Εισαγωγή στον Προγ/μό Υπολογιστών

Διάλεξη 9

Αντικειμενοστραφής προγραμματισμός
(*object-oriented* programming)

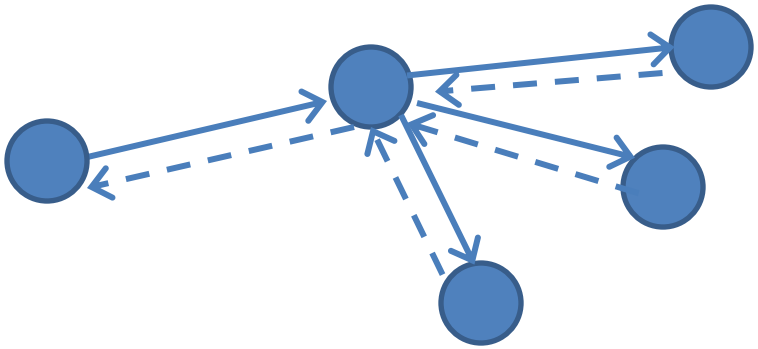
Περιεχόμενα

Αντικείμενα και *τάξεις*

Αντικειμενοστραφής προγραμματισμός

- ***Object-oriented programming***: τρόπος προγραμματισμού όπου βασικό δομικό στοιχείο είναι τα *αντικείμενα*

  – Το πρόγραμμα δομείται από *αντικείμενα* όπου μέσω της ανταλλαγής μηνυμάτων παράγονται χρήσιμοι υπολογισμοί



Τάξεις

- Κάθε *αντικείμενο* έχει ένα ορισμένο τύπο (***τάξη***): *αντικείμενα* της ίδιας *τάξης* ανταποκρίνονται στα ίδια μηνύματα

- Τα *αντικείμενα* που έχουμε χρησιμοποιήσει μέχρι τώρα ανήκουν σε κάποια από τις ενσωματωμένες *τάξεις* της Python

```
>>> type(12)
<class 'int'>
>>> type('hello')
<class 'str'>
>>> type(pow)
<class 'builtin_function_or_method'>
```

Τάξεις

- Ο προγραμματιστής μπορεί να ορίσει τις δικές του *τάξεις*
  - Λειτουργούν ως «καλούπι» κατασκευής *αντικειμένων* της *τάξης* αυτής

- Ο ορισμός περιγράφει τα μηνύματα που μπορούν να λάβουν τα αντικείμενα της τάξης και πως αυτά μεταβάλλουν την κατάστασή τους

- Παράδειγμα: *αντικείμενα* που αναπαριστούν τραπεζικό λογαριασμό, θα πρέπει να:
  - Έχουν τιμή του τρέχοντος *υπολοίπου*
  - Έχουν όνομα *κατόχου*
  - Υπάρχει δυνατότητα *ανάληψης*
  - Υπάρχει δυνατότητα *κατάθεσης*

Τάξεις και *αντικείμενα*

- Παράδειγμα: *τάξη* `Account` για τραπεζικούς λογαριασμούς

  – Κατασκευή *αντικειμένου* που αναπαριστά λογαριασμό που ανήκει στον κάτοχο με όνομα `'Luke'`:

```
>>> a = Account('Luke')
```

  – Η κλήση σε συνάρτηση με ακριβώς ίδιο όνομα με μια *τάξη*, κατασκευάζει και επιστρέφει ένα *αντικείμενο* της *τάξης* αυτής
  – Τέτοιες συναρτήσεις λέγονται ***κατασκευαστές*** **(***constructors***)**
  – Το *αντικείμενο* που κατασκευάστηκε λέγεται ***στιγμιότυπο*** (*instance*) της *τάξης* `Account`

Τάξεις και *αντικείμενα*

- Κάθε *αντικείμενο* έχει **ιδιότητες (*attributes*)**: ονόματα που συνδέονται σε τιμές που αφορούν το *αντικείμενο*

- Οι τιμές των *ιδιοτήτων* μπορούν να επιστραφούν με *εκφράσεις με τελεία* (*dot expressions*)

```
>>> a.holder
'Luke'
>>> a.balance
0
```

Τάξεις και *αντικείμενα*

- Όλα τα *αντικείμενα* μιας *τάξης* έχουν τις ίδιες *ιδιότητες*

```
>>> b = Account('Yoda')
>>> b.holder
'Yoda'
>>> b.balance
0
```

  - ***Ιδιότητες στιγμιοτύπου* (*instance attributes*)**: *ιδιότητες* όπου μπορούν να λάβουν διαφορετικές τιμές σε διαφορετικά *αντικείμενα* (της ίδιας *τάξης*), πχ. `holder, balance`

  - ***Ιδιότητες τάξης* (*class attributes*):** *ιδιότητες* που έχουν την ίδια τιμή σε όλα τα αντικείμενα της ίδιας *τάξης*

Τάξεις και *αντικείμενα*

- ***Μέθοδοι*:** *ιδιότητες* που είναι συναρτήσεις που εκτελούν υπολογισμούς για το *αντικείμενο* που αφορούν ή/και *μεταλλάσσουν* την *κατάστασή* του

```
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.withdraw(10)
'Insufficient funds'
```

- Τα *αντικείμενα* `Account` είναι *μεταλλασσόμενα δεδομένα* αφού η ίδια κλήση επέφερε διαφορετικό αποτέλεσμα

Ορισμός *τάξεων*

Ορισμός *τάξεων*

- Μη ενσωματωμένες *τάξεις* ορίζονται με την εντολή `class`

- Γενική μορφή: `class <όνομα>:`
  `<μπλόκ εντολών>`

- Στο μπλοκ περιέχονται εντολές που ορίζουν τις *ιδιότητες* των *αντικειμένων* της *τάξης* και πως αυτές αλλάζουν στη λήψη *μηνυμάτων* (*κλήση μεθόδων*)
  - Εκτελούνται κατά την εκτέλεση της εντολής `class`
  - Οι *ιδιότητες* ορίζονται με εντολές που *συνδέουν* ονόματα σε τιμές, όπως `def` και *ανάθεση* (=)

Ορισμός *τάξεων*

- Ο ορισμός των *instance attributes* γίνεται στον ορισμό συνάρτησης με το ειδικό όνομα `__init__` (στην Python) και ονομάζεται ***κατασκευαστής*** της *τάξης*

```
>>> class Account:
        def __init__(self, account_holder):
```

Στην *κλήση* `Account('Luke')` γίνονται τα εξής:

1. Δημιουργείται νέο *αντικείμενο* της *τάξης* `Account`
2. Καλείται η συνάρτηση `__init__`, όπου
   - Η πρώτη τυπική παράμετρος (`self`) συνδέεται στο νέο αντικείμενο που μόλις δημιουργήθηκε
   - Οι υπόλοιπες παράμετροι συνδέονται στα ορίσματα της κλήσης `Account('Luke')`

Ορισμός *τάξεων*

- Ο ορισμός των *instance attributes* γίνεται στον ορισμό συνάρτησης με το ειδικό όνομα `__init__` (στην Python) και ονομάζεται ***κατασκευαστής*** της *τάξης*

```
>>> class Account:
        def __init__(self, account_holder):
            self.holder = account_holder
            self.balance = 0
```

Στην *κλήση* `Account('Luke')` γίνονται τα εξής:

3. Ορισμός *instance attributes*: με εντολές = συνδέονται τα ονόματα `holder, balance` σε τιμές

4. Επιστρέφεται το *αντικείμενο* που δημιουργήθηκε

Ορισμός τάξεων

```
1  class Account:
2      def __init__(self, account_holder):
3          self.holder = account_holder
4          self.balance = 0
5
6  a = Account('Luke')
7
```

μετάβαση στο Pythontutor

Ορισμός *τάξεων*

- Διαφορετικές κλήσεις του κατασκευαστή φτιάχνουν διαφορετικά αντικείμενα
  - Το κάθε ένα έχει το δικό του όνομα κατόχου και τιμή υπολοίπου

```
1  class Account:
2      def __init__(self, account_holder):
3          self.holder = account_holder
4          self.balance = 0
5
6  a = Account('Luke')
7  b = Account('Yoda')
8
```

μετάβαση στο Pythontutor

Ορισμός *τάξεων*

- Η κατάσταση ενός *αντικειμένου* δεν εξαρτάται από το όνομα στο οποίο έχει συνδεθεί

```
>>> a = Account('Luke')
>>> a.deposit(20)
20
>>> c = a
>>> c is a
True
>>> c.withdraw(15)
5
>>> a.withdraw(10)
'Insufficient funds'
```

Ορισμός *τάξεων*

- Οι *μέθοδοι* ορίζονται με συναρτήσεις μέσα στον ορισμό *τάξης*

```
>>> class Account:
        def __init__(self, account_holder):
            self.holder = account_holder
            self.balance = 0

        def deposit(self, amount):
                    . . .

        def withdraw(self, amount):
                    . . .
```

Ορισμός *τάξεων*

- Για τον ορισμό μιας *μεθόδου* ορίζεται μια συνάρτηση ως *ιδιότητα τάξης* (*class attribute*)

```
...
 def deposit(self, amount):
     self.balance = self.balance + amount
     return self.balance
...
```

Στην κλήση `a.deposit(10)` γίνονται τα εξής:

1. Καλείται η *συνάρτηση* `deposit` της *τάξης* του `a` (`Account`)
   - Η πρώτη *τυπική παράμετρος* (`self`) συνδέεται στην τιμή του `a`
   - Οι υπόλοιπες παράμετροι συνδέονται στα *ορίσματα* της *κλήσης* της *μεθόδου*

Ορισμός *τάξεων*

```
1  class Account:
2      def __init__(self, account_holder):
3          self.holder = account_holder
4          self.balance = 0
5
6      def withdraw(self, amount):
7          if amount > self.balance:
8              return 'Insufficient funds'
9          self.balance = self.balance - amount
10         return self.balance
11
12     def deposit(self, amount):
13         self.balance = self.balance + amount
14         return self.balance
15
16 a = Account('Luke')
17 a.deposit(30)
18 b = Account('Yoda')
19 b.deposit(50)
```

μετάβαση στο Pythontutor

Εκφράσεις με τελεία (dot expressions)

Εκφράσεις με τελεία

- ***Εκφράσεις με τελεία* (*dot expressions*)**:
  - Γενική μορφή: <έκφραση>.<όνομα>
  - Τιμή = η τιμή της *ιδιότητας* <όνομα> του *αντικειμένου* στο οποίο αποτιμάται η <έκφραση>

- Οι τιμές των *dot expressions* μπορούν ισοδύναμα να υπολογιστούν με τη συνάρτηση `getattr`:

```
>>> a = Account('Luke')
>>> getattr(a, 'holder')
'Luke'
>>> getattr(a, 'deposit')(25)
25
>>> hasattr(a, 'withdraw')
True
```

Συναρτήσεις και *μέθοδοι*

- Οι *μέθοδοι* μπορούν να *κληθούν* με δύο ταυτόσημους τρόπους:

```
>>> a = Account('Luke')
>>> a.deposit(15)
15
>>> type(a.deposit)
<class 'method'>
```

```
>>> a = Account('Luke')
>>> Account.deposit(a, 15)
15
>>> type(Account.deposit)
<class 'function'>
```

Συναρτήσεις και *μέθοδοι*

- Η *έκφραση* `Account.deposit` έχει ως τιμή την *ιδιότητα* (*τάξης*) `deposit` της *τάξης* `Account`, η οποία είναι συνάρτηση με δύο *ορίσματα*

- Η *έκφραση* `a.deposit` εχει ως τιμή τη συνάρτηση `deposit` της *τάξης* `Account`, όπου το πρώτο της *όρισμα* έχει εκ των προτέρων *ανατεθεί* στο *αντικείμενο* `a`
  - Η `a.deposit` είναι ισοδύναμη της `a_deposit`:

```
>>> a = Account('Luke')
>>> a_deposit = lambda x: Account.deposit(a, x)
>>> a_deposit(15)
15
>>> a.deposit(15)
30
```

Συμβάσεις ονομάτων στην Python

- Στα ονόματα *τάξεων* χρησιμοποιείται κεφαλαίο πρώτο γράμμα σε κάθε λέξη που απαρτίζει το όνομα. Ανάμεσα σε δύο λέξεις δεν τοποθετείται _
  - Πχ, `Account, CheckingAccount`

- Με _ αρχίζουν *ιδιότητες* που δεν θα πρέπει να χρησιμοποιούνται από τμήματα κώδικα εκτός *τάξης*. (*Ιδιωτικές ιδιότητες*)

Ιδιότητες τάξης

Ιδιότητες τάξης

- Οι *τιμές* των *ιδιοτήτων τάξης* είναι κοινές για όλα τα *αντικείμενα* της *τάξης*
  - Ορίζονται με εντολή ανάθεσης, `def` ή `import` που δεν βρίσκεται μέσα σε *μέθοδο* – και συνεπώς εκτελείται όταν ορίζεται η *τάξη*

- Παράδειγμα: τραπεζικός λογαριασμός με επιτόκιο καταθέσεων (`interest`) κοινό για όλους τους λογαριασμούς

```
>>> class Account:
        interest = 0.02
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        def withdraw(self, amount):
            ...
        def deposit(self, amount):
            ...
```

Ιδιότητες τάξης

```
>>> a = Account('Luke')
>>> b = Account('Yoda')
>>> a.interest
0.02
>>> b.interest
0.02
```

- Η τιμή μιας *ιδιότητας τάξης* αλλάζει ταυτόχρονα σε όλα τα *αντικείμενα* της ίδιας *τάξης*

```
>>> Account.interest = 0.04
>>> a.interest
0.04
>>> b.interest
0.04
```

Ιδιότητες τάξης

```
1  class Account:
2      interest = 0.02
3      def __init__(self, account_holder):
4          self.holder = account_holder
5          self.balance = 0
6
7  a = Account('Luke')
8  print(a.interest)
9  b = Account('Yoda')
10 print(b.interest)
11
12 Account.interest = 0.04
13 print(a.interest)
14 a.interest = 0.08
15 print(b.interest)
```

μετάβαση στο Pythontutor

Ιδιότητες τάξης

Python 3.6

```
1  class Account:
2      interest = 0.02
3      def __init__(self, account_holder):
4          self.holder = account_holder
5          self.balance = 0
6      def deposit(self, amount):
7          self.balance = self.balance + amount
8          return self.balance
9      def withdraw(self, amount):
10         if amount > self.balance:
11             return 'Insufficient funds'
12         self.balance = self.balance - amount
13         return self.balance
14
15 a = Account('Luke')
16 print(a.balance)
17 print(Account.interest)
18 print(a.interest)
19 Account.deposit(a, 20)
20 a.deposit(10)
```

Print output (drag lower right corner to resize)

Frames Objects

μετάβαση στο Pythontutor

Ιδιότητες τάξης

Python 3.6

```
1  class Account:
2      interest = 0.02
3      def __init__(self, account_holder):
4          self.holder = account_holder
5          self.balance = 0
6      def deposit(self, amount):
7          self.balance = self.balance + amount
8          return self.balance
9      def withdraw(self, amount):
10         if amount > self.balance:
11             return 'Insufficient funds'
12         self.balance = self.balance - amount
13         return self.balance
14
15 a = Account('Luke')
16 print(a.balance)
17 print(Account.interest)
18 print(a.interest)
19 Account.deposit(a, 20)
20 a.deposit(10)
```

Print output (drag lower right corner to resize)

Frames Objects

Global frame

Account

Account class
hide attributes

__init__	function __init__(self, account_holder)
deposit	function deposit(self, amount)
interest	0.02
withdraw	function withdraw(self, amount)

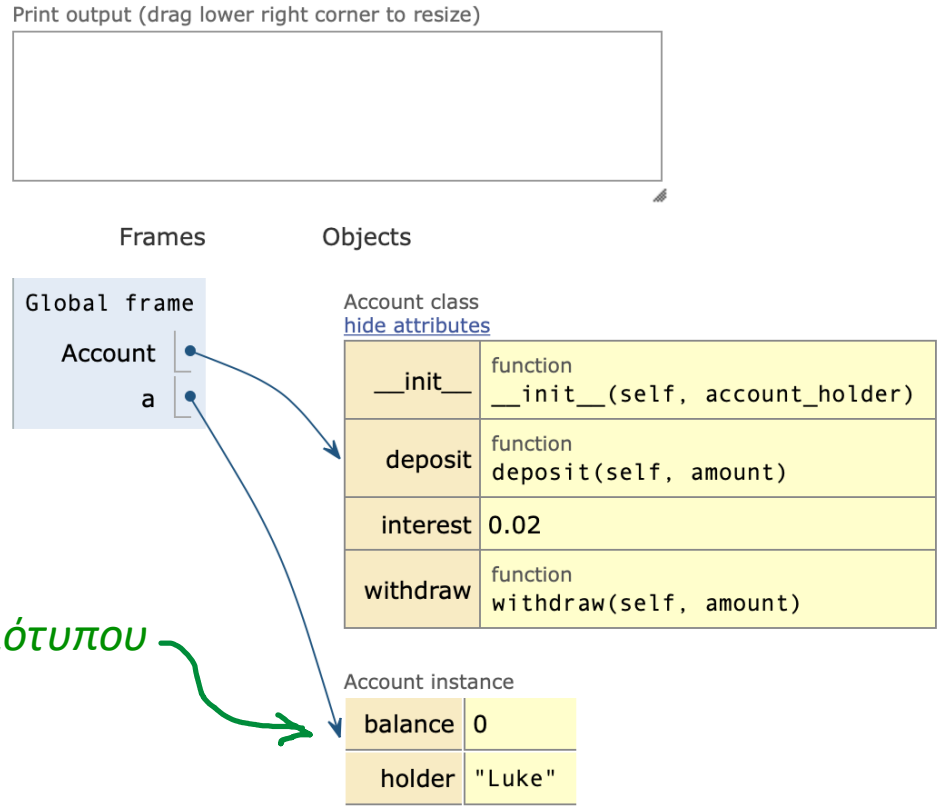
μετάβαση στο Pythontutor

Ιδιότητες τάξης

Python 3.6

```
1  class Account:
2      interest = 0.02
3      def __init__(self, account_holder):
4          self.holder = account_holder
5          self.balance = 0
6      def deposit(self, amount):
7          self.balance = self.balance + amount
8          return self.balance
9      def withdraw(self, amount):
10         if amount > self.balance:
11             return 'Insufficient funds'
12         self.balance = self.balance - amount
13         return self.balance
14
15 a = Account('Luke')
16 print(a.balance)
17 print(Account.interest)
18 print(a.interest)
19 Account.deposit(a, 20)
20 a.deposit(10)
```

ιδιότητα στιγμιότυπου



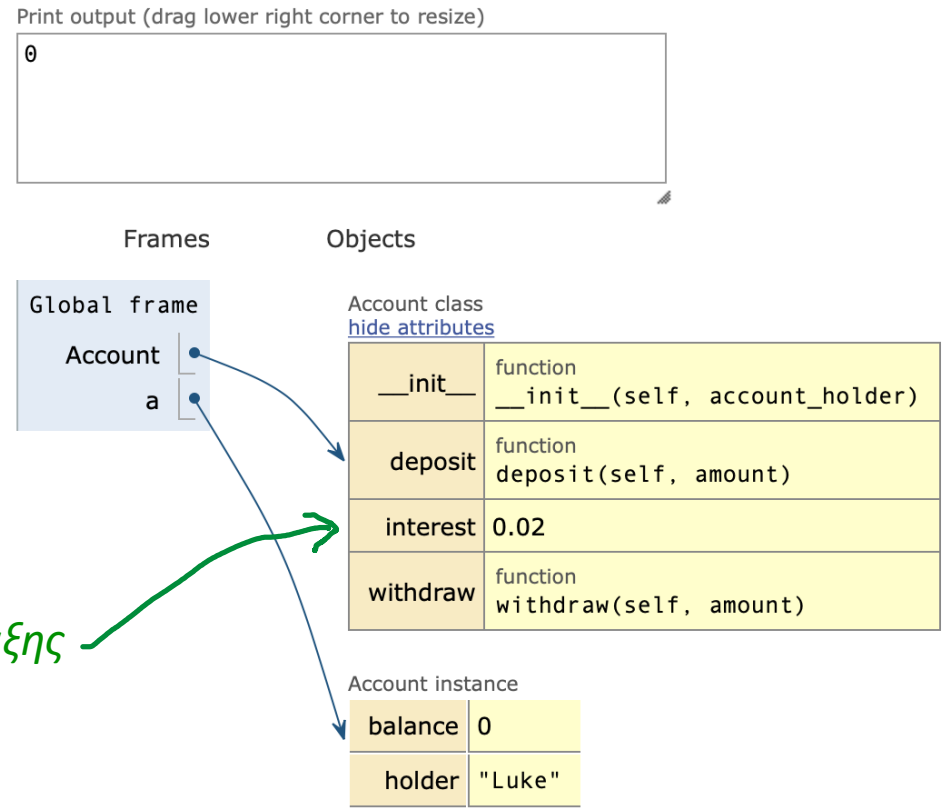
μετάβαση στο Pythontutor

Ιδιότητες τάξης

Python 3.6

```
1  class Account:
2      interest = 0.02
3      def __init__(self, account_holder):
4          self.holder = account_holder
5          self.balance = 0
6      def deposit(self, amount):
7          self.balance = self.balance + amount
8          return self.balance
9      def withdraw(self, amount):
10         if amount > self.balance:
11             return 'Insufficient funds'
12         self.balance = self.balance - amount
13         return self.balance
14
15 a = Account('Luke')
16 print(a.balance)
17 print(Account.interest)
18 print(a.interest)
19 Account.deposit(a, 20)
20 a.deposit(10)
```

ιδιότητα τάξης



μετάβαση στο Pythontutor

Ιδιότητες τάξης

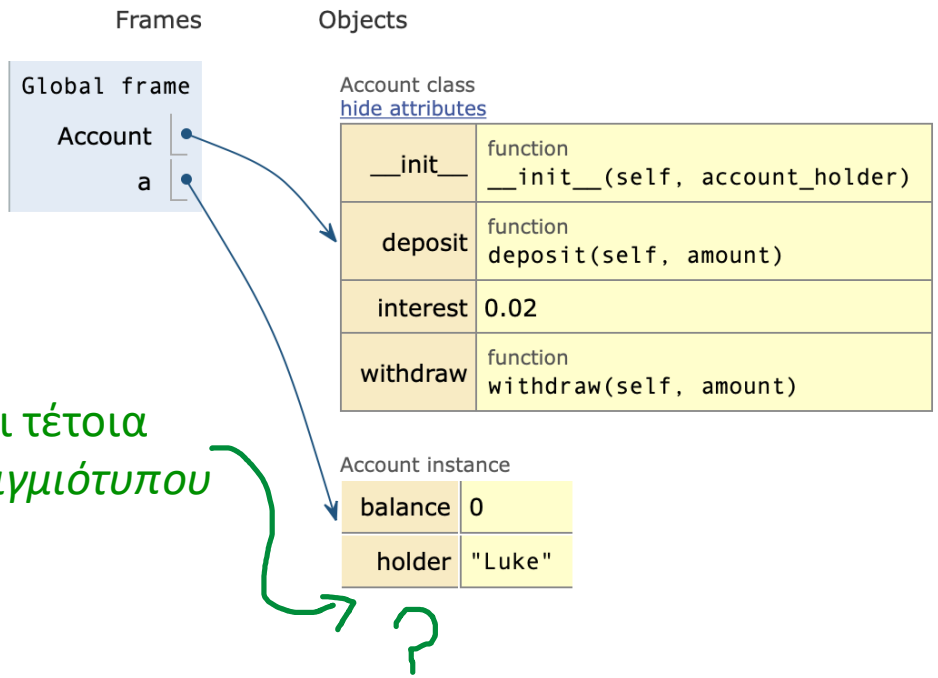
Python 3.6

```
1  class Account:
2      interest = 0.02
3      def __init__(self, account_holder):
4          self.holder = account_holder
5          self.balance = 0
6      def deposit(self, amount):
7          self.balance = self.balance + amount
8          return self.balance
9      def withdraw(self, amount):
10         if amount > self.balance:
11             return 'Insufficient funds'
12         self.balance = self.balance - amount
13         return self.balance
14
15 a = Account('Luke')
16 print(a.balance)
17 print(Account.interest)
18 print(a.interest)
19 Account.deposit(a, 20)
20 a.deposit(10)
```

δεν υπάρχει τέτοια *ιδιότητα στιγμιότυπου*

Print output (drag lower right corner to resize)

```
0
0.02
```



μετάβαση στο Pythontutor

Ιδιότητες τάξης

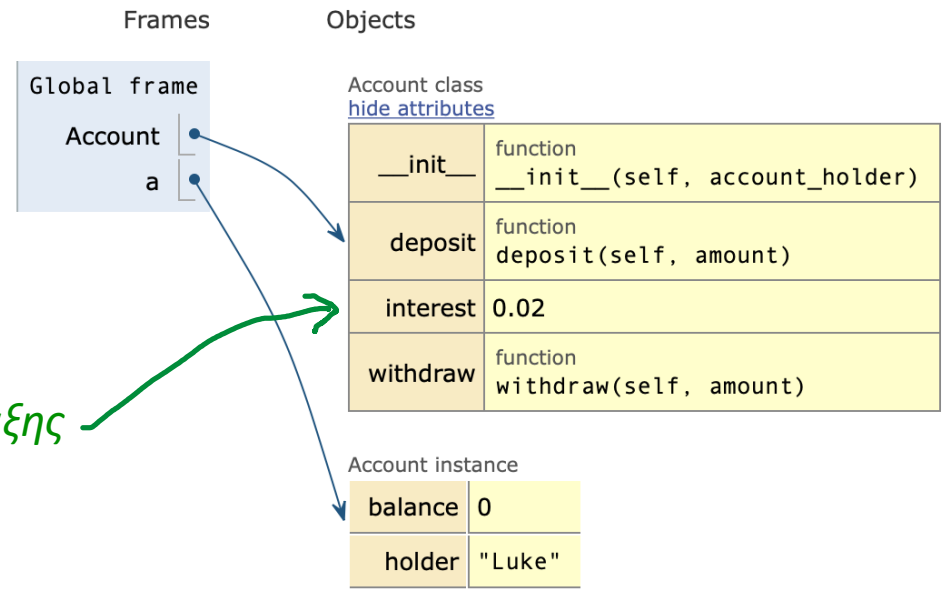
Python 3.6

```
1  class Account:
2      interest = 0.02
3      def __init__(self, account_holder):
4          self.holder = account_holder
5          self.balance = 0
6      def deposit(self, amount):
7          self.balance = self.balance + amount
8          return self.balance
9      def withdraw(self, amount):
10         if amount > self.balance:
11             return 'Insufficient funds'
12         self.balance = self.balance - amount
13         return self.balance
14
15 a = Account('Luke')
16 print(a.balance)
17 print(Account.interest)
18 print(a.interest)
19 Account.deposit(a, 20)
20 a.deposit(10)
```

ιδιότητα τάξης

Print output (drag lower right corner to resize)

```
0
0.02
```



μετάβαση στο Pythontutor

Ιδιότητες τάξης

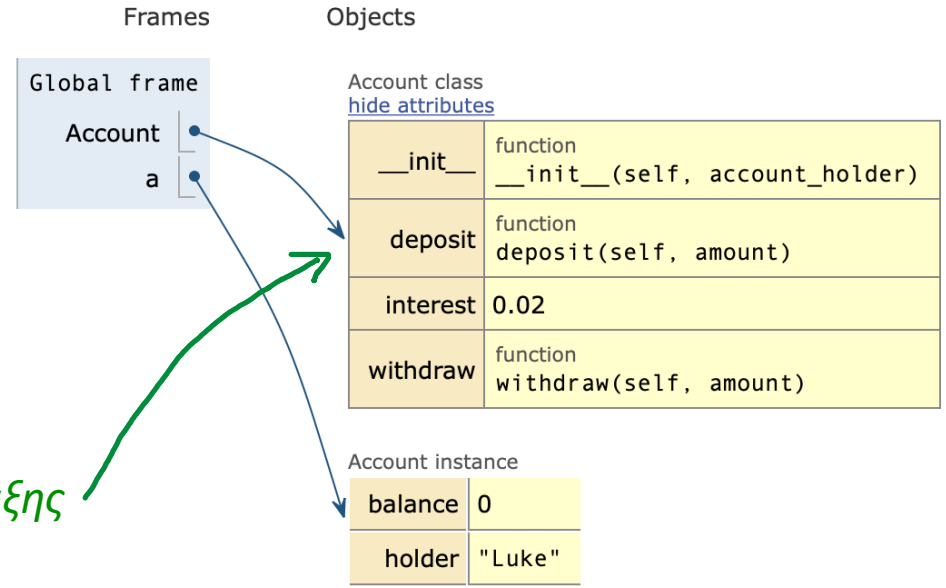
Python 3.6

```
1  class Account:
2      interest = 0.02
3      def __init__(self, account_holder):
4          self.holder = account_holder
5          self.balance = 0
6      def deposit(self, amount):
7          self.balance = self.balance + amount
8          return self.balance
9      def withdraw(self, amount):
10         if amount > self.balance:
11             return 'Insufficient funds'
12         self.balance = self.balance - amount
13         return self.balance
14
15 a = Account('Luke')
16 print(a.balance)
17 print(Account.interest)
18 print(a.interest)
19 Account.deposit(a, 20)
20 a.deposit(10)
```

ιδιότητα τάξης

Print output (drag lower right corner to resize)

```
0
0.02
0.02
```



μετάβαση στο Pythontutor

Ιδιότητες τάξης

Python 3.6

```
1  class Account:
2      interest = 0.02
3      def __init__(self, account_holder):
4          self.holder = account_holder
5          self.balance = 0
6      def deposit(self, amount):
7          self.balance = self.balance + amount
8          return self.balance
9      def withdraw(self, amount):
10         if amount > self.balance:
11             return 'Insufficient funds'
12         self.balance = self.balance - amount
13         return self.balance
14
15 a = Account('Luke')
16 print(a.balance)
17 print(Account.interest)
18 print(a.interest)
19 Account.deposit(a, 20)
20 a.deposit(10)
```

Print output (drag lower right corner to resize)

```
0
0.02
0.02
```

Frames

Global frame

Account

a

Objects

Account class
hide attributes

__init__	function `__init__(self, account_holder)`
deposit	function `deposit(self, amount)`
interest	0.02
withdraw	function `withdraw(self, amount)`

Account instance

balance	20
holder	"Luke"

δεν υπάρχει τέτοια *ιδιότητα στιγμιότυπου*

μετάβαση στο Pythontutor

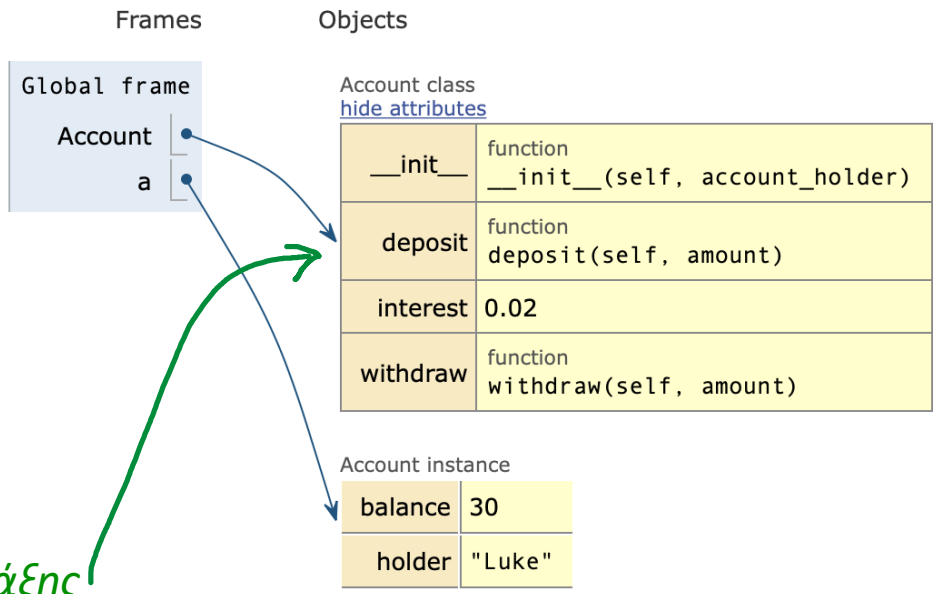
Ιδιότητες τάξης

Python 3.6

```
1  class Account:
2      interest = 0.02
3      def __init__(self, account_holder):
4          self.holder = account_holder
5          self.balance = 0
6      def deposit(self, amount):
7          self.balance = self.balance + amount
8          return self.balance
9      def withdraw(self, amount):
10         if amount > self.balance:
11             return 'Insufficient funds'
12         self.balance = self.balance - amount
13         return self.balance
14
15 a = Account('Luke')
16 print(a.balance)
17 print(Account.interest)
18 print(a.interest)
19 Account.deposit(a, 20)
20 a.deposit(10)
```

Print output (drag lower right corner to resize)

```
0
0.02
0.02
```



Υπάρχει *ιδιότητα τάξης* με το ίδιο όνομα:
η τιμή του `a.deposit` είναι
`lambda amount: Account.deposit(a, amount)`

μετάβαση στο Pythontutor

Ιδιότητες τάξης

- Αποτίμηση *ιδιοτήτων*:

  `<έκφραση>.<όνομα>`

  1. Η αποτίμηση της *έκφρασης* αριστερά της `.` δίνει το *αντικείμενο* που αφορά η *ιδιότητα*
  2. Επιστρέφεται η τιμή της *ιδιότητας στιγμιοτύπου* `<όνομα>` εάν υπάρχει για το *αντικείμενο*
  3. Εάν δεν υπάρχει τέτοια *ιδιότητα στιγμιοτύπου*, επιστρέφεται η τιμή της *ιδιότητας τάξης* `<όνομα>` εάν υπάρχει
  4. Εάν η ιδιότητα είναι συνάρτηση, επιστρέφεται η συνάρτηση αυτή αφού *συνδεθεί* στο αντικείμενο της έκφρασης (δηλ., ως μέθοδος)

- Εάν η *ιδιότητα* βρίσκεται στο αριστερό μέλος μιας ανάθεσης, ορίζεται η *ιδιότητα στιγμιοτύπου* `<όνομα>` *μόνο* στο *αντικείμενο* της *έκφρασης*

Κληρονομικότητα

Κληρονομικότητα

- Διαφορετικές *τάξεις* μπορεί να σχετίζονται

- Μια συνήθης σχέση είναι η μια *τάξη* να είναι *εξειδίκευση* μιας άλλης:
  Η πιο εξειδικευμένη τάξη:
  1. έχει τις ίδιες *ιδιότητες* με τη γενικότερη *τάξη*,
  2. αλλάζει (εξειδικεύει) τη συμπεριφορά ορισμένων *μεθόδων*
  3. διαθέτει επιπλέον *ιδιότητες*

  Παράδειγμα:
  - `Account`: γενικός *λογαριασμός*
  - `CheckingAccount`: *λογαριασμός όψεως* όπου χρεώνεται €1 σε κάθε ανάληψη και έχει χαμηλότερο επιτόκιο

Κληρονομικότητα

- Η *τάξη* `CheckingAccount` εξειδικεύει την `Account`
  - Η `Account` ονομάζεται ***βασική τάξη*** **(*base class*)**
  - Η `CheckingAccount` ονομάζεται ***παραγόμενη τάξη*** **(*derived class*)**

```
>>> class CheckingAccount(Account):
        withdraw_charge = 1
        interest = 0.01
        def withdraw(self, amount):
            return Account.withdraw(self, amount + \
                                    self.withdraw_charge)
```

Κληρονομικότητα

- Παράδειγμα: λογαριασμός όψεως `CheckingAccount` όπου χρεώνεται €1 σε κάθε ανάληψη και έχει χαμηλότερο επιτόκιο:

```
>>> ch = CheckingAccount('Han')
>>> ch.interest
0.01
>>> ch.deposit(20)
20
>>> ch.withdraw(10)
9
```

Κληρονομικότητα

Αποτίμηση *ιδιοτήτων*:

1. Μια *ιδιότητα* ενός *αντικειμένου* αναζητείται ανάμεσα στις *ιδιότητες στιγμιότυπου*
2. Αν δεν βρεθεί, η αναζήτηση συνεχίζεται ανάμεσα στις *ιδιότητες τάξης*
3. Αν δεν βρεθεί, η αναζήτηση συνεχίζεται ανάμεσα στις *ιδιότητες τάξης της βασικής τάξης*
4. Αν δεν βρεθεί, η αναζήτηση συνεχίζεται ανάμεσα στις *ιδιότητες τάξης της βασικής της βασικής τάξης*

…

Κληρονομικότητα

```
1  class Account:
2      interest = 0.02
3      def __init__(self, account_holder):
4          self.holder = account_holder
5          self.balance = 0
6      def withdraw(self, amount):
7          if amount > self.balance:
8              return 'Insufficient funds'
9          self.balance = self.balance - amount
10         return self.balance
11     def deposit(self, amount):
12         self.balance = self.balance + amount
13         return self.balance
14
15 class CheckingAccount(Account):
16     withdraw_charge = 1
17     interest = 0.01
18     def withdraw(self, amount):
19         return Account.withdraw(self, amount + self.withdraw_charge)
20
21 ch = CheckingAccount('Leia')
```

μετάβαση στο Pythontutor

Interfaces

Interfaces

- Δύο *τάξεις* μπορεί να σχετίζονται γιατί υποστηρίζουν το ίδιο *interface*
- ***interface***: σύνολο από *ιδιότητες*

  – Παράδειγμα: τάξη `PiggyBank`

```
>>> class PiggyBank:
        def __init__(self):
            self.amount = 0
        def deposit(self, x):
            self.amount = self.amount + x
>>> winners = [Account('Luke'), PiggyBank(),\
               CheckingAccount('Leia')]
>>> for winner in winners:
        winner.deposit(5)
```

Interfaces

- Τα *αντικείμενα* των *τάξεων* `PiggyBank` και `Account` έχουν *μέθοδο* `deposit` και όλα ανταποκρίνονται στην *κλήση* `winner.deposit(5)`

- Ο κώδικας
```
for winner in winners:
    winner.deposit(5)
```
λειτουργεί για οποιοδήποτε *αντικείμενο* έχει *μέθοδο* `deposit` με αριθμητικό *όρισμα*

  - *Δεν είναι γνωστό από πριν ποια μέθοδος* `deposit` *θα κληθεί. Εξαρτάται από τον τύπο του αντικειμένου* `winner`

- ***Πολυμορφισμός:*** *η δυνατότητα ο κώδικας να έχει διαφορετική συμπεριφορά (αποτέλεσμα) χρησιμοποιώντας αντικείμενα διαφορετικών τύπων*

Interfaces

- Υλοποίηση που δεν χρησιμοποιεί πολυμορφισμό:

```
>>> winners = [Account('Luke'), PiggyBank()]
>>> for winner in winners:
        Account.deposit(winner, 5)
5
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "lec_defs.py", line 16, in deposit
    self.balance = self.balance + amount
AttributeError: 'PiggyBank' object has no attribute 'balance'
```

  - Για να λειτουργήσει σωστά ο *πολυμορφισμός* θα πρέπει να αναφερόμαστε σε ιδιότητες μέσω *στιγμιοτύπων* – όχι μέσω *τάξεων*

Γενικές (*generic*) συναρτήσεις

Γενικές συναρτήσεις

- **Γενικές (*generic*)** : μπορούν να δεχθούν *ορίσματα* πολλών διαφορετικών τύπων

- Παράδειγμα ενσωματωμένης συνάρτησης `str`:

```
>>> str(5)
'5'
>>> from datetime import date
>>> today = date(2023, 12, 12)
>>> str(today)
'2023-12-12'
>>> str(max)
'<built-in function max>'
```

Γενικές συναρτήσεις

- Πως γνωρίζει τι θα επιστρέψει η `str` για κάθε τύπο δεδομένου; για τύπους που θα δημιουργηθούν στο μέλλον;

- Λύση: *interfaces* και *message passing*
  - Όλα τα *αντικείμενα* διαθέτουν τη *μέθοδο* `__str__`
    - Όλες οι τάξεις *κληρονομούν* την *τάξη* `object` η οποία διαθέτει τη *μέθοδο* `__str__`

```
>>> (5).__str__()
'5'
>>> today.__str__()
'2022-1-28'
>>> max.__str__()
'<built-in function max>'
```

  - Η *συνάρτηση* `str` καλεί τη *μέθοδο* (στέλνει *μήνυμα*) `__str__` στο *αντικείμενο* του *ορίσματός* της
    - Η τιμή επιστροφής της `str` υπολογίζεται από το *αντικείμενο* του *ορίσματος*

Γενικές συναρτήσεις

- Ενδεικτική υλοποίηση `str`

```
>>> def mystr(a):
        return a.__str__()
```

- Χρησιμοποιεί *πολυμορφισμό*

Γενικές συναρτήσεις

```
>>> a = Account('Luke')
>>> str(a)
'<__main__.Account object at 0x10cab85c0>'
```

- Πως μπορούμε να εξειδικεύσουμε τον τρόπο εμφάνισης ενός λογαριασμού;

```
>>> class Account2(Account):
        def __str__(self):
            return 'Holder: {0}, balance: {1}'.format(self.holder,\
                                                      self.balance)
>>> a = Account2('Luke')
>>> a.__str__()
'Holder: Luke, balance: 0'
>>> str(a)
'Holder: Luke, balance: 0'
>>> print(a)
Holder: Luke, balance: 0
```

Γενικές συναρτήσεις

- Οι *γενικές συναρτήσεις* ενσωματώνουν νέους τύπους δεδομένων σε ήδη υπάρχουσες λειτουργίες (πχ, `print`)

- Άλλες γενικές συναρτήσεις: `len`, `repr`, …

- Ο νέος τύπος δεδομένων αρκεί να υλοποιήσει την *ειδική μέθοδο* της Python που αντιστοιχεί στη γενική συνάρτηση

Γενικές συναρτήσεις

- Ειδικές *μέθοδοι* της Python: συναρτήσεις `__<όνομα>__` οι οποίες καλούνται από τον διερμηνευτή της Python σε ειδικές περιπτώσεις
  - *Μέθοδοι* `__init__`, `__str__`, `__len__`, ...

```
>>> str(5)
'5'
>>> (5).__str__()
'5'

>>> len('hello world')
11
>>> 'hello world'.__len__()
11
```

Γενικές συναρτήσεις

- Ειδικές *μέθοδοι* που αντιστοιχούν σε *τελεστές*

```
>>> 'hello world'[6]
'w'
>>> 'hello world'.__getitem__(6)
'w'
>>> 5 + 2
7
>>> (5).__add__(2)
7
>>> 5 * 2
10
>>> (5).__mul__(2)
10
```

- Υπάρχουν αντίστοιχες ειδικές *μέθοδοι* για όλους τους *τελεστές* της Python

Γενικές συναρτήσεις

- Ορισμός τύπου δεδομένου `Ritos` που αναπαριστά ρητούς

```
>>> print(Ritos(1, 2) + Ritos(3, 4))
5/4
>>> Ritos(1, 2) * Ritos(3, 4)
Ritos(3, 8)
```

Γενικές συναρτήσεις

```
>>> class Ritos:
        def __init__(self, a, p):
            from math import gcd
            g = gcd(a, p)
            self.a = a // g
            self.p = p // g
        def __add__(self, r):
            a = self.a * r.p + self.p * r.a
            p = self.p * r.d
            return Ritos(a, p)
        def __mul__(self, r):
            return Ritos(self.a * r.a, self.p * r.p)
        def __str__(self):
            return f'{self.a}/{self.p}'
        def __repr__(self):
            return f'Ritos({self.a}, {self.p})'
```

Παράδειγμα: ρολόϊ

Παράδειγμα: ρολόϊ

- Όπως με κάθε προγραμματιστική τεχνική, αναζητούμε υπολογισμούς και δεδομένα που επαναλαμβάνονται ώστε να ορίσουμε τις κατάλληλες προγραμματιστικές αφαιρέσεις (*αντικείμενα*)

- Ένα ρολόϊ αποτελείται από μετρητές ωρών (hr), λεπτών (min) και δευτερολέπτων (sec)
  - Κάθε μετρητής έχει μια αριθμητική τιμή (κατάσταση)
  - Η τιμή αυξάνεται κατά μια μονάδα
  - Ο μετρητής μηδενίζεται όταν φτάσει μια ανώτατη τιμή
  - Ο μηδενισμός επιφέρει αύξηση του αριστερού μετρητή

Παράδειγμα: ρολόϊ

- Τάξη `Counter`: στιγμιότυπα αναπαριστούν μετρητές

```
class Counter:
        def __init__(self, value, res, next = None):
            self.value = value
            self.res = res
            self.next = next
        def advance(self):
            self.value = (self.value + 1) % self.res
            if self.value == 0 and self.next != None:
                self.next.advance()
        def __str__(self):
            return str(self.value)
```

Παράδειγμα: ρολόϊ

- Τάξη `Clock`: στιγμιότυπα αναπαριστούν ρολόϊ

```
class Clock:
    def __init__(self, h, m, s):
        self.hr = Counter(h, 24)
        self.min = Counter(m, 60, self.hr)
        self.sec = Counter(s, 60, self.min)
    def tick(self):
        self.sec.advance()
    def __str__(self):
        return f'{self.hr}:{self.min}:{self.sec}'
```

Παράδειγμα: ρολόϊ

- Εφαρμογή που χρησιμοποιεί ρολόϊ

```
def run(clock):
    from time import sleep
    while True:
        print(clock, end = '\r')
        sleep(1)
        clock.tick()

run(Clock(23, 59, 50))
```

Παράδειγμα: ρολόϊ

- Βελτίωση 1: ρολόϊ με διψήφιους μετρητές

```
class TwoDigitCounter(Counter):
    def __str__(self):
        return str(self.value // 10) + str(self.value % 10)

class Clock2(Clock):
    def __init__(self, h, m, s):
        self.hr = TwoDigitCounter(h, 24)
        self.min = TwoDigitCounter(m, 60, self.hr)
        self.sec = TwoDigitCounter(s, 60, self.min)

run(Clock2(23, 59, 50))
```

Παράδειγμα: ρολόϊ

- Βελτίωση 2: ρολόϊ όπου ο τύπος των μετρητών είναι παράμετρος του κατασκευαστή

```
class ClockX(Clock):
    def __init__(self, h, m, s, X = Counter):
        self.hr = X(h, 24)
        self.min = X(m, 60, self.hr)
        self.sec = X(s, 60, self.min)

run(ClockX(23, 59, 50, TwoDigitCounter))
```

Παράδειγμα: ρολόϊ

- Επέκταση: ρολόϊ με δυαδικά ψηφία

```
class BinaryCounter(Counter):
    def __str__(self):
        return '0' * (6-self.value.bit_length()-(self.value == 0)) \
                  + bin(self.value)[2:]

run(ClockX(23, 59, 50, BinaryCounter))
```