
ΠΕΡΙΕΧΟΜΕΝΑ:

1. Decorators και Κλάσεις
2. Built-in στολισμός μεθόδων
    1. @classmethod και @staticmethod
    2. @property
3. Προχωρημένη σύνταξη decorators:
    1. decorators με ορίσματα
    2. decorators με πληροφορίες κατάστασης

Πάνος Γ.

Χρυσός Χορηγός Μαθήματος

Μυρτώ Ζ.

Σμαραγδένιος Χορηγός Μαθήματος

**Μπορούμε να διακοσμήσουμε μία κλάση.**

- Η ιδιαιτερότητα είναι ότι διακοσμείται **μόνο ο αρχικοποιητής (init) της κλάσης.**

**Παράδειγμα 1: class_decorator.py**

```
def decorate_with_lines(cls):
    def dec():
        print("-"*20)
        ob = cls()
        print("-"*20)
        return ob
    return dec

@decorate_with_lines
class MyClass:
    def __init__(self):
        print("initializing...")

    def func(self):
        print("in func")

c = MyClass()
c.func()
```

και επειδή ξέρουμε (μαθ. 17) ότι μία συνάρτηση είναι απλά μία κλάση που υλοποιεί την dunder method __call__

- **Μπορούμε να χρησιμοποιήσουμε μια κλάση ως διακοσμητή**, θέτοντας τη λειτουργία του διακοσμητή στην __call__

**Παράδειγμα 2: class_as_decorator.py**

```
class decorate_with_lines:
    def __init__(self, decorated):
        self.decorated = decorated
    def __call__(self):
        print("-"*20)
        ob = self.decorated()
        print("-"*20)
        return ob

@decorate_with_lines
class MyClass:
    def __init__(self):
        print("initializing...")
    def func(self):
        print("in func")

c = MyClass()
c.func()
```

- Αν και υπάρχουν σε πακέτα εφαρμογές decorators κλάσεων, το σύνηθες είναι να στολίζονται μέθοδοι κλάσεων.
- Στις βιβλιοθήκες της Python ορίζονται αρκετοί decorators. Βλέπουμε κάποιους από αυτούς εδώ.
- Επίσης σε πακέτα (όπως π.χ. τα Flask, Django, pytorch), αρκετά στοιχεία τους παραμετροποιούνται μέσω decorators.

Έχουμε ήδη δει τον decorator **@abstractmethod** (του module ABC) στο Μαθ. 18 της βασικής σειράς.

- ο οποίος ορίζει ότι μία μέθοδος είναι αφηρημένη

Οι ακόλουθοί decorators μεθόδων είναι επίσης χρήσιμοι στην πράξη:

- **@classmethod**
  - Στολίζει μία μέθοδο κλάσης (και όχι αντικειμένου), άρα θα πρέπει να επενεργεί σε πληροφορίες που αφορούν συνολικά την κλάση
  - Η μέθοδος που στολίζεται πρέπει να έχει ως πρώτο όρισμα ένα όνομα, το οποίο θα περιέχει την μετα-κλάση της κλάσης (βλ. βίντεο - χρήσιμο για να δημιουργήσουμε παραλλαγές της κλάσης)
- **@staticmethod**
  - Αντίστοιχα, αφορά μέθοδο που επενεργεί στην κλάση και όχι στο αντικείμενο
  - Δεν υπάρχει η πληροφορία της κλάσης (όπως στο @classmethod)
  - Μπορεί να συνδυαστεί με χαρακτηριστικά κλάσης (ορίζονται εκτός του κατασκευαστή και είναι κοινές σε όλα τα αντικείμενα της κλάσης)
  - [Γενικά θεωρείται un-pythonic, βλ. και βίντεο]
- και οι δύο είναι built-in.

**Παράδειγμα 3: classmethod_staticmethod.py**

```
class MyClass:
    objects_initiated = 0

    def __init__(self, x, y):
        MyClass.objects_initiated += 1
        self.x = x
        self.y = y

    @staticmethod
    def print_cnt_objects():
        print(MyClass.objects_initiated)

    @classmethod
    def constructJustX(cls, x):
        print(cls.__name__)
        print(cls.__dict__)
        return cls(x, 0)

ob1 = MyClass(1, 1)
ob2 = MyClass(2, 1)
MyClass.print_cnt_objects()

ob3 = MyClass.constructJustX(3)
MyClass.print_cnt_objects()
```

- **Ένα property** έχει παρόμοια συμπεριφορά με ένα μέλος μίας κλάσης,
  - αλλά **προσφέρει ένα επίπεδο αφαίρεσης επιπλέον (παρόμοιο με τους getter/setter σε OOP γλώσσες)**
- Ένα property ορίζεται ως μέθοδος που επιστρέφει μία τιμή (αντίστοιχα με τον getter) χρησιμοποιώντας τον decorator **@property**.

**Παρατηρήσεις:**

- Τα properties χρησιμεύουν για να μπορούμε να αλλάζουμε την υλοποίηση που προσφέρει η κλάση μας (π.χ. τα μέλη της και την αλληλεπίδρασή τους), χωρις να πειράζουμε τη διεπαφή που δίνουμε σε άλλους προγραμματιστές που χρησιμοποιούν την κλάση μας.
- Και χρησιμοποιούνται κατά κόρον σε πακέτα, βιβλιοθήκες κ.λπ.

**Παράδειγμα 4: property_getter.py**

```
class Salary:
    def __init__(self, amount):
        self._amount = amount
    @property
    def amount(self):
        return self._amount

s = Salary(500)
print(s.amount)
```

- Επεκτείνεται ορίζοντας **setter**:
  - Ορίζουμε καινούργιο decorator με όνομα ίδιο με το όνομα του property ακολουθούμενο από ".setter" και έπειτα μία μέθοδο που αναθέτει τιμή στην «προστατευμένη» μεταβλητή

**Παράδειγμα 5: property_setter.py**

```
class Salary:
    ...
    @amount.setter
    def amount(self, amount):
        if amount > 10000:
            raise ValueError("Too high salary")
        else:
            self._amount = amount
...
s.amount = 600
print(s.amount)
```

- Επεκτείνεται ορίζοντας **deleter**:
  - Ορίζουμε καινούργιο decorator με όνομα ίδιο με το όνομα του property ακολουθούμενο από ".deleter" και έπειτα μία μέθοδο που διαγράφει το «προστατευμένο» μέλος

**Παράδειγμα 6: property_deleter.py**

```
class Salary:
    ...
    @amount.deleter
    def amount(self):
        del self._amount
...
try:
    print(s.amount)
except AttributeError:
    print("property doesn't exist")
```

Μπορούμε να ορίσουμε έναν **decorator με ορίσματα**.

- Η συνάρτηση- διακοσμητής ορίζει μία εσωτερική συνάρτηση που είναι διακοσμητής, με τα επιθυμητά ορίσματα
- Επιστρέφει αυτήν την εσωτερική συνάρτηση.

**Παράδειγμα 7: with_arguments.py**

```
def lines_below(n):
    def line_below(func):
        def line(*args):
            func(*args)
            for _ in range(n):
                for i in range(m):
                    for arg in args:
                        print(arg, end="")
                print()

        return line

    return line_below

@lines_below(n=4)
def my_print2(m, *args):
    print("test")

my_print2(5, "*", "-", "*")
```

Μπορούμε να ορίσουμε **decorator που διατηρεί πληροφορίες για την κατάστασή** του.

- είτε μέσω κλάσης που είναι callable,
- είτε προσθέτοντας δυναμικα μέλη στη συνάρτηση - decorator

**Παράδειγμα 8: with state.py**

```
from math import sqrt

def valid_int_gt_zero(func):
    def decorated(val):
        if not isinstance(val, int):
            raise TypeError("Not an integer")
        elif val < 0:
            raise ValueError("Negative value")
        else:
            decorated.calls += 1
            return func(val)

    decorated.calls = 0
    return decorated

@valid_int_gt_zero
def int_sqrt(val):
    return int(sqrt(val))

print(int_sqrt(2))
print(int_sqrt(4))
print("calls: " + str(int_sqrt.calls)
```

## Παράδειγμα 9: example_timer.py

```
import time
def timeit(func):
    def decorator(*args, **kwargs):
        t0 = time.time()
        ret = func(*args, **kwargs)
        t1 = time.time()
        print("Total time: " + str(t1-t0))
        return ret
    return decorator
```

```
@timeit
def dummy(n):
    for i in range(n):
        pass
dummy(100000000)
```

## Παράδειγμα 10: example_debug.py

```
def debug_args(func):
    def decorator(*args):
        for arg in args:
            print(repr(arg), end=", ")
        return func(*args)

    return decorator
```

```
class MyClass:
    def __init__(self):
        self.x = 5
    def __repr__(self):
        return f"MyClass({self.x})"

@debug_args
def dummy(n, ob):
    pass

dummy(5, MyClass())
```

## Παράδειγμα 11: example_memoization.py

```
def memoize(func):
    def decorator(arg):

        if arg in results_memory:
            return results_memory[arg]
        else:
            results_memory[arg] = func(arg)
            return results_memory[arg]

    results_memory = {}
    return decorator
```

```
@memoize
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1)+fib(n-2)

print(fib(10))
```

## Παράδειγμα 12: multiple decorators.py

```
@timeit
@decorate_with_lines
@valid_int_gt_zero
def fib(n):
    @memoize
    def recursive(n):
        if n == 0:
            return 0
        elif n == 1:
            return 1
        else:
            return recursive(n - 1) + recursive(n - 2)
    print(f"fib({n})={recursive(n)}")
```