα προγραμματισμού την οποία χρησιμοποιούμε. Δεν επιτρέπουν όλες οι γλώσσες προγραμματισμού τη χρήση όλων των προγραμματιστικών παραδειγμάτων. Π.χ. υπάρχουν γλώσσες που επιτρέπουν τη δημιουργία μόνο διαδικαστικών προγραμμάτων όπως η Fortran. Υπάρχουν άλλες που επιτρέπουν τη δημιουργία μόνο (ή κυρίως) συναρτησιακών προγραμμάτων, όπως η Lisp και οι πολλές παραλλαγές της. Ενώ υπάρχουν και γλώσσες που επιτρέπουν, με περισσότερη η λιγότερη ευκολία, να εφαρμόσεις περισσότερα από ένα προγραμματιστικά παραδείγματα. Η Python ανήκει σε αυτήν την κατηγορία καθώς σου επιτρέπει να γράψεις κώδικα που χρησιμοποιεί και τα τρία προγραμματιστικά παραδείγματα.

Το πιο χαρακτηριστικό ίσως παράδειγμα προβλήματος που προσφέρεται για λύση μέσω αντικειμενοστραφούς προγραμματισμού είναι ο σχεδιασμός GUI.

Τα πάντα είναι αντικείμενα (Everything is an object)

Μία από τις πλέον συνήθεις εκφράσεις στα κείμενα που αναφέρονται στην Python είναι ότι "*τα πάντα είναι αντικείμενα*" (everything is an object). Ευθύς αμέσως φυσικά, μέσα από την φράση αυτή ξεπηδούν δύο ερωτήματα:

- Τι είναι τα αντικείμενα (objects);
- Πως δημιουργούνται τα αντικείμενα;

Το πρώτο ερώτημα θα το απαντήσουμε στην επόμενη ενότητα. Το δεύτερο ερώτημα μπορεί όμως να απαντηθεί πολύ πολύ σύντομα. Τα *αντικείμενα* (objects) δημιουργούνται από τις *κλάσεις* (classes). Για την ακρίβεια μάλιστα, οι κλάσεις ορίζονται ως *εργοστάσια αντικειμένων* (object factories). Είναι δηλαδή, τα στοιχεία εκείνα της γλώσσας, τα οποία κατασκευάζουν αντικείμενα (objects). Προκειμένου βέβαια να γίνει καλύτερα κατανοητό αυτό θα πρέπει πρώτα να δούμε τι είναι τα αντικείμενα.

Αντικείμενα (object)

Προκειμένου να εξηγήσουμε τον όρο αντικείμενο (object) στην Python συχνά βοηθάει να σκεφτόμαστε τα αντικείμενα (objects) ως κάτι ανάλογο με αυτό που ονομάζουμε στη γραμματική της φυσικής γλώσσας "ουσιαστικό".

Για να γίνει πιο σαφές αυτό ας σκεφτούμε ορισμένα ουσιαστικά. *Καρέκλα*, *τραπέζι*, *σκύλος*, *γάτα*, *κύκλος*, *ορθογώνιο*, *οδηγός* είναι ορισμένα που πιθανά έρχονται στο μυαλό. Αν προσπαθήσουμε να δούμε το κοινό σημείο όλων των παραπάνω θα δούμε ότι πρόκειται για:

Οντότητες (έμψυχες ή άψυχες) οι οποίες έχουν συγκεκριμένες **ιδιότητες** ή/και μπορούν να **εκτελούν συγκεκριμένες ενέργειες**.
Ας δούμε μερικά παραδείγματα:

- Μία *καρέκλα* έχει τις ιδιότητες `χρώμα`, `ύψος`, `υλικό` κτλ.
- Ένα *ορθογώνιο* αντίστοιχα έχει τις ιδιότητες `πλάτος`, `ύψος`, `εμβαδόν`, `περίμετρος` κτλ.
- Μία *γάτα* έχει τις ιδιότητες `όνομα`, `ηλικία`, `φύλλο` κτλ αλλά επίσης μπορεί και να εκτελεί διάφορες ενέργειες όπως π.χ. να `νιαουρίσει`, να `τρέξει`, να `περπατήσει`, να `σκαρφαλώσει`, να `φάει` κτλ.
- Ένας *οδηγός* μπορεί να προβεί σε διάφορες ενέργειες, όπως πχ να `στρίψει`, να `φρενάρει`, να `επιταχύνει`, να `βάλει μπροστά τη μηχανή` κτλ.

Όπως γίνεται σαφές από τα παραπάνω, σε όρους φυσικής γλώσσας, οι ιδιότητες των αντικειμένων είναι άλλα ουσιαστικά, ενώ οι ενέργειες που μπορούν να εκτελέσουν είναι ρήματα.

Την αντιστοίχιση αυτή μεταξύ ιδιοτήτων - ουσιαστικών και ενεργειών - ρημάτων καλό είναι να την κρατήσουμε στο νου μας καθώς θα μας χρειαστεί αργότερα όταν θα δούμε πως μπορούμε να χρησιμοποιήσουμε τις κλάσεις.
Ο παραπάνω “ορισμός” των αντικειμένων είναι φυσικά πολύ γενικός και δύσκολα θα άντεχε σε ενδελεχή εξέταση από έναν φιλόλογο. Παρόλα αυτά, είναι ένας ορισμός που μας βολεύει ιδιαίτερα όταν επιστρέφουμε στον κόσμο του Αντικειμενοστραφούς Προγραμματισμού και αυτό γιατί τα αντικείμενα (objects) είναι εκείνες οι “οντότητες” οι οποίες μας επιτρέπουν να περιγράφουμε κάθε τι που έχει ιδιότητες και μπορεί να εκτελεί ενέργειες. Με λίγα λόγια δηλαδή, μέσω των αντικειμένων μπορούμε να περιγράψουμε πρακτικά σχεδόν οτιδήποτε συναντάμε στον υλικό κόσμο.

Χρησιμοποιώντας την ορολογία της Python, οι ιδιότητες ενός αντικειμένου ονομάζονται **attributes**, ενώ οι ενέργειές που μπορεί να εκτελέσει ονομάζονται **methods** (μέθοδοι).

Κλάσεις (Classes)

Η σύνταξη των κλάσεων στην Python είναι αρκετά απλή. Παρόλα αυτά, σε πρώτη φάση θα χρησιμοποιήσουμε μια ακόμη πιο απλοποιημένη σύνταξη (η οποία όμως μοιάζει αρκετά με Python) προκειμένου να καταλάβουμε ευκολότερα ορισμένες βασικές έννοιες του Αντικειμενοστραφούς Προγραμματισμού.

Την κανονική σύνταξη της Python θα την δούμε στη συνέχεια.
Μια κλάση δεν είναι τίποτα άλλο παρά ένας **ορισμός**. Αυτό που ορίζει είναι το ποιες ιδιότητες (attributes) και ποιες μεθόδους (methods) θα έχει ένα αντικείμενο.

Ας δούμε ένα παράδειγμα. Σε πρώτη φάση ας φτιάξουμε μια κλάση που ορίζει ένα *Αντικείμενο Γάτας*, (με άλλα λόγια δηλαδή μια Γάτα!):

```
class Cat():
    name
    age
    sex

    def eat():
        print("%s is eating" % name)

    def sleep():
        print("%s is sleeping" % name)
```

Η πρώτη γραμμή είναι αυτή στην οποία ορίζεται το όνομα της κλάσης. Προσέξτε την παρουσία των παρενθέσεων. Τη χρήση τους θα τη δούμε στη συνέχεια. Οι γραμμές που ακολουθούν, αποτελούν το *σώμα της κλάσης* (class body). Στη συγκεκριμένη περίπτωση, ορίσαμε ότι οι γάτες που θα δημιουργηθούν από την κλάση αυτή, θα έχουν 3 ιδιότητες (`όνομα, ηλικία` και `φύλο`) και θα μπορούν να εκτελούν 2 ενέργειες (θα έχουν 2 μεθόδους δηλαδή) , να `τρώνε` και να `κοιμούνται`.

Όλες οι γάτες που θα δημιουργηθούν από την κλάση αυτή θα έχουν μόνο αυτές τις ιδιότητες (attributes) και θα μπορούν να εκτελούν μόνο τις συγκεκριμένες ενέργειες.

Θα μπορούσαμε φυσικά να κάνουμε την κλάση μας πολύ πιο σύνθετη. Μια γάτα εξάλλου είναι ένας πολύ σύνθετος οργανισμός... Αλλά εδώ για διδακτικούς λόγους προτιμήσαμε να κρατήσουμε τα πράγματα απλά. Στην πράξη, συνήθως, οι κλάσεις μας θα είναι αρκετά πιο σύνθετες.
Τώρα θα χρησιμοποιήσουμε την κλάση που ορίσαμε παραπάνω για να δημιουργήσουμε δύο νέες γάτες (δηλαδή δύο νέα *αντικείμενα Γάτας*). Τη γάτα του Joe, μια θηλυκή γάτα δύο ετών που τη λένε “Kitty” και τη γάτα της Mary, μια αρσενική γάτα οκτώ ετών που τη λένε “Paul”:

```
joes_cat = Cat(name="Kitty", age=2, sex="female")
marys_cat = Cat(name="Paul", age=8, sex="male")
```

Η σύνταξη για τη δημιουργία των νέων *αντικειμένων Γάτας* είναι πολύ απλή. Απλά χρησιμοποιούμε το όνομα της κλάση και μέσα στις παρενθέσεις δίνουμε τις τιμές των ιδιοτήτων (attributes) της κλάσης. Με τον τρόπο αυτό, δίνοντας δηλαδή διαφορετικές τιμές στα attributes της κλάσης, μπορούμε να δημιουργήσουμε πολλές, διαφορετικές μεταξύ τους γάτες. Δεν υπάρχει κανένας περιορισμός στον αριθμό των νέων αντικειμένων που θα δημιουργήσουμε από μία κλάση. Μπορούμε να δημιουργήσουμε όσα νέα αντικείμενα θέλουμε.

Όλα τα νέα αντικείμενα θα έχουν ακριβώς τις ίδιες ιδιότητες (attributes) και τις ίδιες μεθόδους. Οι τιμές των ιδιοτήτων τους όμως θα είναι διαφορετικές μεταξύ τους. Μπορούμε φυσικά να δώσουμε ακριβώς τις ίδιες τιμές στις ιδιότητες, οπότε θα δημιουργηθούν δύο όμοια αντικείμενα, αλλά συνήθως δεν έχουμε λόγο να κάνουμε κάτι τέτοιο.

Υπενθυμίζουμε ότι προηγουμένως ορίσαμε τις κλάσεις σαν *εργοστάσιο αντικειμένων* (factory object). Ακριβώς όπως ένα εργοστάσιο που φτιάχνει καρέκλες μπορεί να κατασκευάσει καρέκλες διαφορετικού σχεδίου, διαστάσεων και χρώματος σε όποια ποσότητα επιθυμεί, έτσι και οι κλάσεις μπορούν να κατασκευάσουν αντικείμενα με διαφορετικές ιδιότητες σε όποια ποσότητα επιθυμούμε.
Στην ορολογία του Αντικειμενοστραφούς Προγραμματισμού, όλα τα νέα αντικείμενα που δημιουργούνται από μία κλάση ονομάζονται *στιγμιότυπα* (instances).

Όπως μία φωτογραφία ενός αθλητή που τρέχει αποτελεί την απεικόνιση μίας μόνο από όλες τις θέσεις που πέρασε κατά τη διάρκεια του αγώνα του, έτσι και μία instance αποτελεί μία μόνο αποτύπωση όλων των δυνατών συνδυασμών που μπορούν να έχουν οι ιδιότητες μιας κλάσης.

Πρόσβαση σε ιδιότητες και μεθόδους

Προκειμένου να αποκτήσουμε πρόσβαση στις ιδιότητες και στις μεθόδους των αντικειμένων που δημιουργήσαμε, χρησιμοποιούμε το λεγόμενο *dot notation*. Πχ. για να εκτυπώσουμε στην οθόνη το όνομα της κάθε γάτας θα δίναμε:

```
print(joes_cat.name)
print(marys_cat.name)
```

Το αποτέλεσμα της εκτέλεσης του παραπάνω κώδικα θα είναι:

```
Kitty
Paul
```

Αντίστοιχα για να πούμε στη γάτα του Joe να κοιμηθεί και στη γάτα της Mary να φάει θα το κάναμε ως εξής:

```
joes_cat.sleep()
```

```
marys_cat.eat()
```
Το αποτέλεσμα της εκτέλεσης του παραπάνω κώδικα θα είναι:

```
Kitty is eating.
Paul is sleeping.
```

Περισσότερα για τις Μεθόδους

Αν προσέξουμε τον ορισμό των μεθόδων μέσα στο σώμα της κλάσης, θα δούμε ότι δε διαφέρουν από τον ορισμό των συναρτήσεων. Μία μέθοδος δεν είναι τίποτα άλλο παρά μία συνάρτηση που *ανήκει* σε ένα αντικείμενο. Όλα όσα ξέρουμε για τις τυπικές συναρτήσεις της Python ισχύουν και εδώ. Μπορούμε να δώσουμε έξτρα ορίσματα (arguments) σε μία μέθοδο κτλ. Πχ ας ξαναορίσουμε την κλάση της Γάτας, βάζοντας αυτή τη φορά υποχρεωτικά ορίσματα στις μεθόδους της:

```
class Cat():
    name
    age
    sex

    def eat(food):
        print("%s is eating %s." % (name, food))

    def sleep(time):
        print("%s is sleeping for %d minutes." % (name, time))
```
Αυτή τη φορά λοιπόν, θα μπορούμε να πούμε στις γάτες που θα δημιουργήσουμε τι να φάνε και πόση ώρα να κοιμηθούνε. Πχ με τον ακόλουθο κώδικα, θα δημιουργήσουμε μια γάτα στην οποία θα πούμε πρώτα να φάει ποντίκια, στη συνέχεια να κοιμηθεί για 120 λεπτά και μετά να πιει γάλα:

```
alley_cat = Cat(name="Leo", age=4, sex="male")

alley_cat.eat("mice")
alley_cat.sleep(120)
alley_cat.eat("milk")
```
Το αποτέλεσμα της εκτέλεσης του παραπάνω κώδικα θα είναι:

```
Leo is eating mice.
Leo is sleeping for 120 minutes.
Leo is eating milk.
```

Το ότι οι μέθοδοι είναι ακριβώς ίδιες με τις συναρτήσεις δεν είναι απόλυτα ακριβές. Στην ψευδογλώσσα που χρησιμοποιούμε, ισχύει μεν κάτι τέτοιο αλλά στην Python οι μέθοδοι έχουν μία διαφορά από τις συναρτήσεις. Για την ώρα δεν είναι σημαντικό. Απλά κρατήστε το στο μυαλό σας.

Κληρονομικότητα (Inheritance)

Ίσως η πλέον κεφαλαιώδης έννοια του Αντικειμενικοστραφούς Προγραμματισμού είναι η **κληρονομικότητα** (inheritance).

Με τον όρο κληρονομικότητα, εννοούμε τη δυνατότητα που έχει μια κλάση να *κληρονομεί* όλες τις ιδιότητες και τις μεθόδους μιας άλλης κλάσης. Χρησιμοποιώντας λίγο πιο επίσημη ορολογία, λέμε ότι η κλάση που ορίζεται πρώτη είναι η *βασική κλάση* (base class) και η κλάση που κληρονομεί τη βασική κλάση ονομάζεται *παράγωγη κλάση* (derived class). Εναλλακτικά οι βασικές κλάσεις ονομάζονται και *υπερκλάσεις* ενώ οι παράγωγες κλάσεις ονομάζονται *υποκλάσεις*.

Ας δούμε ένα παράδειγμα:

```
class BaseClass():
    attr1
    attr2

    def method1():
        print("You just called method1.")

    def method2():
        print("You just called method2.")

class DerivedClass(BaseClass):
    pass
```
Η υπερκλάση (`BaseClass`) δεν έχει καμία διαφορά από τις κλάσεις που έχουμε δει ως τώρα. Η υποκλάση (`DerivedClass`) όμως χρησιμοποιεί ελαφρά διαφορετική σύνταξη. Αντί οι παρενθέσεις της πρώτης γραμμής του ορισμού της να είναι κενές:

```
class DerivedClass():
    pass
```
περιέχουν το όνομα της υπερκλάσης:

```
class DerivedClass(BaseClass):
    pass
```
Αυτή η μικρή διαφορά στη σύνταξη κάνει όλη τη διαφορά! Με τον τρόπο αυτό, οι instances της `DerivedClass` αποκτούν όλες τις ιδιότητες και όλες τις μεθόδους που ορίστηκαν στην `BaseClass`! Ας δούμε ένα παράδειγμα. Θα δημιουργήσουμε δύο instances της `DerivedClass` και θα καλέσουμε τις μεθόδους που έχουν οριστεί στην `BaseClass`:

```
# Class Instantiation
instance1 = DerivedClass(attr1="value1", attr2="value2")
instance2 = DerivedClass(attr1="value3", attr2="value4")

instance1.method1()
instance2.method2()
```
Το αποτέλεσμα του παραπάνω κώδικα θα είναι:

```
You just called method1.
You just called method2.
```
Όπως βλέπουμε, αν και το class body της `DerivedClass` είναι κενό, παρόλα αυτά, τα στιγμιότυπά της, τα αντικείμενα δηλαδή που δημιουργούνται από την κλάση, μπορούν να καλούν κανονικά τις μεθόδους της `BaseClass`.

Δηλαδή, προσθέτοντας το όνομα της υπερκλάσης στον ορισμό της κλάσης, (μέσα στις παρενθέσεις της πρώτης γραμμής) ο ορισμός της `DerivedClass` γίνεται ισοδύναμος με τον ακόλουθο:

```
class DerivedClass():
    attr1
    attr2

    def method1():
        print("You just called method1.")

    def method2():
        print("You just called method2.")
```
Επειδή το παραπάνω είναι πολύ γενικό, ας δούμε και ένα πιο χειροπιαστό παράδειγμα. Θα ορίσουμε λοιπόν μία κλάση η οποία θα δημιουργεί Ζώα:

```
class Animal():
    species
    race
    sex

    def speak():
        print("I don't know what to say...")
```
Στη συνέχεια θα ορίσουμε δύο κλάσεις που *εξειδικεύουν* την κλάση Ζώου:

```
class Cat(Animal):
    def speak():
        print("Meow!")

class Dog(Animal):
    def speak():
        print("Woof!")
```
Οι κλάσεις *Σκύλου* και *Γατας* κληρονομούν τις ιδιότητες και τις μεθόδους της κλάσης *Ζώου*. Προσοχή στη μέθοδο `speak()` όμως! Όπως βλέπουμε και οι δύο υποκλάσεις **ξαναορίζουν** την μέθοδο που κληρονόμησαν από την υπερκλάση τους. Αυτό είναι μία από τις βασικότερες τεχνικές της Κληρονομικότητας (Inheritance). Οι υποκλάσεις δεν κληρονομούν απλά ιδιότητες (attributes) και μεθόδους (methods), αλλά μπορούν να ξαναορίσουν τις μεθόδους που κληρονόμησαν, **εξειδικεύοντάς** τις.

Ας το δούμε και στην πράξη:

```
# Class Instances
my_dog = Dog()
my_cat = Cat()

my_dog.speak()
my_cat.speak()
```
Το αποτέλεσμα του παραπάνω κώδικα είναι:

```
Woof!
Meow!
```
Φυσικά οι υποκλάσεις, εκτός από το να ξαναορίσουν τις μεθόδους που κληρονομούν, μπορούν να ορίσουν και νέες μεθόδους. Παράδειγμα δε θα δούμε για αυτό γιατί είναι μάλλον απλό.

Απλή vs Πολλαπλή Κληρονομικότητα (Single vs Multiple Inheritance)

Αργά ή γρήγορα (μάλλον γρήγορα) θα ακούσετε τον όρο *Πολλαπλή Κληρονομικότητα* (Multiple Inheritance). Η κληρονομικότητα που έχουμε δει έως τώρα είναι η λεγόμενη *Απλή Κληρονομικότητα* (Single Inheritance). Η διαφορά μεταξύ της απλής και της πολλαπλής κληρονομικότητας έγκειται στον αριθμό των υπερκλάσεων που έχει μία συγκεκριμένη υποκλάση. Αν κληρονομεί από μία μόνο υπερκλάση τότε μιλάμε για απλή κληρονομικότητα. Αν κληρονομεί από δύο ή περισσότερες υπερκλάσεις, τότε μιλάμε για πολλαπλή.

Ένα παράδειγμα πολλαπλής κληρονομικότητας είναι το ακόλουθο:

```
class BaseClass1():
    pass

class BaseClass2():
    pass

class DerivedClass(BaseClass1, BaseClass2):
    pass
```

Η πολλαπλή κληρονομικότητα είναι περίπλοκη. Το παράδειγμά μας δεν αναδεικνύει τη δυσκολία της, αλλά πιστέψτε με, δεν είναι εύκολο να την κάνεις να συμπεριφερθεί σωστά. Η χρήση της πολλαπλής κληρονομικότητας είναι συνήθως ένδειξη κακού design. Αυτός είναι και ο λόγος που υπάρχουν πολλές γλώσσες προγραμματισμού που υποστηρίζουν μόνο απλή κληρονομικότητα (πχ Java). Στην πλειοψηφία των περιπτώσεων, υπάρχει απλούστερος τρόπος να κάνουμε αυτό που θέλουμε από το να καταφύγουμε στην πολλαπλή κληρονομικότητα. Καλό είναι να την αποφεύγετε.
Στο σημείο αυτό, οφείλουμε να διασαφηνίσουμε ότι μία αλληλουχία κλάσεων που διαδοχικά κληρονομούν η μία την άλλη **δεν** είναι πολλαπλή κληρονομικότητα. Παραδείγματος χάρη το ακόλουθο παράδειγμα είναι απόλυτα τυπική απλή κληρονομικότητα:

```
class Animal():
    pass

class Mamal(Animal):
    pass

class Feline(Mamal):
    pass

class Cat(Feline):
    pass
```

Θα μπορούσαμε φυσικά να έχουμε πολύ περισσότερες από τέσσερις κλάσεις. Τα γενεαλογικά δέντρα του ζωικού και του φυτικού βασιλείου αποτελούν πολύ χαρακτηριστικά παραδείγματα τέτοιου είδους αλληλουχιών.

Πότε χρησιμοποιούμε την Κληρονομικότητα;

Το θεμελιώδες αυτό ερώτημα, ευτυχώς, έχει πάρα πολύ απλή απάντηση. Η (απλή) *Κληρονομικότητα* χρησιμοποιείται όταν έχουμε μια ακολουθία (η ιεραρχία αν προτιμάτε) αντικειμένων τα οποία πηγαίνουν από το γενικότερο στο ειδικότερο. Το παράδειγμα με της προηγούμενης ενότητας (Ζώα ⇒ Θηλαστικά ⇒ Αιλουροειδή ⇒ Γάτες) είναι πολύ χαρακτηριστικό.

Αν μελετήσουμε τις σχέσεις μεταξύ των διαδοχικών κλάσεων θα δούμε ότι η “κάθε υποκλάση **είναι** η υπερκλάση της”. Δηλαδή:

Μία *Γάτα* είναι *Αιλουροειδές*.

Ένα *Αιλουροειδές* είναι *Θηλαστικό*.

Ένα *Θηλαστικό* είναι *Ζώο*.

Η σχέση αυτή ισχύει όχι μόνο για την αμέσως ανώτερη υπερκλάση, αλλά για κάθε υπερκλάση. Δηλαδή:

Μία *Γάτα* είναι *Αιλουροειδές*.

Μια *Γάτα* είναι και *Θηλαστικό*.

Μια *Γάτα* είναι και *Ζώο*.

Κάθε φορά που έχουμε μια παρόμοια σχέση μεταξύ αντικειμένων, δηλαδή μπορούμε να πούμε ότι **κάτι είναι κάτι άλλο** (is-a relationship) τότε μπορούμε/πρέπει να χρησιμοποιήσουμε κληρονομικότητα. Αυτό όμως δεν απαντάει στο *γιατί* να το κάνουμε αυτό... Γιατί να μην ορίσουμε κατευθείαν την κλάση της *Γάτας*; Γιατί να μπερδευόμαστε με κλάσεις, υποκλάσεις κτλ κτλ;

Η απάντηση είναι η εξής. Αν έχουμε να ασχοληθούμε μόνο με *αντικείμενα Γάτας* τότε δεν μας χρειάζονται όλα αυτά. Αν όμως έχουμε να κατασκευάσουμε *Γάτες, Λιοντάρια, Τίγρεις* κτλ τότε τα οφέλη αρχίζουν να φαίνονται. Ορίζουμε αρχικά τα κοινά στοιχεία όλων αυτών των κλάσεων στην υπερκλάση *Αιλουροειδές*, την κληρονομούμε και την εξειδικεύουμε καταλλήλως στις υποκλάσεις.

Αν μάλιστα εκτός από *Αιλουροειδή* έχουμε να κατασκευάσουμε και αντικείμενα *Σκύλου*, *Αλεπούς*, *Λύκου*, αλλά και *Αρκούδας*, *Αλόγου*, *Φώκιας* κτλ τότε η κλάση *Θηλαστικό* κατευθείαν αποκτάει νόημα.

Με τον ίδιο τρόπο, αν στο πρόγραμμά μας χρειαζόμαστε και *Ψάρια* ή *Έντομα* κτλ τότε η υπερκλάση *Ζώο* βρίσκει και αυτή τη θέση της. Η κάθε μία από τις κλάσεις *Ψάρι*, *Έντομο* κτλ θα έχει φυσικά το δικό της ιεραρχικό δέντρο, στο οποίο θα εξειδικεύεται καταλλήλως.

Συνοψίζοντας, όταν για δύο αντικείμενα *A* και *B* μπορούμε να πούμε ότι **το A είναι το B**, τότε μπορούμε να χρησιμοποιήσουμε κληρονομικότητα.

Σύνθεση (Composition)

Τι είναι η Σύνθεση;

Η *Σύνθεση* (Composition) είναι η δεύτερη τεχνική του OOP που θα αναπτύξουμε.

Όπως θα θυμάστε, είχαμε αναφέρει προηγουμένως ότι υπάρχει μια αντιστοιχία μεταξύ των "ιδιοτήτων" (attributes) μιας κλάσης και των "ουσιαστικών" της φυσικής γλώσσας. Εκτός από αυτό όμως, αναφέραμε ότι και τα ίδια τα αντικείμενα (objects) είναι "ουσιαστικά".

Συνδυάζοντας τις δύο παραπάνω προτάσεις προκύπτει ότι:

οι ιδιότητες (attributes) ενός αντικειμένου είναι και αυτές με τη σειρά τους αντικείμενα!
Η παρατήρηση αυτή είναι η βασική ιδέα πίσω από την τεχνική της Σύνθεσης.

Η Σύνθεση στην πράξη

Ας δούμε ένα παράδειγμα. Ας σκεφτούμε μια μηχανή αυτοκινήτου. Η μηχανή αυτή αποτελείται από πολλά εξαρτήματα. Κύλινδροι, βαλβίδες, πιστόνια, μπουζί είναι μερικά μόνο από αυτά. Η σχέση που συνδέει την μηχανή με τα εξαρτήματά της είναι η εξής:

Η *Μηχανή* **έχει** τους *κυλίνδρους*.

Η *Μηχανή* **έχει** τις *βαλβίδες*.

Η *Μηχανή* **έχει** τα *πιστόνια*.

Δηλαδή, η *μηχανή* **έχει** *εξαρτήματα-ιδιότητες*, το καθένα από τα οποία είναι ένα ξεχωριστό αντικείμενο (object). Στην ορολογία του OOP αυτού του είδους η σχέση ονομάζεται "**has-a relationship**". Ας το δούμε και στην πράξη:

```
class Piston():
    pass

class Valves():
    pass

class Cylinders():
    pass

class Engine():
    pistons = Pistons()
    valves = Valves()
    cylinders = Cylinders()
```

Όπως βλέπουμε οι *ιδιότητες* (attributes) της *Μηχανής* **είναι** *instances* μιας άλλης κλάσης. Στο παράδειγμα αυτό οι κλάσεις *Πιστονιού*, *Βαλβίδας* και *Κυλίνδρου* δεν έχουν δικές τους ιδιότητες/μεθόδους, αυτό όμως έχει γίνει

καθαρά για λόγους απλότητας. Δεν υπάρχει κανένας τέτοιου τύπου περιορισμός. Ας δούμε ένα ακόμα παράδειγμα:

```
class Tire():
    size
    max_pressure

class Engine():
    cubic_capacity
    engine_type

class Car():
    model
    color

    tires = Tire(size=18, max_pressure=30)
    engine = Engine(type="V8", cubic_capacity=1800)
```

Λογικά πλέον δεν χρειάζονται πολλές εξηγήσεις. Ένα αυτοκίνητο *έχει* ελαστικά και μηχανή. Ορίζουμε αρχικά λοιπόν τις κλάσεις *Ελαστικό* και *Μηχανή* με τις κατάλληλες ιδιότητες και μεθόδους. Στη συνέχεια ορίζουμε την κλάση *Αυτοκίνητο* η οποία έχει ως ιδιότητες (attributes) *αντικείμενα Ελαστικού* και *αντικείμενα μηχανής*. Πέρα από αυτές τις δύο ιδιότητες, η κλάση μπορεί να έχει και άλλες ιδιότητες όπως *μοντέλο* και *χρώμα*.

Ας φτιάξουμε τώρα μερικά *αντικείμενα Αυτοκινήτου*:

```
my_car = Car(model="Ford Escort", color="Blue")
your_car = Car(model="Ferrari Testarossa", color="Red")
```

Στο παράδειγμα αυτό, οι instances των ελαστικών και της μηχανής δημιουργούνται μέσα στο σώμα της κλάσης *Αυτοκίνητο*. Ως αποτέλεσμα αυτού όλα τα *αντικείμενα Αυτοκινήτου* θα έχουν την ίδια μηχανή και τα ίδια ελαστικά. Αν θέλουμε να φτιάξουμε αυτοκίνητα με διαφορετική μηχανή και ελαστικά πως θα το πετύχουμε; Αυτό το ερώτημα θα το απαντήσουμε αργότερα, καθώς είναι καλύτερα να το εξηγήσουμε χρησιμοποιώντας την κανονική σύνταξη της Python. Για την ώρα λοιπόν, δώστε περισσότερη προσοχή στην *Ιδέα* της *Σύνθεσης* και λιγότερο στην υλοποίηση

Κλάσεις στην Python

Στον προγραμματισμό γενικά προσπαθούμε να αποφεύγουμε τις επαναληπτικές εργασίες. Προσπαθούμε να γράφουμε κώδικα μία φορά και να τον χρησιμοποιούμε ξανά. Η αποφυγή της επανάληψης κώδικα επιτυγχάνεται με την χρήση συναρτήσεων. Οι συναρτήσεις τυποποιούν λειτουργίες. Όποτε το πρόγραμμα μας πρέπει να εκτελέσει αυτές τις λειτουργίες, καλούμε τις συναρτήσεις αυτές, περνώντας τις κατάλληλες παραμέτρους. Η χρήση συναρτήσεων δεν μας δίνει την δυνατότητα να αποθηκεύσουμε μέσα τους πληροφορίες. Οι τιμές των τοπικών τους μεταβλητών χάνονται μετά την επιστροφή εκτός αν περαστούν σαν παράμετροι, ή τις επιστρέψουν, οπότε η καλούσα συνάρτηση μπορεί να κρατήσει τις τιμές.

Με τον αντικειμενοστραφή προγραμματισμό και την χρήση κλάσεων δίνεται η δυνατότητα διαχείρισης δεδομένων και συναρτήσεων με πολύ ολοκληρωμένο τρόπο.

Στον αντικειμενοστραφή προγραμματισμό προτεραιότητα δίνεται στα δεδομένα και όχι στις συναρτήσεις όπως γίνεται στον διαδικαστικό προγραμματισμό (Procedural Programming).

Η Python, σε αντίθεση με τη Java επιτρέπει δυναμικούς τύπους.

Δημιουργία κλάσεων

Μία κλάση είναι ένα πρότυπο. Δεν έχει υπόσταση από μόνη της. Είναι όπως τα σχέδια κατασκευής για αντικείμενα. Περιγράφει πως μπορεί να λειτουργήσει ένα ή περισσότερα αντικείμενα, που έχει ή έχουν δημιουργηθεί υλοποιώντας την κλάση.

Παράδειγμα 1: Κώδικας για τον ορισμό μίας κλάσης

```
# Defining a class

class class_name:
        [statement 1]
        [statement 2]
        [statement 3]
        [etc.]
```

Οι κλάσεις περιέχουν:

- πεδία (δεδομένα)
- μεθόδους (συναρτήσεις)

Παράδειγμα 2: Κώδικας για τη δημιουργία της κλάσης Shape:

```
#An example of a class

class Shape:
    def __init__(self,x,y):
        self.x = x
        self.y = y
        description = "This shape has not been described yet"
        author = "Nobody has claimed to make this shape yet"

    def area(self):
        return self.x * self.y

    def perimeter(self):
        return 2 * self.x + 2 * self.y

    def describe(self,text):
        self.description = text

    def authorName(self,text):
        self.author = text

    def scaleSize(self,scale):
        self.x = self.x * scale
        self.y = self.y * scale
```

Έχουμε δημιουργήσει μια περιγραφή ενός σχήματος όπως και τις μεταβλητές και τις ενέργειες που μπορούμε να κάνουμε με το Shape. Όμως δεν έχουμε κατασκευάσει ένα πραγματικό σχήμα, αλλά απλά έχουμε ορίσει την περιγραφή του τι είναι ένα Shape. Το Shape έχει πλάτος (x), ύψος (y), και μας δίνει την δυνατότητα να υπολογίσουμε το εμβαδόν και την περίμετρο του.

Η συνάρτηση _init__ εκτελείται όταν δημιουργήσουμε ένα αντικείμενο (instance ή “στιγμιότυπο”) της Shape, δηλαδή, όταν δημιουργήσουμε ένα πραγματικό Shape, σε αντίθεση με το “σχέδιο” που έχουμε εδώ, τότε η_init εκτελείται αυτόματα.

Το self δείχνει ότι αναφερόμαστε σε στοιχεία της ίδιας της κλάσης. Αποτελεί την πρώτη παράμετρο σε όλες τις λειτουργίες που ορίζονται μέσα σε μια τάξη. Κάθε λειτουργία ή μεταβλητή που δημιουργήθηκε στο πρώτο επίπεδο της εσοχής, δηλαδή γραμμές κώδικα που ξεκινούν ένα tab δεξιά της θέσης όπου βάζουμε την κλάση Shape. Για πρόσβαση σε αυτές τις λειτουργίες και τις μεταβλητές

αλλού μέσα στην τάξη, το όνομά τους θα πρέπει να προηγείται η λέξη self ακολουθούμενη από μία τελεία (π.χ. self.variable_name).

Χρήση μίας κλάσης

Ενώ η κλάση είναι τα σχέδια, ή οι οδηγίες για την κατασκευή αντικειμένων, η δημιουργία των αντικειμένων (instances) είναι αυτή που δεσμεύει μνήμη για την αποθήκευση των πεδίων τους. (Ο κώδικας είναι κοινός για όλα τα αντικείμενα μίας κλάσης).

Παράδειγμα 3: Κώδικας για τη δημιουργία αντικειμένου της κλάσης

```
#creating an object

rectangle = Shape(100, 45)
```

Η συνάρτηση_init_καλείται αυτόματα. Δημιουργούμε ένα αντικείμενο (“στιγμιότυπο”) της κλάσης δίνοντας το όνομά του (σε αυτή την περίπτωση, το Shape) και, στη συνέχεια, σε παρένθεση, οι τιμές για να περάσει στη λειτουργία_init_. Η συνάρτηση init εκτελείται (χρησιμοποιώντας τις παραμέτρους που δόθηκαν σε παρένθεση) και, στη συνέχεια, κατασκευάζει ένα αντικείμενο της κλάσης, η οποία στην περίπτωση αυτή αποδίδεται στη μεταβλητή rectangle.

Το αντικείμενο rectangle αποτελεί αυτοτελή συλλογή μεταβλητών και συναρτήσεων. Με τον ίδιο τρόπο που χρησιμοποιήσαμε την self για να αποκτήσουμε πρόσβαση στις συναρτήσεις και τις μεταβλητές της κλάσης μέσω της self, χρησιμοποιούμε το όνομα που της έχει ανατεθεί τώρα (rectangle) για την πρόσβαση στις λειτουργίες και τις μεταβλητές της κλάσης από έξω από την κλάση.

Παράδειγμα 4: Πρόσβαση χαρακτηριστικών εκτός κλάσης

```
#creating an object
rectangle=Shape(100,45)

#finding the area of your rectangle:
print(rectangle.area())

#finding the perimeter of your rectangle:
print(rectangle.perimeter())

#describing the rectangle
rectangle.describe('A wide rectangle')

#making the rectangle 50% smaller
```

```
rectangle.scaleSize(0.5)

#re-printing the new area of the rectangle
print(rectangle.area())
```

Παράδειγμα 5: Δημιουργία περισσότερων του ενός αντικειμένων

```
long_rectangle = Shape (120,10)
fat_rectangle = Shape (130,120)
```

Κληρονομικότητα

Παράδειγμα 6: Η κλάση Shape μπορεί να χρησιμοποιηθεί σαν βάση για τη δημιουργία παράγωγων κλάσεων με πιο εξειδικευμένα χαρακτηριστικά.

```
#An example of a class class Shape:
    def __init__(self,x,y):
        self.x = x
        self.y = y
        description = "This shape has not been described yet"
        author = "Nobody has claimed to make this shape yet"

    def area(self):
        return self.x * self.y

    def perimeter(self):
        return 2 * self.x + 2 * self.y

    def describe(self,text):
        self.description = text

    def authorName(self,text):
        self.author = text

    def scaleSize(self,scale):
        self.x = self.x * scale
        self.y = self.y * scale
```

Για να ορίσουμε μια νέα κλάση, π.χ. την κλάση Square, δηλαδή τετράγωνο, με βάση την προηγούμενη κλάση Shape, κάνουμε αυτό:

Παράδειγμα 7: Κληρονομικότητα

```
#An inherited class

class Square(Shape):
    def __init__(self,x):
        self.x = x
        self.y = x
```

Ο ορισμός της υποκλάσης γίνεται ακριβώς όπως ορίζεται μια κλάση, με την διαφορά ότι βάζουμε σε παρένθεση μετά το όνομα, το όνομα της μητρικής κλάση από την οποία κληρονομήσαμε. Έτσι έχουμε περιγράψει ένα τετράγωνο πολύ γρήγορα, χρησιμοποιώντας την κλάση Shape. Αυτό συμβαίνει γιατί έχουμε κληρονομήσει τα πάντα, από την κλάση Shape και αλλάξαμε μόνο ότι χρειάζεται να αλλάξει. Σε αυτή την περίπτωση, έχουμε επαναπροσδιορίσει την λειτουργία_init_ του Shape, έτσι ώστε οι τιμές x και y να είναι πάντα ίδιες.

Παράδειγμα 8 - DoubleSquare κατηγορία

```
# The shape looks like this:
# __________
#|    |    |
#|    |    |
#|____|____|

class DoubleSquare(Square):
    def __init__(self,y):
        self.x = 2 * y self.y = y

    def perimeter(self):
        return 2 * self.x + 2 * self.y
```

Στο παράδειγμα αυτό ορίζουμε ξανά την μέθοδο που υπολογίζει την περίμετρο, έτσι ώστε να μπορέσει να υπολογιστεί και η κοινή πλευρά εύκολα.

Πλεονεκτήματα του Αντικειμενοστραφούς προγραμματισμού.

Η απάντηση σε αυτό το ερώτημα δεν είναι και τόσο απλή. Μια σύντομη αναζήτηση στο internet σχετικά με τα πλεονεκτήματα του Αντικειμενοστραφούς Προγραμματισμού (benefits/advantages of Object-Oriented Programming) θα επιστρέψει πλήθος σελίδων που απαντούν στο ερώτημα αυτό. Οι απαντήσεις χωρίζονται σε δύο σκέλη:

- Στα οφέλη σε επίπεδο κώδικα
- Στα οφέλη σε επίπεδο οργάνωσης.

Τα οφέλη της πρώτης κατηγορίας είναι εύκολο να γίνουν αντιληπτά. Χάρις στην κληρονομικότητα αποφεύγουμε να επαναλάβουμε κώδικα (αρχή “Do not Repeat Yourself” - DRY). Αυτό γίνεται γιατί όλα τα αντικείμενα μας μοιράζονται τον ίδιο κώδικα. Οι κοινές μέθοδοι ορίζονται μία φορά στην υπερκλάση και όλες οι υποκλάσεις απλά την κληρονομούν. Με τον τρόπο αυτό μειώνεται το μέγεθος του κώδικα. Λιγότερος κώδικας σημαίνει λιγότερα bugs και πιο κατανοητός κώδικας, δηλαδή κώδικας που είναι πιο εύκολο να συντηρηθεί.

Τα οφέλη της δεύτερης κατηγορίας, η αλήθεια είναι ότι μπορείς να τα εκτιμήσεις, μόνο αφού έχεις ήδη κατανοήσει και χρησιμοποιήσει τις αρχές του OOP στα προγράμματα σου. Εν πάση περιπτώσει προκειμένου να δώσουμε μία απάντηση, θα πρέπει να πούμε ότι ο OOP είναι μία προσέγγιση που μας επιτρέπει να φτάσουμε σε υψηλά επίπεδα **αφαίρεσης** (abstraction) μειώνοντας με αυτόν τον τρόπο την πολυπλοκότητα των προβλημάτων που καλούμαστε να λύσουμε .

Λεξικά

Τα λεξικά είναι συλλογές από δεδομένα, αταξινόμητα, τα οποία αποθηκεύουν ζευγάρια τιμών και χρησιμοποιούν σαν δείκτη την πρώτη τιμή του ζευγαριού.

```
>>> d={}
>>> d['Nikos']='2810-323322'
>>> d['Makis']='2810-226676'
>>> d['Maria']='2810-334229'
>>> d
{'Nikos': '2810-323322', 'Makis': '2810-226676', 'Maria': '2810-
334229'}
>>> print(d['Makis'])
2810-226676
>>> for x in d:
     print(x)

Nikos
Makis
Maria
>>> for x in d:
     print(d[x])

2810-323322
2810-226676
2810-334229
>>> for x, y in d.items():
     print(x, y)

Nikos 2810-323322
Makis 2810-226676
Maria 2810-334229
>>> for x in d.values():
     print(x)

2810-323322
2810-226676
2810-334229
>>> if 'Maria' in d:
     print('Maria is in the d dictionary')
```

```
Maria is in the d dictionary
>>>
```

Παράδειγμα 9: Λεξικό

```
# First, create a dictionary:
dictionary = {}

# Then, create some instances of classes in the dictionary:
dictionary["DoubleSquare1"] = DoubleSquare(5)
dictionary["long_rectangle"] = Shape(600,45)
dictionary["DoubleSquare1"].authorName("The Gingerbread Man")

#You can now use them like a normal class:
print(dictionary["long_rectangle"].area())
print(dictionary["DoubleSquare1"].author)
```