ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ Η/Υ
σε Python

Διάλεξη 8: Αντικειμενοστραφής Προγραμματισμός

Μάρκος Ζάμπογλου, mzampoglou@aegean.gr

Προγραμματιστικά μοντέλα

1. **Δομημένος Προγραμματισμός** (Structured Programming): Χρησιμοποιούμε δομές επιλογής και επανάληψης αντί για `GoTo`
2. **Διαδικαστικός Προγραμματισμός** (Procedural Programming): χωρίζουμε τη λύση σε βήματα που τα υλοποιούμε με *διαδικασίες/συναρτήσεις*
3. **Αντικειμενοστραφής Προγραμματισμός** (Object-Oriented Programnming, OOP): Ο κώδικας οργανώνεται σε **Αντικείμενα** που ανήκουν σε **Κλάσεις**

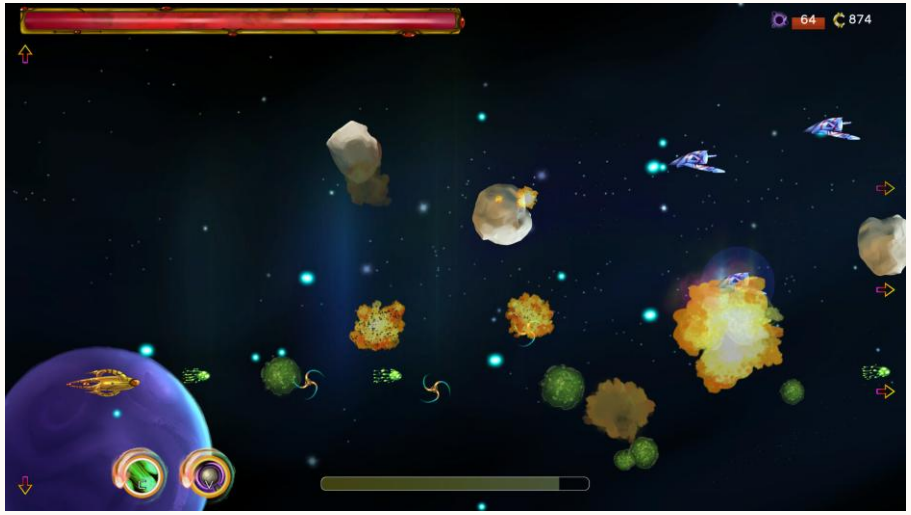
Ο Αντικειμενοστραφής Προγραμματισμός αντικαθιστά τον Διαδικαστικό.

- Και οι δύο ενσωματώνουν το Δομημένο Προγραμματισμό.

Αντικείμενα και Κλάσεις

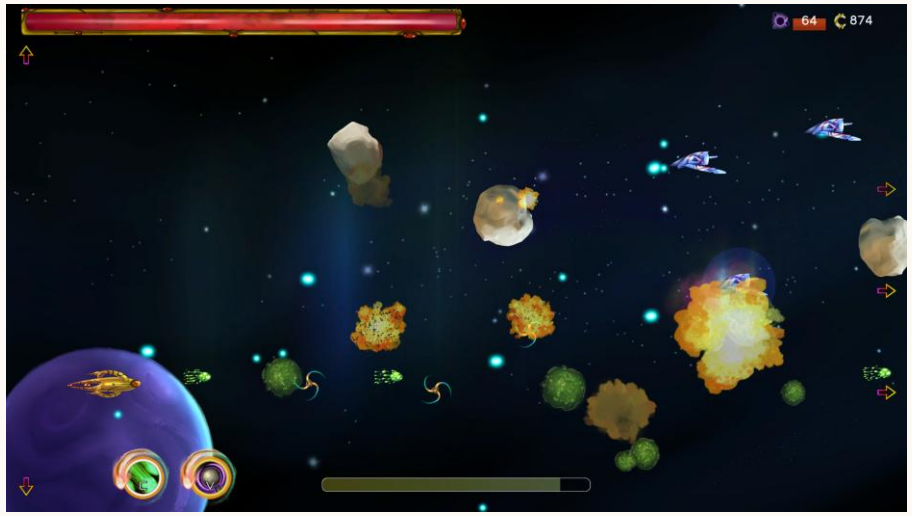
- **Κλάση (Class)**: ένα πρότυπο οργάνωσης δεδομένων και λειτουργιών
  - **Ιδιότητα (Attribute)**: ένα πεδίο που περιέχει δεδομένα του αντικειμένου
  - **Μέθοδος (Method)**: μια λειτουργία (δλδ κώδικας, εντολές) του αντικειμένου
- **Αντικείμενο (Object)**: ένα *στιγμιότυπο* μιας κλάσης

Παράδειγμα



- Υλοποιώ ένα παιχνίδι 2D Shooter
- Αν προσπαθήσω να το γράψω σε διαδικαστικό προγραμματισμό:
  - Δεδομένα σε κλασικές δομές δεδομένων (π.χ. λίστες)
    - Μια λίστα με όλα τα Health των αντιπάλων
    - Μια άλλη με όλα τα Damage των όπλων τους
    - Μια άλλη με όλα τα όπλα του παίχτη…
  - Συναρτήσεις που συνδέουν όλα αυτά μεταξύ τους
    - Εξέλιξη του χρόνου, αλληλεπιδράσεις μεταξύ αντικειμένων κλπ
- Το αποτέλεσμα γίνεται σύντομα χαοτικό

Η αντικειμενοστραφής προσέγγιση



Οργανώνω τα πάντα ως **αντικείμενα**:

- Μια κλάση **`Enemy`**
  - Ιδιότητες **`health`**, **`speed`**, **`weapons`** κλπ
  - Μέθοδοι **`take_damage()`**, **`shoot()`**, **`move()`** κλπ
- Μια κλάση **`Player`**
  - Παρόμοιες ιδιότητες και μέθοδοι με την **`Enemy`**, συν **`special_abilities`**, **`level`** κλπ
- Επιπλέον κλάσεις για κάθε στοιχείο του παιχνιδιού
  - Π.χ. **`HealthBar`** για την οριζόντια μπάρα, **`CombatBackground`** για την εικόνα που σκρολάρει από πίσω κλπ
- **Το κάθε τι** είναι ένα αντικείμενο
  - Περιέχει τα **δεδομένα** του ως ιδιότητες
  - Εφαρμόζει τις **μεθόδους** του πάνω σε αυτά (παίρνοντας **παραμέτρους** από άλλα αντικείμενα)
- Τα αντικείμενα της εφαρμογής μπορούν να "περιέχονται" το ένα μέσα στο άλλο
  - Π.χ. η ιδιότητα **`weapons`** της **`Player`** είναι μια λίστα από αντικείμενα της κλάσης **`Weapon`**, τα οποία «εκτοξεύουν» αντικείμενα της κλάσης **`Projectile`**.

Ορίζοντας μια νέα κλάση

```
class MyClass:
  # Μια ιδιότητα με όνομα data
  data = [0, 0, 0]
  # Μια μέθοδος με όνομα show_output
  def show_output(self):
    print(f"Η πρώτη μου μέθοδος.")
```

Αυτό το **self** είναι πολύ σημαντικό –θα το δούμε σε λίγο.

Οι ιδιότητες και οι μέθοδοι είναι στην πραγματικότητα μεταβλητές και συναρτήσεις που ορίζονται εντός της κλάσης.

```
a = MyClass()
print(a)
<__main__.MyClass object at 0x7ef6dc0b6720>
print(type(a))
<class '__main__.MyClass'>
```

Τύπος αντικειμένου

- Κάποιες φορές είναι σημαντικό να ελέγξω την κλάση ενός αντικειμένου
- Αυτό το πετυχαίνω με τη συνάρτηση **`isinstance()`**

```
print(isinstance(a, MyClass))
True
print(isinstance(a.data, tuple))
False
```

Δημιουργώντας δικές μου βιβλιοθήκες

- Μπορώ να πάρω τον κώδικα της κλάσης **MyClass** και να τον αποθηκεύσω σε ένα διακριτό αρχείο **.py**, π.χ. με όνομα **my_lib.py**
- Θα της δώσω διαφορετικό όνομα, π.χ. **MyClass2**. Και θα αλλάξω το μήνυμα της **show_output()**.
- Με τον τρόπο αυτό, έχω φτιάξει τη βιβλιοθήκη **my_lib** από την οποία μπορώ να εισαγάγω την κλάση **MyClass2**.

```
import my_lib
b = my_lib.MyClass2()
b.show_output()
Η δεύτερή μου μέθοδος.
print(dir(my_lib))
['MyClass2',…]
```

Εναλλακτικά:

```
from my_lib import MyClass2
c = MyClass2()
c.show_output()
```

Χρησιμοποιώντας ιδιότητες και καλώντας μεθόδους

Γνωρίζουμε ήδη πώς γίνεται αυτό: με τη χρήση της τελείας (.)

```
# Τροποποιώ την ιδιότητα
a.data[0] = -100
# Παίρνω την ιδιότητα για επεξεργασία
print(a.data)
[-100, 0, 0]
# Εκτελώ μια μέθοδο
a.show_output()
Η πρώτη μου μέθοδος.
```

Όταν δεν προσδιορίζουμε για τι ιδιότητες μιλάμε, κανονικά εννοούμε ιδιότητες των αντικειμένων. Οι ιδιότητες κλάσης είναι η *εξαίρεση*.

Ιδιότητες της κλάσης
(δηλαδή: όχι των αντικειμένων)

```
b=MyClass()
print(b.data)
[-100, 0, 0]
```

- Δημιουργώντας ένα νέο αντικείμενο **b** της MyClass, η ιδιότητα **b.data** παίρνει την τιμή που είχα δώσει στο **a.data** κι όχι την αρχική **[0, 0, 0]**
- Λέμε ότι η data είναι ιδιότητα όλης της κλάσης **MyClass**
- Όμως συνήθως θέλω κάθε αντικείμενο να έχει ανεξάρτητη τιμή στις ιδιότητές του
  - Π.χ. κάθε εχθρός έχει το δικό του ανεξάρτητο **health**

Ιδιότητες αντικειμένων

Χρήση των `__init__()` και `self`

Η `__init__()` εκτελείται αυτόματα κατά τη δημιουργία του αντικειμένου

`self` σημαίνει «του εκάστοτε αντικειμένου και όχι όλης της κλάσης»

`self.data` σημαίνει «απευθύνομαι/δημιουργώ την ιδιότητα **data** μόνο του συγκεκριμένου αντικείμενου»

```
class MyClass2:
  def __init__(self):
    self.data = [0, 0, 0]
  def increase(self):
    for i in range(len(self.data)):
      self.data[i]+=1
```

Εδώ, καλείται αυτόματα η `__init__()` είτε την έχω ορίσει μέσα στην κλάση είτε όχι – αν όχι, εκτελείται κενή και δεν κάνει τίποτα.

Τόσο η `__init__()` όσο και η `increase()` φαίνεται να καλούνται χωρίς παραμέτρους, όμως στέλνουν πάντα μια αόρατη `self` που σημαίνει «σε εσένα, το αντικείμενο που απευθύνομαι»

```
a = MyClass2()
a.increase()
print(a.data)
[1, 1, 1]
b = MyClass2()
print(b.data)
[0, 0, 0]
```

Αρχικοποίηση ιδιοτήτων

- Έχουμε δει πως μπορούμε να δημιουργήσουμε αντικείμενα δίνοντας αρχικές τιμές
  - Π.χ. η `a=set([2,5,6,2,8])` δημιουργεί ένα **set** από τα στοιχεία της λίστας
- Μπορούμε να κάνουμε το ίδιο και στα δικά μας αντικείμενα:

```
class MyClass3:
  def __init__(self, value1, value2):
    self.attrib = value1
    self.attrib2 = value2

a = MyClass3(5,-2.5)
print(a.attrib)
5
print(a.attrib2)
-2.5
```

Το **5** ανατίθεται στο **value1** και το **-2.5** στο **value2**. Η πρώτη από τις τρεις παραμέτρους, η **self**, ανατίθεται **αόρατα**.

Παράδειγμα 8.1

Θέλω να δημιουργήσω μια εφαρμογή διαχείρισης χρηστών. Να γραφεί μια κλάση σε Python με όνομα **`User`** που να έχει:

- δύο ιδιότητες, **`username`** και **`password`**, οι οποίες θα παίρνουν τιμή κατά τη δημιουργία του αντικειμένου
- μία μέθοδο, την **`validate_password(self, password)`** που θα δέχεται ένα string και θα ελέγχει αν είναι ο σωστός κωδικός

Παράδειγμα 8.1

Θέλω να δημιουργήσω μια εφαρμογή διαχείρισης χρηστών. Να γραφεί μια κλάση σε Python με όνομα **`User`** που να έχει:

- δύο ιδιότητες, **`username`** και **`password`**, οι οποίες θα παίρνουν τιμή κατά τη δημιουργία του αντικειμένου
- μία μέθοδο, την **`validate_password(self, password)`** που θα δέχεται ένα string και θα ελέγχει αν είναι ο σωστός κωδικός

```
class User:
  def __init__(self, user, passw):
    self.username = user
    self.password = passw

  def validate_password(self,passw):
    if self.password == passw:
      return True
    else:
      return False
```

Ξανά για τις ιδιότητες κλάσης

Είδαμε πως υπάρχουν ιδιότητες που ανήκουν στην κλάση και όχι στο κάθε αντικείμενο/στιγμιότυπο:

```
class MyClass:
  data = [0, 0, 0]
a = MyClass()
a.data[0] = -100
b=MyClass()
print(b.data)
[-100, 0, 0]
```

Στην πραγματικότητα, μπορούμε να εμφανίσουμε την ιδιότητα **data** κατευθείαν από την κλάση **MyClass**, χωρίς να μπούμε σε κάποιο στιγμιότυπο:

```
print(MyClass.data)
[-100, 0, 0]
```

Μέθοδοι κλάσης

Με αντίστοιχο τρόπο μπορούμε να έχουμε μεθόδους κλάσης αντί για μεθόδους του κάθε στιγμιότυπου:

```
class Counter:
    total_count = 0

    def __init__():
        self.count = 0

    def add(self):
        self.count += 1
        self.__class__.total_count += 1

    @classmethod
    def get_total_count(cls):
        return cls.total_count
```

Ιδιότητα της κλάσης

Ιδιότητα του στιγμιοτύπου

Κάθε στιγμιότυπο έχει μια ιδιότητα `__class__` που μας επιστρέφει το όνομα της κλάσης του ώστε να μη χρειάζεται να το γράφουμε εμείς.

Έτσι μπορεί το στιγμιότυπο να απευθύνεται στην ίδια την κλάση του.

Θα μπορούσα να γράψω `Counter.total_count += 1` αλλά είναι προτιμότερο να πάρω το όνομα αυτόματα.

Στην Python οι λέξεις που ξεκινάνε με @ λέγονται **decorators** και μας επιτρέπουν να εμπλουτίζουμε συναρτήσεις με επιπλέον λειτουργίες.

Όπως οι μέθοδοι στιγμιοτύπου φέρνουν το στιγμιότυπο ως παράμετρο με τη `self`, έτσι και οι μέθοδοι κλάσης φέρνουν την κλάση ως παράμετρο με την `cls`.

Παράδειγμα 8.2

Να γραφεί μια κλάση **`BankClient`** που θα αντιστοιχεί σε έναν τραπεζικό πελάτη. Θα έχει μια ιδιότητα **`balance`** που θα ξεκινάει 0 και θα κρατάει το υπόλοιπο του λογαριασμού του, μια μέθοδο **`deposit()`** που θα δέχεται ένα ποσό και θα το προσθέτει στο υπόλοιπο, και μια μέθοδο **`withdraw()`** που θα δέχεται ένα ποσό και θα το αφαιρεί από το υπόλοιπο.

Η κλάση να έχει επίσης μια στατική ιδιότητα **`all_clients`** που θα είναι μια λίστα με όλους τους πελάτες. Κάθε φορά που δημιουργείται ένας νέος πελάτης, να προστίθεται στη λίστα. Τέλος, να έχει και μια στατική μέθοδο **`sum_balances()`** που θα επιστρέφει το άθροισμα όλων των καταθέσεων.

Παράδειγμα 8.2

Να γραφεί μια κλάση **BankClient** που θα αντιστοιχεί σε έναν τραπεζικό πελάτη. Θα έχει μια ιδιότητα **balance** που θα ξεκινάει 0 και θα κρατάει το υπόλοιπο του λογαριασμού του, μια μέθοδο **deposit()** που θα δέχεται ένα ποσό και θα το προσθέτει στο υπόλοιπο, και μια μέθοδο **withdraw()** που θα δέχεται ένα ποσό και θα το αφαιρεί από το υπόλοιπο.

Η κλάση να έχει επίσης μια στατική ιδιότητα **all_clients** που θα είναι μια λίστα με όλους τους πελάτες. Κάθε φορά που δημιουργείται ένας νέος πελάτης, να προστίθεται στη λίστα. Τέλος, να έχει και μια στατική μέθοδο **sum_balances()** που θα επιστρέφει το άθροισμα όλων των καταθέσεων.

```
class BankClient:
  all_clients = []
  @classmethod
  def sum_balances(cls):
    all_balances = [client.balance for
                client in cls.all_clients]
    return sum(all_balances)

  def __init__(self):
    self.balance = 0
    self.__class__.all_clients.append(self)
  def deposit(self,amount):
    self.balance+=amount
  def withdraw(self,amount):
    self.balance-=amount
```

Στατικές μέθοδοι

- Με αντίστοιχο τρόπο μπορούμε να έχουμε στατικές μεθόδους:

```
class TemperatureConverter:
    @staticmethod
    def celsius_to_fahrenheit(celsius):
        return (celsius * 9/5) + 32

    @staticmethod
    def fahrenheit_to_celsius(fahrenheit):
        return (fahrenheit - 32) * 5/9

print(TemperatureConverter.celsius_to_fahrenheit(100))
212
```

Αντί για τον decorator `@classmethod` χρησιμοποιούμε τον decorator `@staticmethod`.

- Όπως μια μέθοδος κλάσης, έτσι και μια στατική μέθοδος δε χρειάζεται κάποιο στιγμιότυπο της κλάσης για να κληθεί
  - Μπορεί να κληθεί από την ίδια την κλάση
  - Μπορεί να κληθεί και από ένα στιγμιότυπο αν το θέλουμε
- Όμως, δεν έχει «εσωτερική» πρόσβαση στις μεταβλητές της κλάσης όπως έχει η μέθοδος κλάσης που έχει την παράμετρο **`cls`**
- Τις στατικές μεθόδους τις χρησιμοποιούμε για λειτουργίες γενικής χρήσης που σχετίζονται με την κλάση αλλά δεν απαιτούν να γνωρίζουν την εκάστοτε τρέχουσα κατάστασή της

Στατικές μέθοδοι
σε ενσωματωμένα αντικείμενα

- Για να κατανοήσουμε το ρόλο των στατικών μεθόδων, ας δούμε μια από τις ελάχιστες στατικές μεθόδους που υπάρχουν σε ενσωματωμένη κλάση της Python: τη **`float.fromhex()`**
- Η **`.fromhex`** δέχεται ένα string που περιέχει ένα δεκαεξαδικό αριθμό, και επιστρέφει ένα float

```
print(float.fromhex('0x3a.0c'))
```

- Είναι μια χρήσιμη μέθοδος, αλλά δεν σχετίζεται με κάποιο συγκεκριμένο αντικείμενο, ούτε έχει ανάγκη κάποιο δεδομένο πέρα από την είσοδό του
- Μπορούμε να πούμε ότι σχετίζεται γενικώς με τους floating point numbers καθώς δέχεται και εκθετική μορφή στο string εισόδου)
  - ...οπότε μπήκε στην κλάση float

Κλάσεις και ενσωματωμένες συναρτήσεις

- Τα built-in αντικείμενα της Python μπορούν να μπουν ως είσοδοι σε διάφορες ενσωματωμένες συναρτήσεις της Python:

```
a = '3'
print(2+int(a))
b=[3,6,1]
print(len(b))
```

- Βέβαια, όχι όλα σε όλες…

```
c=3.4
print(len(c))
```

- Ως τώρα, τα αντικείμενα που φτιάχνουμε εμείς δεν συνεργάζονται με αυτές τις συναρτήσεις
- Μπορούμε να το πετύχουμε με τις ***ειδικές*** *ή* ***μαγικές*** μεθόδους
  - Magic ή *Special Methods*
    - Ή και *Dunder* (από το "Double Underscore") *Methods*
  - Με ονόματα όπως `__int__(),__len__()` κλπ

Python Magic Methods
Παράδειγμα

- Έχω μια κλάση **BookShelf** που έχει μια ιδιότητα **Books**, που είναι οι τίτλοι των βιβλίων που έχω βάλει στο συγκεκριμένο ράφι
  - Θέλω η μέθοδος **len()** να μου επιστρέφει το πλήθος των βιβλίων του ραφιού

```
class BookShelf:
  def __init__(self):
    self.books = []
  def add_book(self, book_title):
    self.books.append(book_title)
  def __len__(self):
    return len(self.books)
```

Σε αυτό το παράδειγμα χρησιμοποιώ τη **len** για να μετρήσω τα βιβλία. Σε άλλες περιπτώσεις, θα μπορούσε η **__len__()** να επιστρέφει έναν οποιονδήποτε άλλο αριθμό –είναι στην ευχέρειά μου.

Python Magic Methods 1/4

Το πλήθος των magic methods είναι υπερβολικά μεγάλο για να τις απαριθμήσουμε

- Μερικές βασικές κατηγορίες:
  - **Αριθμητικές** (`__round__()`, `__abs__()`, `__ceil__()`, `floor__()`...)
  - **Πράξεων** (`__add__()`, `__sub__()`, `__pow__()`...)
  - **Σύγκρισης** (`__eq__()`, `__gt__()`, `__lt__()`...)
  - **Σύνθετων αντικειμένων** (`__contains__()`, `__len__()`...)
  - **Μετατροπής τύπου** (`__str__()`, `__int__()`, `__bool__()`...)
- Και πάρα πολλές άλλες:
  - https://www.reddit.com/r/Python/comments/br9ok2/list_of_all_python_dunder_methods/
  - https://github.com/thebjorn/pymagic/blob/master/magicmethods.py
  - https://www.geeksforgeeks.org/python/dunder-magic-methods-python/

Python Magic Methods 2/4

- **Αριθμητικές**
  - **`__round__(self)`**, **`__abs__(self)`**, **`__ceil__(self)`**, **`floor__(self)`**: Ορίζουν τι θα επιστρέφει η συνάρτηση με το αντίστοιχο όνομα
- **Πράξεων**
  - **`__add__(self,other)`**: Ορίζει τι θα επιστρέφει η πράξη **`self + other`**
  - **`__sub__(self,other)`**: Η πράξη **`self - other`**
  - **`__mul__(self,other)`**: Η πράξη **`self * other`**
  - **`__truediv__(self,other)`**: Η πράξη **`self / other`**
  - **`__mod__(self,other)`**: Η πράξη **`self % other`**
  - **`__div__(self,other)`**: Η πράξη **`self // other`**
  - **`__pow__()`** : Η **`pow(self,other)`**

Python Magic Methods 3/4

- **Σύγκρισης**
  - **`__eq__(self,other)`**: Ορίζει τι θα επιστρέφει η πράξη **`self == other`**
  - **`__ne__(self,other)`**: Η πράξη **`self != other`**
  - **`__lt__(self,other)`**: Η πράξη **`self < other`**
  - **`__gt__(self,other)`**: Η πράξη **`self > other`**
  - **`__le__(self,other)`**: Η πράξη **`self <= other`**
  - **`__ge__(self,other)`**: Η πράξη **`self >= other`**
- **Σύνθετων αντικειμένων**
  - **`__contains__(self,other)`**: Ορίζει τι θα επιστρέφει η *λογική* πράξη **`other in self`**
  - **`__len__(self)`**: Ορίζει τι θα επιστρέφει η **`len()`**
  - **`__iter__(self)`**, **`__next__(self)`**, **`__reversed__(self)`**: Χρησιμοποιούνται για να κάνω τα αντικείμενά μου iterables (π.χ. να λειτουργούν με **`for i in my_object:`**) *–εκτός ύλης*

Python Magic Methods 4/4

- **Μετατροπής τύπου**
  - **`__str__(self)`**: Ορίζει τι θα επιστρέφει η **`str()`**
  - **`__int__(self)`**: Ορίζει τι θα επιστρέφει η **`int()`**
  - **`__float__(self)`**: Ορίζει τι θα επιστρέφει η **`float()`**
  - **`__bool__(self)`**: Ορίζει τι θα επιστρέφει η **`bool()`**
- **Εμφάνιση εκτός string**
  - **`__repr__(self)`**: Ορίζει πώς θα εμφανίζεται εντός σύνθετων αντικειμένων και στην κονσόλα

Δεν υπάρχουν magic functions για τις λογικές πράξεις **`and`**, **`or`**, **`not`** κλπ. Κατά τις πράξεις αυτές, εκτελείται πρώτα η bool στο αντικείμενο. Οπότε αν θέλουμε να επηρεάσουμε το αποτέλεσμα τους, καθορίζουμε τη **`__bool__()`**.

Παράδειγμα 8.3

- Να γραφεί μια κλάση με όνομα **`CarModel`** που θα αντιστοιχεί στο μοντέλο ενός αυτοκινήτου. Θα έχει ιδιότητες **`name`**, **`top_speed`** και **`fuel_consumption`**, που θα ορίζονται ως παράμετροι κατά την αρχικοποίηση.
- Χρησιμοποιήστε μαγικές μεθόδους ώστε η σύγκριση δυο μοντέλων με την **>** ή την **<** να θεωρεί ως "μεγαλύτερο" αυτό με την υψηλότερη τελική ταχύτητα.
- Κατά τη μετατροπή ενός **`CarModel`** σε string, το string να παίρνει τη μορφή:

"{**`name`**}. Top speed: {**`top_speed`**}.
Fuel consumption: {**`fuel_consumption`**}."

Παράδειγμα 8.3

- Να γραφεί μια κλάση με όνομα **CarModel** που θα αντιστοιχεί στο μοντέλο ενός αυτοκινήτου. Θα έχει ιδιότητες **name**, **top_speed** και **fuel_consumption**, που θα ορίζονται ως παράμετροι κατά την αρχικοποίηση.
- Χρησιμοποιήστε μαγικές μεθόδους ώστε η σύγκριση δυο μοντέλων με την **>** ή την **<** να θεωρεί ως "μεγαλύτερο" αυτό με την υψηλότερη τελική ταχύτητα.
- Κατά τη μετατροπή ενός **CarModel** σε string, το string να παίρνει τη μορφή:

"{**name**}. Top speed: {**top_speed**}.
Fuel consumption: {**fuel_consumption**}."

```
class CarModel:
  def __init__(self, name, top_speed,
                          fuel_consumption):
    self.name = name
    self.top_speed = top_speed
    self.fuel_consumption = fuel_consumption
  def __lt__(self,other):
    return self.top_speed < other.top_speed
  def __gt__(self,other):
    return self.top_speed > other.top_speed
  def __str__(self):
    return f"Model name: {self.name}. Top
        speed: {self.top_speed}. Fuel
        consumption: {self.fuel_consumption}."
```

Σύγκριση αντικειμένων 1/2

- Η εμπειρία μας είναι πως αν συγκρίνουμε δυο αντικείμενα με ίδια στοιχεία (π.χ. `[2,4,6]==[2,4,6]`) το αποτέλεσμα θα είναι **`True`**
- Αυτή δεν είναι η προεπιλεγμένη συμπεριφορά για αντικείμενα.
  - Π.χ. η `CarModel("L1",240,5)==CarModel("L1",240,5)` θα επιστρέψει **`False`**
- Αν θέλω να επιστρέψει **`True`**, θα πρέπει να υλοποιήσω την **`__eq__()`** της **`CarModel()`** και να γράψω κατάλληλο κώδικα

Σύγκριση αντικειμένων 2/2 🤓

Η μόνη περίπτωση που η == επιστρέφει εξ' ορισμού **`True`** είναι αν στις δύο πλευρές έχει **το ίδιο αντικείμενο**

```
a=CarModel("L1",240,5)
b=a
print(b==a)
```

```
True
```

Η == λοιπόν μπορεί να επηρεαστεί από την **`__eq__()`**. Αν θέλουμε μια σύγκριση που να επιστρέφει **`True` μόνο** όταν τα αντικείμενα ταυτίζονται, και να μη μπορεί να τροποποιηθεί, υπάρχει η **`is`**.

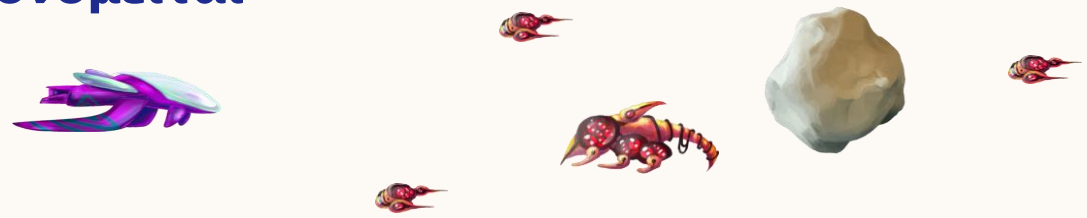
```
print(b is a)
```

```
True
```

Κληρονομικότητα

Ας επιστρέψουμε για μια στιγμή στο space shooter:

- Έχω τρεις κατηγορίες αντικειμένων που μπορούν να καταστραφούν αν τελειώσει η "ζωή" τους
  - Το διαστημόπλοιο του παίχτη
  - Καταστρεφόμενα εμπόδια,
  - Διαστημόπλοια των εχθρών
- Θα ήταν πολύ λάθος να γράψω τον κώδικα «**`take_damage()`**» τρεις διαφορετικές φορές, σε κάθε κλάση
  - Μπορώ να τα ομαδοποιήσω σε μια ιεραρχία από *κλάσεις-υποκλάσεις*, ώστε ο κοινός κώδικας να **κληρονομείται** αυτόματα



Κληρονομικότητα: παράδειγμα

Damageable

Method: "take_damage"
Attributes: "hit_points", "positionXY"

Obstacle: Damageable

Method: "take_damage"
Attributes: "hit_points", "positionXY"
"damage_reduction"

Spaceship: Damageable

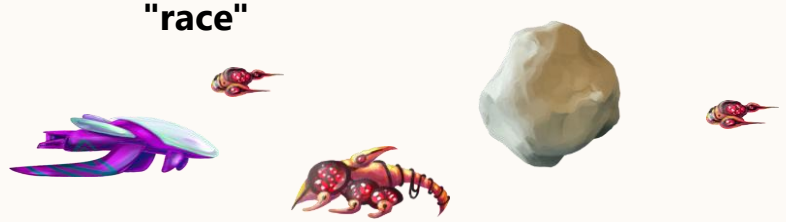
Methods: take_damage", **"shoot"**, **"move"**
Attributes: "hit_points", "positionXY", **"weapons"**

Enemy: Spaceship

Methods: take_damage", "shoot", "move"
Attributes: "hit_points", "positionXY", "weapons"
"race"

Player: Spaceship

Methods: take_damage", "shoot", "move"
Attributes: "hit_points", "positionXY", "weapons"
"specials"



Κληρονομικότητα

- Η **κληρονομικότητα** (inheritance) είναι μια πάρα πολύ σημαντική λειτουργικότητα του αντικειμενοστραφούς προγραμματισμού
- Μας επιτρέπει να δημιουργούμε παράγωγες κλάσεις ή *υποκλάσεις* (derived classes / subclasses) που **κληρονομούν** τις ιδιότητες και τις μεθόδους των βασικών κλάσεων (base classes)
  - Και επιπλέον τους προσθέτουμε τις δικές τους μεθόδους και ιδιότητες

Παράδειγμα: Πανεπιστημιακή κοινότητα

Να γραφεί μια ιεραρχία κλάσεων για τα μέλη του πανεπιστημίου (διοικητικό, φοιτητές/τριες, ΔΕΠ).

- Κάποιες ιδιότητες θα τις έχουν όλα τα άτομα (ονοματεπώνυμο, ηλικία), καθώς και μια μέθοδο **`__str__()`** που θα παράγει ένα string που θα τις περιέχει
- Επιπλέον οι φοιτητές/τριες θα έχουν μια ιδιότητα "αριθμός μητρώου", τα μέλη ΔΕΠ θα έχουν μια ιδιότητα "βαθμίδα", και το διοικ. προσωπικό μια ιδιότητα "θέση"

Σχέδιο: θα δημιουργήσω μια βασική κλάση **`UnivMember`** για όλον τον κόσμο, και τρεις υποκλάσεις **`Admin`**, **`Student`**, **`Prof`** που θα την κληρονομούν

Πρόβλημα: αφού η **`UnivMember`** θα έχει μια **`__init__(self, name, age)`** που θα την κληρονομούν όλες οι άλλες, πώς θα γίνει η **`Student`** να έχει μια **`__init__(self, name, age, AM)`**;

Λύση: **Υπέρβαση/Επέκταση** (***Overriding/Extending***)

Overriding & Extending

Βασική κλάση / Base class

```
class UnivMember:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def __str__(self):
    return f"Όνομα: {self.name}, Ηλικία:{self.age}"
```

Έτσι φροντίζω η **Admin** να **κληρονομήσει** τις μεθόδους και τις ιδιότητες της **UnivMember**

```
class Admin(UnivMember):
  def __init__(self, name, age, position):
    super().__init__(name,age)
    self.position = position
  def __str__(self):
    return f"Όνομα: {self.name}, Ηλικία:
              {self.age}, Θέση: {self.position}"
```

Η συνάρτηση **super()** επιστρέφει τη γονεϊκή κλάση. Εναλλακτικά θα μπορούσα να γράψω **UnivMember.__init__(name,age)**

Η **__init__()** της **Admin** **υπερβαίνει** την **__init__()** της **UnivMember**

Όμως η **Admin** την καλεί εκ νέου με τις παραμέτρους που χρειάζεται, και προσθέτει την επιπλέον ιδιότητα **position**. Οπότε τελικά η **__init__()** της **Admin** επεκτείνεται με την **position**.

Αντιθέτως η **__str__()** της **Admin** υπερβαίνει τη **__str__()** της **UnivMember** χωρίς να την καλεί μέσα από τον κώδικά της –άρα όταν καλώ τη **__str__()** της **Admin**, η **__str__()** της **UnivMember** αγνοείται.

Παράδειγμα 8.4

Θέλω να επεκτείνω την εφαρμογή διαχείρισης χρηστών του Παραδείγματος **8.1**. Να δημιουργηθεί μια υπο-κλάση της **`User`** με όνομα **`Administrator`** που θα έχει επιπλέον:

- μια ιδιότητα **`privileges`** που θα είναι μια λίστα από strings, π.χ. **`["delete_users", "edit_content"]`** που θα δίνεται κατά την αρχικοποίηση
- μια μέθοδο **`has_privilege()`** που θα δέχεται ένα string και θα ελέγχει αν το string αυτό υπάρχει στη λίστα

Παράδειγμα 8.4

Θέλω να επεκτείνω την εφαρμογή διαχείρισης χρηστών του Παραδείγματος **8.1**. Να δημιουργηθεί μια υπο-κλάση της **`User`** με όνομα **`Administrator`** που θα έχει επιπλέον:

- μια ιδιότητα **`privileges`** που θα είναι μια λίστα από strings, π.χ. **`["delete_users", "edit_content"]`** που θα δίνεται κατά την αρχικοποίηση
- μια μέθοδο **`has_privilege()`** που θα δέχεται ένα string και θα ελέγχει αν το string αυτό υπάρχει στη λίστα

```
class User:
  def __init__(self, user, passw):
    self.username = user
    self.password = passw
  def validate_password(self,passw):
    if self.password == passw:
      return True
    else:
      return False


class Administrator(User):
  def __init__(self, user, passw, priv):
    self.privileges = priv
    User.__init__(self, user, passw)
  def has_privilege(self, privilege):
    if privilege in self.privileges:
      return True
    else:
      return False
```

Ιδιωτικές μεταβλητές & Ενθυλάκωση 🤓

- Στον αντικειμενοστραφή προγραμματισμό, πολλές φορές θα θέλαμε να κρατήσουμε την τιμή μιας ιδιότητας «κρυμμένη» από το υπόλοιπο πρόγραμμα, ώστε να μεταβάλλεται μόνο μέσα από συγκεκριμένες μεθόδους
  - Π.χ. στο παιχνίδι, θα ήταν πιο ασφαλές να μην επιτρέπουμε στην **`Damageable.hit_points`** να τροποποιείται από άλλα αντικείμενα, αλλά μόνο από τις μεθόδους "**`take_damage`**" ή "**`heal`**"
    - Ώστε π.χ. να μπορούμε να ελέγχουμε σε κάθε αλλαγή μήπως έπεσε στο 0
- Στον Αντικειμενοστραφή Προγραμματισμό, η πρακτική αυτή του «κρυψίματος» των ιδιοτήτων μέσα στα αντικείμενα λέγεται **ενθυλάκωση** (*encapsulation*)
  - ...και **η Python δεν την υποστηρίζει**
- Αυτό που κάνουμε είναι, όταν θέλουμε να ενθυλακώσουμε μια ιδιότητα, να της δίνομε όνομα που ξεκινάει με τους χαρακτήρες "_" ή "__"
  - Π.χ. **`Damageable._hit_points`**
  - Και αυτό λειτουργεί ως μήνυμα προς τους developers (και εμάς): «*μην πειράζετε την τιμή της ιδιότητας αυτής εξωτερικά*»

Last but not least: Αντιγράφοντας αντικείμενα

- Τα αντικείμενα που δημιουργούμε είναι εξ' ορισμού **mutable**
- Αυτό σημαίνει ότι η ανάθεση γίνεται **by reference**:

```
a = CarModel("L1",240,5)
b = a
b.top_speed = 180
print(a.top_speed)
180
```

- Αν θέλουμε να αντιγράψουμε ένα αντικείμενο by value, χρειαζόμαστε τη βιβλιοθήκη **`copy`**
- Η να υλοποιήσουμε τη δική μας μέθοδο **`.copy()`** όπως υλοποιούν τα ενσωματωμένα mutable αντικείμενα (λίστες, λεξικά, σύνολα)