Πληροφορική ΙΙ

Αντικειμενοστραφής Προγραμματισμός με Python

Μιχάλης Δρακόπουλος

Ιούνιος 2019

1 Διαδικαστικός και Αντικειμενοστραφής Προγραμματισμός

Στο μοντέλο του διαδικαστικού προγραμματισμό η υπολογιστική επίλυση ενός προβλήματος επιτυγχάνεται με διάσπαση του αρχικού προβλήματος σε απλούστερα υποπροβλήματα (που και αυτά με τη σειρά τους μπορούν να διασπαστούν περαιτέρω). Η επίλυση του κάθε υποπροβλήματος γίνεται από κάποια *συνάρτηση (διαδικασία)* που δέχεται δεδομένα εισόδου και επιστρέφει στην έξοδο τα δεδομένα που αντιστοιχούν στην επίλυση του συγκεκριμένου υποπροβλήματος που επιλύει. Στο διαδικαστικό μοντέλο η έμφαση είναι στη δημιουργία συναρτήσεων, ενώ τα δεδομένα είναι *ανεξάρτητα* από τις συναρτήσεις. Ένα διαδικαστικό πρόγραμμα είναι μια αλληλουχία συναρτήσεων που καλούνται διαδοχικά: τα δεδομένα εξόδου κάθε συνάρτησης χρησιμοποιούνται ως δεδομένα εισόδου για τις επόμενες συναρτήσεις.

Στον διαδικαστικό προγραμματισμό χρησιμοποιούμε έτοιμους τύπους δεδομένων όπως `int`, `float`, `str`, `list`, κλπ των οποίων τα χαρακτηριστικά είναι οι τιμές τους και οι ενέργειες που κάνουμε με αυτά είναι πράξεις, συγκρίσεις, ταξινομήσεις κ.α.

Για μεγάλα προβλήματα, ο διαχωρισμός των δεδομένων από τις συναρτήσεις που τα διαχειρίζονται μπορεί να οδηγήσει σε πολύπλοκα προγράμματα τα οποία μπορεί να είναι δύσκολο να συντηρηθούν ή να επεκταθούν μελλοντικά. Αυτό είναι κάτι που αντιμετωπίζεται σε μεγάλο βαθμό από το προγραμματιστικό μοντέλο του *αντικειμενοστραφούς προγραμματισμού*.

Στον *αντικειμενοστραφή προγραμματισμό (ΑΣΠ)* η έμφαση είναι στη δημιουργία εξειδικευμένων τύπων δεδομένων (*κλάσεις*) και στην κατασκευή και διαχείριση δεδομένων (*αντικείμενα*) από αυτές. Ένα αντικειμενοστραφές πρόγραμμα προσομοιώνει το πρόβλημα του πραγματικού κόσμου που θέλει να επιλύσει. Κατασκευάζει αντικείμενα που αντιστοιχούν σε απλοποιημένες εκδοχές των (έμψυχων, άψυχων, ιδεατών) αντικειμένων του πραγματικού κόσμου. Όπως και στον πραγματικό κόσμο, τα αντικείμενα κάνουν και δέχονται διάφορες ενέργειες, αλλά αλληλεπιδρούν και μεταξύ τους. Η απλοποίηση έχει να κάνει με τη διατήρηση μόνο των χαρακτηριστικών και των ενεργειών εκείνων που έχουν σχέση με το προς επίλυση πρόβλημα. Αυτή είναι η έννοια της **αφαίρεσης** που είναι μια βασική έννοια του ΑΣΠ. Για παράδειγμα, σε ένα πρόγραμμα διαχείρισης φοιτητών από τη γραμματεία, μας ενδιαφέρουν πληροφορίες (χαρακτηριστικά) όπως το ονοματεπώνυμο του φοιτητή, ο Α.Μ. του, στοιχεία επικοινωνίας κλπ, και ενέργειες όπως εγγραφή σε μάθημα, υπολογισμός βαθμολογίας κλπ. Αντίθετα δεν μας ενδιαφέρουν χαρακτηριστικά όπως το ύψος του, τα χόμπι του κλπ, ούτε το αν παίζει κάποιο μουσικό όργανο ή αν ταξιδεύει συχνά κλπ.

2 Αντικείμενα και Κλάσεις

Αντικείμενα του ιδίου τύπου ανήκουν στην ίδια κλάση και κατασκευάζονται από αυτήν. Οι κλάσεις είναι αφηρημένοι τύποι δεδομένων για τον ορισμό αντικειμένων. Μια βασική έννοια του ΑΣΠ είναι η **ενθυλάκωση**: σε κάθε αντικείμενο ενσωματώνονται τόσο τα δεδομένα του όσο και οι συναρτήσεις που δρουν πάνω σε αυτά τα δεδομένα.

- Τα δεδομένα αντιστοιχούν σε *ιδιοχαρακτηριστικά* του αντικειμένου, και ονομάζονται και *πεδία* στα πλαίσια του ΑΣΠ.
- Οι ενσωματωμένες συναρτήσεις λέγονται *μέθοδοι* και αντιστοιχούν στις ενέργειες που μπορούμε να κάνουμε **με** το αντικείμενο ή **στο** αντικείμενο.

2.1 Δομή μιας κλάσης

Μια κλάση περιέχει τα πεδία και τις μεθόδους που προσδιορίζουν τα αντικείμενά της.

Παράδειγμα η κλάση `Coin` που προσομοιώνει τη ρίψη ενός κέρματος και *ορίζεται* ως εξής:

```
import random

class Coin:
    """Προσομοίωση ρίψης κέρματος"""

    def __init__(self):
        self.sideup = 'Κορώνα'

    def toss(self):
        """Ρίψη του κέρματος"""
        self.sideup = random.choice(['Κορώνα', 'Γράμματα'])

    def get_sideup(self):
        """Εμφάνιση πάνω όψης"""
        return self.sideup
```

Το μοναδικό πεδίο της κλάσης `sideup` αντιστοιχεί στην πάνω όψη του κέρματος. Προφανώς, στα πλαίσια της αφαίρεσης, σε ένα παιχνίδι κορώνα-γράμματα δεν μας ενδιαφέρουν άλλα ιδιοχαρακτηριστικά του κέρματος, όπως η ονομαστική του αξία, ή το υλικό κατασκευής του.

Η `Coin` περιέχει τρεις μεθόδους:

1. Η μέθοδος `__init__` είναι *ειδική μέθοδος αρχικοποίησης* που θα την συναντήσουμε στον ορισμό όλων των κλάσεων που θα εξετάσουμε στη συνέχεια. Εδώ αρχικοποιεί το ιδιοχαρακτηριστικό `sideup` στην τιμή `'Κορώνα'` (φανταστείτε ότι ξεκινάμε με ένα κέρμα πάνω σε ένα τραπέζι που η πάνω όψη του είναι κορώνα). Η `__init__` είναι συνήθως η πρώτη μέθοδος στον ορισμό μιας κλάσης.

   Παρατήρηση: Τα ονόματα κάποιων μεθόδων με *ειδική* σημασία για την Python ξεκινούν και τελειώνουν με 2 κάτω παύλες.

2. Η μέθοδος `toss` που προσομοιώνει μια τυχαία ρίψη.
3. Η μέθοδος `get_sideup` που μας επιτρέπει να εξετάσουμε τι εμφανίζεται στην πάνω όψη του κέρματος.

Η `self` είναι παράμετρος και στις 3 μεθόδους και επίσης χρησιμοποιείται για τον προσδιορισμό του ιδιοχαρακτηριστικού `sideup`. Θυμηθείτε ότι από μια κλάση κατασκευάζουμε αντικείμενα τα οποία είναι όλα *διαφορετικά* μεταξύ τους. Όταν δημιουργείται ένα αντικείμενο, η Python συνδέει την παράμετρο `self` με το αντικείμενο αυτό και μας επιτρέπει να αναφερόμαστε στις συγκεκριμένες μεθόδους του. Συνεπώς η `self` είναι **απαραίτητη** στις μεθόδους και στα πεδία μιας κλάσης. Δεν είναι απαραίτητο να την ονομάσετε `self`, αλλά συνιστάται κάτι τέτοιο για να συμφωνεί με τη συνήθη πρακτική.

2.2 Κατασκευή αντικειμένου

Έχοντας στη διάθεσή μας την κλάση `Coin` μπορούμε να κατασκευάσουμε αντικείμενά της και να τα χρησιμοποιήσουμε σε οποιοδήποτε πρόγραμμα χρειάζεται να παίξουμε κορώνα-γράμματα. Έχουμε δηλαδή δημιουργήσει έναν νέο τύπο δεδομένου με τον οποίο κατασκευάζουμε και διαχειριζόμαστε κέρματα για τέτοιου είδους παιχνίδια.

Αν η κλάση `Coin` είναι ορισμένη σε ένα αρχείο (module) με το όνομα `games.py`, τότε μπορούμε να κατασκευάσουμε κέρματα σε οποιοδήποτε πρόγραμμα. Στο παρακάτω πρόγραμμα κατασκευάζουμε 2 κέρματα, τα στρίβουμε, και ελέγχουμε αν έφεραν και τα δύο το ίδιο αποτέλεσμα.

```
from games import Coin

def main():
    # Δημιουργία 2 αντικειμένων τύπου Coin
    c1 = Coin()
    c2 = Coin()
    # Ρίψη των κερμάτων
    c1.toss()
    c2.toss()
    # Έλεγχος αποτελεσμάτων
    if c1.get_sideup() == c2.get_sideup():
        print('Και τα 2 κέρματα είναι ' + c1.get_sideup())
    else:
        print('Το κέρμα 1 έφερε ' + c1.get_sideup())
        print('Το κέρμα 2 έφερε ' + c2.get_sideup())

        main()
```

```
Το κέρμα 1 έφερε Γράμματα
Το κέρμα 2 έφερε Κορώνα
```

Ένα αντικείμενο της κλάσης, π.χ. το `c1` κατασκευάζεται ως εξής

```
c1 = Coin()
```

Αυτό κάνει να συμβούν δύο πράγματα:

1. Δημιουργείται στη μνήμη ένα αντικείμενο της κλάσης `Coin`.
2. Εκτελείται η μέθοδος αρχικοποίησης `__init__` της κλάσης, και η παράμετρος `self` αναφέρεται αυτόματα στο αντικείμενο που δημιουργήθηκε, εδώ στο `c1`.

Στη συγκεκριμένη εκτέλεση είναι ξεκάθαρο ότι τα 2 αντικείμενα είναι διαφορετικά μεταξύ του (το ιδιοχαρακτηρίστικό τους `sideup` είναι διαφορετικό).

Παρατηρούμε επίσης τον τρόπο κλήσης των μεθόδων: τη θέση του `self` στον ορισμό των μεθόδων στην κλάση, παίρνει κατά την εκτέλεση το αντικείμενο στο οποίο θέλουμε να αναφερθούμε (το `c1` ή το `c2`) και η μέθοδος καλείται ως `όνομα_αντικειμένου.όνομα_μεθόδου()`.

2.3 Μέθοδοι πρόσβασης και τροποποιητικές μέθοδοι

Μια υποτυπώδης κλάση για την κατασκευή υπαλλήλων θα μπορούσε να είναι η εξής

```
class Employee:

    def __init__(self, name, salary):
        self.name = name.upper()
        self.salary = salary

    def get_raise(self, percent):
        self.salary *= (1+percent)
```

Και ένα παράδειγμα χρήση θα μπορούσε να είναι:

```
e = Employee('Μπάμπης', 1000)
e.salary = 2000
print('Ο μισθός του', e.name, 'είναι', e.salary)
e.name = 'Babis'
e.get_raise(0.2)
print('Ο ', e.name, 'άλλαξε το όνομά του και ο μισθός του έγινε', e.salary)
```

Παρατηρούμε ότι μπορούμε να διαχειριστούμε άμεσα τα ιδιοχαρακτηριστικά του αντικειμένου: να δούμε τις τιμές τους και να τις αλλάξουμε αν θέλουμε. Αυτός είναι **κακός** αντικειμενοστραφής σχεδιασμός και πρέπει να αποφεύγεται. Μάλιστα σε κάποιες άλλες γλώσσες ΑΣΠ, όπως η Java, υπάρχει η δυνατότητα να εξασφαλίσουμε ότι κάτι τέτοιο δεν μπορεί να γίνει.

Ο **καλός** αντικειμενοστραφής σχεδιασμός:

- Διαχειρίζεται τα πεδία μιας κλάσης ως *ιδιωτικά (private)* και άμεση πρόσβαση σε αυτά "επιτρέπεται" να γίνεται μόνο μέσα στον ορισμό της κλάσης.
- Για τις ανάγκες των εφαρμογών που χρησιμοποιούν αντικείμενα της κλάσης, η κλάση παρέχει μεθόδους *πρόσβασης (getters)* που επιστρέφουν τις τιμές των ιδιοχαρακτηριστικών του αντικειμένου και *τροποποιητικές (setters)* μεθόδους με τις οποίες μπορούμε να αλλάξουμε τις τιμές των ιδιοχαρακτηριστικών (εφόσον αυτό είναι απαραίτητο να γίνει).

Με χρήση τροποποιητικών και μεθόδων πρόσβασης η κλάση `Employee` γράφεται

```python
class Employee:

    def __init__(self, name, salary):
        self.name = name.upper()
        self.salary = salary

    # Μέθοδοι πρόσβασης

    def get_name(self): return self.name

    def get_salary(self): return self.salary

    # Τροποποιητικές μέθοδοι

    def set_name(self, new_name):
        self.name = new_name.upper()

    def set_salary(self, money):
        self.salary = money

    def give_raise(self, percent):
        self.salary *= (1+percent)
```

Εδώ βλέπουμε ότι υπάρχει η απαίτηση το όνομα του υπαλλήλου να γράφεται με κεφαλαία γράμματα (χρήση της `upper`). Αν οι αλλαγές ονόματος σε τυχόν εφαρμογές γίνουν αποκλειστικά μέσω της τροποποιητικής μεθόδου `set_name`, η απαίτηση αυτή διασφαλίζεται πράγμα που δεν μπορούσε να γίνει με άμεση αλλαγή του ιδιοχαρακτηριστικού `name`. Αν μελλοντικά η απαίτηση αυτή αλλάξει, τότε όλες οι εφαρμογές που χρησιμοποιούν την κλάση `Employee` θα εξακολουθήσουν να δουλεύουν σωστά.

```python
e = Employee('Bob', 1200)
print(e.get_name(), e.get_salary())
e.set_name('babis')
e.give_raise(0.1)
```

Η έξοδος του παραπάνω κώδικα είναι:

```
BOB 1200
Ονομα: BABIS, Μισθός:1320.0
```

2.4 Παράδειγμα: κλάση κατασκευής σημείων στο επίπεδο

Τα ιδιοχαρακτηριστικά ενός σημείου στο επίπεδο είναι οι συντεταγμένες του. Οι μέθοδοι που μας χρειάζονται είναι:

- Οι μέθοδοι πρόσβασης `getX`, `getY`
- Η μέθοδος `dist` που υπολογίζει την απόσταση του σημείου μας (`self`) από κάποιο άλλο (`other`).
- Η μέθοδος `equals` που ελέγχει αν το σημείο μας (`self`) ταυτίζεται με κάποιο άλλο (`other`).

- Η μέθοδος `move` που μεταθέτει το σημείο κατά $\Delta x, \Delta y$.

```
import math

class Point:
    "Κλάση κατασκευής σημείων στο επίπεδο"

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def getX(self): return self.x

    def getY(self): return self.y

    def dist(self, other):
        return math.sqrt((self.x - other.x)**2 + (self.y - other.y)**2)

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def equals(self, other):
        return self.x == other.x and self.y == other.y

    def __str__(self):
        return '('+ format(self.x,'.3f') + ',' + format(self.y,'.3f') + ')'
```

Η ειδική μέθοδος `__str__`

Η `Point` περιλαμβάνει επίσης την ειδική μέθοδο `__str__` η οποία, εφόσον είναι ορισμένη μέσα σε μια κλάση, μετατρέπει τα αντικείμενα της κλάσης σε συμβολοσειρές. Η `print` της Python, μετατρέπει τα ορίσματά της σε συμβολοσειρές προκειμένου να τα εμφανίσει. Για να κάνει την μετατροπή ελέγχει αν η κλάση παρέχει τη μέθοδο μετατροπής `__str__`, και αν ναι τη χρησιμοποιεί. Αν δεν είχαμε ορίσει τη μέθοδο `__str__` στην `__Point__`, τότε ο κώδικας

```
pt = Point(1/3, math.pi)
print(pt)
```

θα εμφάνιζε

```
<__main__.Point object at 0x7f87ea33a588>
```

ένω τώρα εμφανίζει

```
(0.333,3.142)
```

το σημείο μορφοποιημένο σύμφωνα με την `__str__` που έχουμε ορίσει.

2.5 Εφαρμογή - Αποθήκευση αντικειμένων σε λίστα

Έστω 100 τυχαία σημεία στο επίπεδο τα οποία είναι αποθηκευμένα σε λίστα. Ζητάμε να βρούμε τη μέγιστη απόσταση (διάμετρο) ανάμεσα σε όλα τα δυνατά ζεύγη τους καθώς και ποια είναι αυτά τα πιο απομακρυσμένα σημεία.

Αν η κλάση `Point` βρίσκεται στο module `geom.py`, τότε ο κώδικας της εφαρμογής είναι ο εξής

```
import random
from geom import Point

def main():
    # Εύρεση πιο απομακρυσμένων σημείων σε ένα σύνολο
    # 100 τυχαίων σημείων
    n = 100
    points = []
    for i in range(100):
        x = random.random()
        y = random.random()
        points.append(Point(x,y))
    d, Q1, Q2 = diameter(points)
    print("Μέγιστη απόσταση: " + format(d,'.3f'))
    print("ανάμεσα στα σημεία "+ str(Q1) + " και "+ str(Q2))

def diameter(pt):
    n = len(pt)
    dmax = 0
    for i in range(n-1):
        for j in range(i+1,n):
            dij = pt[i].dist(pt[j])
            if dij > dmax:
                dmax = dij
                Q1 = pt[i]
                Q2 = pt[j]
    return dmax, Q1, Q2

main()
```

2.6 Εφαρμογή - Κοινά ιδιοχαρακτηριστικά

Ένα ηλεκτρικό φορτίο q που βρίσκεται στο επίπεδο, δημιουργεί γύρω του ηλεκτροστατικό πεδίο. Το δυναμικό εξαιτίας του q, σε κάθε σημείο του επιπέδου που βρίσκεται σε απόσταση r από το q δίνεται από

$$V = k\frac{q}{r}$$

Τα ιδιοχαρακτηριστικά μιας κλάσης `Charge` για φορτία είναι η τιμή του φορτίου `q` και η θέση του στο επίπεδο `pos`. Η θέση είναι αντικείμενο τύπου `Point`, παρατηρούμε δηλαδή ότι κάποια από τα

ιδιοχαρακτηριστικά μιας κλάσης μπορεί να είναι αντικείμενα κάποιας άλλης κλάσης.

Κάθε αντικείμενο τύπου `Charge` που θα δημιουργηθεί σε μια εφαρμογή θα έχει τις δικές του τιμές για τα ιδιοχαρακτηριστικά του. Η τιμή της ηλεκτροστατικής σταθεράς k είναι κοινή για όλα τα φορτία, είναι με άλλα λόγια ένα χαρακτηριστικό της κλάσης και δεν έχει νόημα να αποθηκευτεί σε κάθε αντικείμενο της κλάσης. Ο τρόπος αντιμετώπισης τέτοιων κοινών ιδιοχαρακτηριστικών σε μια κλάση φαίνεται στο παρακάτω παράδειγμα

```
from geom import Point

class Charge:

    k = 8.99e9

    def __init__(self, q, pt):
        self.pos = pt
        self.q = q

    def potential_at(self, pt):
        return Charge.k*self.q/self.pos.dist(pt)

    def __str__(self): return str(self.q)+' at '+ str(self.pos)


def main():
    t = Point(0.51, 0.63)
    p = Point(1, 1)
    c = Charge(21.3, t)
    print(c)
    print(c.potential_at(p))

main()
```

Παρατηρήστε ότι η τιμή του `k` μέσα από την μέθοδο `potential_at` δίνεται ως `Charge.k`.

2.7 Ένα πρόβλημα καταδίωξης

Στο επόμενο παράδειγμα θα δούμε πώς μπορούμε να συνδυάσουμε αντικείμενα διαφόρων κλάσεων για την επίλυση ενός προβλήματος καταδίωξης. Ένας ταύρος κυνηγάει έναν πεζοπόρο. Ο πεζοπόρος τρέχει προς ένα ασφαλές σημείο, έναν φράχτη. Η κίνηση του πεζοπόρου είναι σε ευθεία γραμμή προς τον φράχτη, ενώ ο ταύρος σε κάθε χρονική στιγμή κατευθύνεται προς την τρέχουσα θέση του πεζοπόρου.

Στα προβλήματα αυτά, παρακολουθούμε την εξέλιξη του φαινομένου σε *διακριτά* χρονικά σημεία $t_0, t_0+\Delta t, t_0+2\Delta t, \ldots$. Εδώ θα ορίσουμε μια καθολική μεταβλητή `DT` για τον σκοπό αυτό. Εκτός της `Point` που θα χρησιμοποιήσουμε για τον καθορισμό των θέσεων, θα χρειαστούμε 3 ακόμα κλάσεις: `Bull`, `Hiker`, `Fence`, που θα περιέχονται στο παρακάτω module `pursuit.py`.

```
from geom import Point

DT = float(input('DT? '))


class Hiker:

    def __init__(self, p, v=8):
        self.position = p
        self.speed = v

    def position(self): return self.position

    def run(self): return self.position.move(self.speed*DT, 0)

    def dist(self, other): return self.position.dist(other.position())

    def saved(self, fence): return self.dist(fence) <= 1


class Bull:

    def __init__(self, p, v=8.5):
        self.position = p
        self.speed = v

    def position(self): return self.position

    def dist(self, other): return self.position.dist(other.position())

    def chase(self, hiker):
        s = self.dist(hiker)
        dx = (hiker.position().getX() - self.position.getX())*self.speed*DT/s
        dy = -self.position.getY()*self.speed*DT/s
        self.position.move(dx, dy)

    def gotcha(self,other): return self.dist(other) <= 1


class Fence:
    def __init__(self, p):
        self.position = p

    def position(self): return self.position
```

Έχοντας στη διάθεση μας τις παραπάνω κλάσεις μπορούμε να γράψουμε προγράμματα που να εξετάζουν διάφορα σενάρια αυτού του είδους καταδίωξης. Π.χ. ένας ταύρος κυνηγάει ένα πεζοπόρο

```
def main1bull():

    b = Bull(Point(50,100))
    h = Hiker(Point(0,0))
    f = Fence(Point(400,0))

    while not h.saved(f):
        b.chase(h)
        h.run()
        if b.gotcha(h):
            print('Hiker caught')
            print('Distance from fence: '+str(h.dist(f)))
            break
    else:
        print('Hiker saved')
        print('Distance from bull: '+str(h.dist(b)))
```

Σε ένα άλλο σενάριο για παράδειγμα, θα μπορούσαν να υπάρχουν 2 ταύροι και 1 πεζοπόρος.

```
def main2bulls():

    bulls = [Bull(Point(50,100)), Bull(Point(-10,100),20)]
    h = Hiker(Point(0,0))
    f = Fence(Point(400,0))

    while not h.saved(f):
        for b in bulls:
            if b.gotcha(h):
                print('Hiker caught')
                print('Distance from fence: '+str(h.dist(f)))
                return
            else:
                b.chase(h)
        h.run()
    print('Hiker saved')
    for b in bulls:
        print('Distance from bull: '+str(h.dist(b)))
```

Η ευελιξία στην δημιουργία διαφόρων παραλλαγών σε ένα πρόγραμμα που χρησιμοποιεί διάφορα αντικείμενα που αλληλεπιδρούν, είναι ένα από τα σημαντικότερα πλεονεκτήματα του ΑΣΠ.

2.8 Μεταβίβαση αντικειμένων ως ορίσματα συναρτήσεων

Στο παρακάτω πρόγραμμα χρησιμοποιούμε τη συνάρτηση `incr_int` για να αυξήσουμε την τιμή ενός ακεραίου κατά 1.

```
def incr_int(n):
    n += 1
```

```
    print("n =", n)

def main():
    x = 17
    incr_int(x)
    print('x =', x)

main()
```

Η εκτέλεση δίνει, όπως αναμενόταν

```
n = 18
x = 17
```

αφού τα ορίσματα των συναρτήσεων της Python περνάνε με τιμή (pass-by-value) και η αλλαγή της τιμής της *τοπικής* μεταβλητής n απλώς κάνει το n να αναφέρεται σε ένα διαφορετικό αντικείμενο (το 18 στην προκειμένη περίπτωση). Το x εξακολουθεί να αναφέρεται στο 17.

Αν τώρα ορίσουμε την παρακάτω κλάση

```
class embedded_int:

    def __init__(self, n):
        self.value = n

    def set_value(self, n):
        self.value = n

    def get_value(self):
        return self.value

    def __str__(self):
        return str(self.value)
```

που στην ουσία είναι μια κλάση περιτύλιγμα για τους ακεραίους. Ένα αντικείμενο τύπου `embedded_int` είναι στην ουσία ένας ακέραιος με ιδιοχαρακτηριστικό την τιμή του και μεθόδους διαχείρισής της.

Το αντίστοιχο πρόγραμμα που χρησιμοποιεί `embedded_int` αντί για `int`

```
def incr_embedded_int(N):
    N.set_value(N.get_value()+1)
    print("N =", N)

def change_embedded_int(N):
    t = embedded_int(N.get_value() + 1)
    N = t
    print("N =", N)

def main():
    y = embedded_int(17)
    incr_embedded_int(y)
```

```
    print("y =", y)
    change_embedded_int(y)
    print("y =", y)

main()
```

δίνει

```
N = 18
y = 18
N = 19
y = 18
```

Η κλήση στη συνάρτηση `incr_embedded_int`, κάνει την τοπική μεταβλητή `N` να αναφέρεται στο αντικείμενο στο οποίο αναφέρεται η `y` και μπορεί να τροποποιήσει άμεσα στην τιμή του μέσω της `set_value`.

Η συνάρτηση `change_embedded_int` είναι η αντίστοιχη της `incr_int` του προηγούμενου παραδείγματος, αφού η τοπική μεταβλητή `N` αναφέρεται σε ένα διαφορετικό αντικείμενο από την `y`. Στη `main`, η μεταβλητή `y` εξακολουθεί να αναφέρεται στον `embedded_int` με τιμή 18.

2.9 Υπερφόρτωση τελεστών

Κλάση για ρητούς αριθμούς

Αν θέλουμε να δουλεύουμε αποκλειστικά με ρητούς αριθμούς, μπορούμε να σχεδιάσουμε μια κλάση `Rational` με την οποία θα κατασκευάζουμε ρητούς αριθμούς και θα μπορούμε να κάνουμε πράξεις, συγκρίσεις κλπ με αυτούς. Τα ιδιοχαρακτηριστικά ενός ρητού αριθμού είναι ο αριθμητής και ο παρανομαστής του. Μια τέτοια κλάση με μερικές μόνο ενδεικτικές μεθόδους πράξεων είναι η παρακάτω.

```
import math

class Rational:
    "Κλάση για πράξεις με ρητούς αριθμούς"

    def __init__(self, x = 0, y = 1):
        self.num = x
        if y == 0:
            # 0 στον παρανομαστή γίνεται 1
            self.den = 1
        else:
            # Το πρόσημο στον αριθμητή
            self.den = abs(y)
            if y < 0: self.num = -self.num
            self.simplify()               # απλοποίηση

    def get_num(self): return self.num
```

```
    def get_den(self): return self.den

    def __str__(self):
        "Μετατροπή σε συμβολοσειρά"
        if self.den == 1:
            return str(self.num)
        else:
            return str(self.num) + '/' + str(self.den)

    def add(self, other):
        "Άθροισμα με κλήση μεθόδου"
        return Rational(self.num*other.den + self.den*other.num,
                        self.den*other.den)
    def times(self, other):
        "Γινόμενο ρητών"
        return Rational(self.num*other.num, self.den*other.den)

    def equals(self, other):
        return self.num==other.num and self.den==other.den

    def simplify(self):
        "Απλοποίηση με κλήση της math.gcd"
        g = math.gcd(abs(self.num), abs(self.den))
        self.num //= g
        self.den //= g
```

Παράδειγμα χρήσης:

```
from rational import Rational

x = Rational(2, 4)
y = Rational(5)
z = Rational(1, 3)

a = x.add(y)
print(x, '+', y, '=', a)
b = x.add(y).add(z)
print(x, '+', y, '+', z, '=', b)
c = x.times(y)
print(x, '*', y, '=', c)
d = x.add(y).times(z)
print(x, '+', y, '*', z, '=', d)
```

Αποτελέσματα:

```
1/2 + 5 = 11/2
1/2 + 5 + 1/3 = 35/6
1/2 * 5 = 5/2
1/2 + 5 * 1/3 = 11/6
```

Επειδή οι πράξεις με αντικείμενα ρητούς επιστρέφουν ένα νέο ρητό, μπορούμε να συνδυάσουμε πολλές πράξεις στον ίδιο υπολογισμό όπως στην

```
d = x.add(y).times(z)
```

Θα ήταν βολικό αν αντί της προηγούμενης έκφρασης να μπορούσαμε να γράψουμε

```
d = x + y * z
```

Οπότε οι υπολογισμοί με ρητούς στο πρόγραμμά μας να θυμίζουν απευθείας τις αντίστοιχες μαθηματικές πράξεις. Αυτό *υποστηρίζεται* άμεσα στην Python και ονομάζεται **υπερφόρτωση τελεστών**. Έχουμε δει παραδείγματα υπερφόρτωσης τελεστών, όταν για παράδειγμα χρησιμοποιήσαμε τον τελεστή "+" στη συνένωση συμβολοσειρών.

Για να μπορέσουμε να χρησιμοποιήσουμε τον τελεστή "+" στην `Rational`, αρκεί να μετονομάσουμε τη μέθοδο `add` σε `__add__`. Ομοίως για τον τελεστή "*", θα πρέπει να μετονομάσουμε την `times` σε `__mul__`. Τα ονόματα αυτά σηματοδοτούν στην Python, ότι η εκτέλεση των συναρτήσεων αυτών μπορεί να γίνει με χρήση τελεστών αντί της συνηθισμένης κλήσης.

Κλάση για ρητούς αριθμούς με υπερφόρτωση τελεστών

Η παραπάνω κλάση με υπερφόρτωση των τελεστών "+", "*" και "==" (ο τελεστής ισότητας) γράφεται

```
import math

class Rational:
    "Κλάση για πράξεις με ρητούς αριθμούς"

    def __init__(self, x = 0, y = 1):
        self.num = x
        if y == 0:
            # 0 στον παρανομαστή γίνεται 1
            self.den = 1
        else:
            # Το πρόσημο στον αριθμητή
            self.den = abs(y)
            if y < 0: self.num = -self.num
            self.simplify()

    def get_num(self): return self.num

    def get_den(self): return self.den

    def __str__(self):
        "Μετατροπή σε συμβολοσειρά"
        if self.den == 1:
            return str(self.num)
        else:
            return str(self.num) + '/' + str(self.den)
```

```
    def __add__(self, other):
        "Άθροισμα ρητών με υπερφόρτωση του τελεστή +"
        return Rational(self.num*other.den + self.den*other.num,
                        self.den*other.den)

    def __mul__(self, other):
        "Γινόμενο ρητών με υπερφόρτωση του τελεστή *"
        return Rational(self.num*other.num, self.den*other.den)

    def __eq__(self, other):
        return self.num==other.num and self.den==other.den

    def simplify(self):
        # Απλοποίηση με κλήση της math.gcd
        g = math.gcd(abs(self.num), abs(self.den))
        self.num //= g
        self.den //= g
```

και το προηγούμενο παράδειγμα χρήσης:

```
from rational import Rational

x = Rational(2, 4)
y = Rational(5)
z = Rational(1, 3)

a = x + y
b = x + y + z)
c = x*y
d = x + y*z
```

Αντιστοιχία τελεστών - ειδικών μεθόδων

Στον παρακάτω πίνακα φαίνονται μερικά από τα ονόματα των ειδικών μεθόδων της Python, που εφόσον οριστούν σε κάποια κλάση υπερφορτώνουν τους αντίστοιχους τελεστές.

Τελεστής	Μέθοδος	Έκφραση	Εσωτερική υλοποίηση
+	__add__	x + y	x.__add__(y)
-	__sub__	x - y	x.__sub__(y)
*	__mul__	x * y	x.__mul__(y)
**	__pow__	x **y	x.__pow__(y)
/	__truediv__	x / y	x.__truediv__(y)
//	__floordiv__	x // y	x.__floordiv__(y)
%	__mod__	x % y	x.__mod__(y)
==	__eq__	x == y	x.__eq__(y)

Αν x, y είναι 2 αντικείμενα μιας κλάσης με υπερφορτωμένους τελεστές τότε οι πράξεις της τρίτης

στήλης υλοποιούνται εσωτερικά από την Python σύμφωνα με την τέταρτη στήλη.

Να σημειωθεί ότι οι τελεστές αυτοί δεν είναι απαραίτητο να αντιστοιχούν σε μαθηματικές πράξεις μετά την υπερφόρτωση.

Κλάση για πολυώνυμα

Παρακάτω δίνεται άλλη μια ενδεικτική κλάση για πολυώνυμα.

```
class Polynomial:

    def __init__(self, coeff):
        self.coeff = coeff[::-1]

    def __call__(self, x):
        value = 0
        xpower = 1
        for term in self.coeff:
            value += term*xpower
            xpower *= x
        return value

    def __str__(self):
        s = ''
        xpower = len(self.coeff)
        for term in self.coeff[::-1]:
            xpower -= 1
            if term != 0:
                s += ' + {}*x^{}'.format(term, xpower)
            s = s.replace('+ -', '- ')
            s = s.replace('x^0', '1')
            s = s.replace(' 1*', ' ')
            s = s.replace('*1', '')
            s = s.replace('x^1 ', 'x ')
            s = s.replace('x^1', 'x')
            if s[0:3] == ' + ': s = s[3:]
            if s[0:3] == ' - ': s = '-' + s[3:]
        return s

    def __add__(self, other):
        nself, nother = len(self.coeff), len(other.coeff)
        if nself <= nother:
            res_coeff = other.coeff[:]
            for i in range(nself):
                res_coeff[i] += self.coeff[i]
        else:
            for i in range(nother):
```

```
            res_coeff += other.coeff[i]
        return Polynomial(res_coeff[::-1])
```

Το μοναδικό ιδιοχαρακτηριστικό της κλάσης είναι μια λίστα που περιέχει όλους τους συντελεστές ενός πολυωνύμου (συμπεριλαμβανομένων και των μηδενικών) οι οποίοι, για λόγους αποτελεσματικότερης διαχείρισης στην κλάση, είναι αποθηκευμένοι από τον σταθερό όρο έως τον συντελεστή του μεγιστοβάθμιου όρου.

Για την κατασκευή όμως του πολυωνύμου, οι συντελεστές θα δίνονται με αντίστροφη σειρά. Π.χ. το πολυώνυμο $5x^2 - 8$ θα κατασκευάζεται

```
p = Polynomial([5, 0, 2])
```

ενώ εσωτερικά η κλάση θα δουλεύει με τους συντελεστές αποθηκευμένους στη λίστα `[2, 0, 5]`.

Μια άλλη ειδική μέθοδος της Python είναι η `__call__`, με την οποία ένα αντικείμενο μιας κλάσης συμπεριφέρεται ως συνάρτηση:

```
p = Polynomial([2, -1, 3])
print('Η τιμή στο 3 είναι', p(3))
```

Δηλαδή η `p(3)` στην ουσία καλεί την ειδική μέθοδο `p.__call__(3)`.

Προσέξτε επίσης τη κάπως πολύπλοκη `__str__` που φιλοδοξεί να μετατρέψει το πολυώνυμο σε μια συμβολοσειρά που να μοιάζει με τον μαθηματικό τρόπο γραφής του πολυωνύμου.

3 Κληρονομικότητα

Η κληρονομικότητα είναι μια από τις βασικές έννοιες του ΑΣΠ, που και πάλι προέρχεται από έννοιες του πραγματικού κόσμου. Στην καθημερινότητα συναντάμε διάφορα αντικείμενα που εξειδικεύουν μια γενικότερη (και συχνά αφηρημένη έννοια). Ένα αυτοκίνητο είναι εξειδίκευση της έννοιας του οχήματος, όπως και ένα λεωφορείο ή ένα φορτηγό. Ένα σπορ αυτοκίνητο είναι μια περαιτέρω εξειδίκευση του αυτοκινήτου. Αντίστοιχες σχέσεις κληρονομικότητας μπορούμε να συναντήσουμε παντού (ζώο <- θηλαστικό <- σκύλος <- Λαμπραντόρ).

Με την έννοια της κληρονομικότητας στον ΑΣΠ, δημιουργούμε μια ιεραρχία κλάσεων, στην οποία κάθε κλάση ”κληρονομεί” ιδιοχαρακτηριστικά και μεθόδους από τις κλάσεις που βρίσκονται πιο πάνω από αυτήν στην ιεραρχία. Μια κλάση **A** που βρίσκεται ψηλότερα στην ιεραρχία από μια κλάση **B** λέγεται *υπερκλάση* της **B**. Ενώ η **B** είναι μια *υποκλάση* της **A**. Επιπλέον των χαρακτηριστικών και μεθόδων που κληρονομεί από τις υπερκλάσεις της, μια υποκλάση μπορεί

- Να προσθέσει νέα ιδιοχαρακτηριστικά και μεθόδους που εξειδικεύουν την υποκλάση.
- Να *υπερκαλύψει* μεθόδους της υπερκλάσης. Να δημιουργήσει δηλαδή νέες μεθόδους με το ίδιο όνομα με αυτές της υπερκλάσης οι οποίες χρησιμοποιούνται για κάποιες εξειδικευμένες ανάγκες της υποκλάσης.

  Έστω η γενική κλάση `Person` για φυσικά πρόσωπα, με μοναδικό ιδιοχαρακτηριστικό το όνομα ενός προσώπου.

  ```
  class Person:
  ```

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

    def get_name(self): return self.name

    def __str__(self): return self.name
```

Μια κλάση `Student` για φοιτητές, μπορεί να γραφεί ως υποκλάση της `Student` αφού κάθε φοιτητής είναι ένα φυσικό πρόσωπο και έχει ένα όνομα. Ένας φοιτητής έχει επιπλέον ως ιδιοχαρακτηριστικά έναν ΑΜ και τα μαθήματα που έχει παρακολουθήσει και τους βαθμούς που έχει πάρει σε αυτά. Τα μαθήματα και τους βαθμούς επιλέγουμε να τα αποθηκεύσουμε σε ένα λεξικό.

```
1  class Student(Person):
2
3      def __init__(self, name, AM):
4          Person.__init__(self, name)
5          self.AM = str(AM)
6          self.courses = {}
7
8      def __str__(self): return Person.__str__(self)+' ('+self.AM+')'
9
10     def get_AM(self): return self.AM
11
12     def add_course_grade(self, course, grade):
13         self.courses[course] = int(grade)
14
15     def show_grades(self):
16         for course in self.courses:
17             print(course, self.courses[course])
18
19     def get_MO(self):
20         return sum(self.courses.values())/len(self.courses)
```

- Η γραμμή 1 λέει ότι η `Student` είναι υποκλάση της `Person`
- Η `Student` κληρονομεί από την `Person` το ιδιοχαρακτηριστικό `name`, και τη μέθοδο πρόσβασης `get_name`.
- Η μέθοδος `__init__` υπερκαλύπτει την αντίστοιχη μέθοδο της `Person`.
  - Προκειμένου να αρχικοποιήσουμε έναν `Student` πρέπει πρώτα να αρχικοποιήσουμε την υπερκλάση. Αυτό γίνεται στη γραμμή 4, στην οποία καλείται η `__init__` της υπερκλάσης.
  - Στη γραμμή 6, αρχικοποιούμε ένα κενό λεξικό για τις βαθμολογίες.
- Η μέθοδος `__str__` υπερκαλύπτει επίσης την αντίστοιχη της `Person`. Η μετατροπή ενός αντικειμένου `Student` σε συμβολοσειρά γίνεται καλώντας την `__str__` της υπερκλάσης και προσθέτοντας τον ΑΜ στη συμβολοσειρά.

– Η `__Student_` έχει επιπλέον τις μεθόδους `get_AM`, `add_course_grade`, `show_grades` και `get_MO`, οι οποίες δεν μπορούν να χρησιμοποιηθούν από αντικείμενα τύπου `Person`, αφού ένα τέτοιο αντικείμενο δεν είναι φοιτητής.

  Παράδειγμα χρήσης:

```
s1 = Student('Παπαδόπουλος', 1112201700344)
s2 = Student('Παπαδάκη', 1112201500058)
s1.add_course_grade('Πληροφορική ΙΙ', 8)
s1.add_course_grade('Απειροστικός Λογισμός ΙΙ', 5)
s1.add_course_grade('Γραμμική Άλγεβρα ΙΙ', 7)
print(s1.get_name())                    # από την Person
print(s2.get_MO())                      # από την Student
```

Σύνθεση-έχει, Κληρονομικότητα-είναι

Ως τώρα οι κλάσεις που συναντήσαμε αποτελούν παραδείγματα *σύνθεσης*: ένα φορτίο **έχει μια** θέση στο επίπεδο και μία τιμή. Ένας υπάλληλος **έχει ένα** όνομα και ένα μισθό. Ένας ρητός **έχει** αριθμητή και παρανομαστή.

Στις σχέσεις κληρονομικότητας όμως, ένα αυτοκίνητο **είναι ένα** όχημα, ένα σπορ αυτοκίνητο **είναι ένα** αυτοκίνητο αλλά **είναι** και ένα όχημα. Ο φοιτητής **είναι ένα** πρόσωπο. Ο μεταπτυχιακός φοιτητής **είναι ένας** φοιτητής αλλά **είναι** και ένα πρόσωπο.

Επομένως αν ένα αντικείμενο **A έχει** ένα αντικείμενο **B**, τότε το **B** είναι ένα ιδιοχαρακτηριστικό της κλάσης του **A**: *σύνθεση*.

Αν ένα αντικείμενο **Y είναι** και ένα αντικείμενο **X**, τότε η κλάση του **Y** είναι μια υποκλάση της **X**: *κληρονομικότητα*.

Το πρόβλημα καταδίωξης με κληρονομικότητα

Κοινό χαρακτηριστικό του πεζοπόρου, του ταύρου και του φράχτη είναι ότι **είναι** αντικείμενα στον χώρο. Επομένως μπορούμε να θεωρήσουμε ότι είναι υποκλάσεις μιας κλάσης `ObjectInSpace` με ιδιοχαρακτηριστικό τη θέση τους (ένα σημείο) και τη μέθοδο πρόσβασης `position` και τη μέθοδο `dist` για την απόσταση ανάμεσα σε 2 αντικείμενα της κλάσης.

Επιπλέον ο ταύρος και ο πεζοπόρος μπορούν να κινηθούν με κάποια ταχύτητα, συνεπώς μπορούμε να θεωρήσουμε ότι είναι υποκλάσεις μιας κλάσης `MovingObject`. Η `MovingObject` είναι μια υποκλάση της `ObjectInSpace` με επιπλέον ιδιοχαρακτηριστικό την ταχύτητα κίνησης.

Με αυτές τις παρατηρήσεις μπορούμε να γράψουμε τις κλάσεις για το πρόβλημα καταδίωξης ως εξής:

```
from geom import Point

DT = float(input('DT? '))

class ObjectInSpace:
    def __init__(self, p):
```

```
        self.position = p

    def position(self): return self.position

    def dist(self, other): return self.position.dist(other.position())


class MovingObject(ObjectInSpace):

    def __init__(self, p, v):
        ObjectInSpace.__init__(self, p)
        self.speed = v


class Hiker(MovingObject):

    def run(self): return self.position.move(self.speed*DT, 0)

    def saved(self, fence): return self.dist(fence) <= 1


class Bull(MovingObject):

    def chase(self, hiker):
        s = self.dist(hiker)
        dx = (hiker.position().getX() - self.position.getX())*self.speed*DT/s
        dy = -self.position.getY()*self.speed*DT/s
        self.position.move(dx, dy)

    def gotcha(self,other): return self.dist(other) <= 1


class Fence(ObjectInSpace):
```

Η κλάση `Fence` τώρα είναι απλώς ένα άλλο όνομα για την `ObjectInSpace`, που όμως είναι ξεκάθαρο σε τι αντικείμενο αναφέρεται (σε φράχτη).

Οι κλάσεις `Bull` και `Hiker` δεν έχουν δική τους `__init__` γιατί δεν προσθέτουν κάποιο επιπλέον ιδιοχαρακτηριστικό σε αυτά που κληρονομούν από τη `MovingObject`.

4 Πολυμορφισμός

Ο *πολυμορφισμός* είναι μαζί με την *αφαίρεση*, την *ενθυλάκωση* και την *κληρονομικότητα* μια από τις 4 βασικές έννοιες του ΑΣΠ. Δίνει τη δυνατότητα κλήσης μιας υπερκαλυμμένης μεθόδου ανάλογα με τον τύπο του αντικειμένου με το οποίο καλείται. Αν χρησιμοποιηθεί το αντικείμενο της υποκλάσης για την κλήση μιας υπερκαλυμμένης μεθόδου, τότε εκτελείται η μέθοδος της υποκλάσης. Αν χρησιμοποιηθεί

το αντικείμενο της υπερκλάσης, τότε εκτελείται η αντίστοιχη μέθοδος της υπερκλάσης.

Έστω η κλάση `Player` η οποία έχει 2 υποκλάσης: την `FootballPlayer` και την `BasketballPlayer`.

```
class Player:

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

    def get_name(self): return self.name

    def job(self):
        print('I play games')


class FootballPlayer(Player):

    def __init__(self, name, goals):
        Player.__init__(self, name)
        self.total_goals = goals

    def get_goals(self): return self.total_goals

    def job(self):
        print('I play football')


class BasketballPlayer(Player):

    def __init__(self, name, points):
        Player.__init__(self, name)
        self.total_points = points

    def job(self):
        print('I play basketball')

    def get_points(self): return self.total_points
```

Έστω μια απλή εφαρμογή που κατασκευάζει ένα αντικείμενο από κάθε κλάση, και καλεί μια γενική συνάρτηση `player_sport` που εμφανίζει το όνομα και το αγώνισμα κάθε παίχτη.

```
def main():
    p = Player('John Doe')
    f = FootballPlayer('Messi', 614)
    b = BasketballPlayer('Galis', 24805)
    player_sport(p)
    player_sport(f)
    player_sport(b)
```

```
def player_sport(x):
    print('I am', x.get_name(), end=' : ')
    x.job()
```

Η έξοδος του παραπάνω προγράμματος είναι:

```
I am John Doe : I play games
I am Messi : I play football
I am Galis : I play basketball
```

Παρατηρούμε ότι ανάλογα με το είδος του αντικειμένου που περνάμε στην `player_sport` καλείται και η κατάλληλη έκδοση της μεθόδου `job`.

Το παραπάνω δεν θα δουλέψει, αν περάσουμε στην συνάρτηση `player_sports` κάποιο αντικείμενο από άλλη κλάση που δεν διαθέτει τις μεθόδους `get_name` και `job`. Μπορούμε να τροποποιήσουμε την `player_sports` έτσι ώστε να ελέγχει αν το αντικείμενο στο όρισμά της είναι του κατάλληλου τύπου χρησιμοποιώντας τη συνάρτηση `isinstance` της Python.

```
def player_sport(x):
    if isinstance(x, Player):
        print('I am', x.get_name(), end=' : ')
        x.job()
    else:
        print('Not a player')
```

Η `isinstance(αντικείμενο, Κλάση)` ελέγχει αν το `αντικείμενο` είναι στιγμιότυπο της `Κλάσης`. Ένα αντικείμενο είναι και στιγμιότυπο όλων των υπερκλάσεων του, αλλά όχι το ανάποδο.

```
isinstance(p, Player)                  # True
isinstance(f, Player)                  # True
isinstance(b, Player)                  # True
isinstance(f, FootballPlayer)          # True
isinstance(f, BasketballPlayer)        # False
isinstance(p, FootballPlayer)          # False
isinstance(p, BasketballPlayer)        # False
```