ΠΕΡΙΕΧΟΜΕΝΑ:

1. Οι συναρτήσεις ως αντικείμενα
2. Decorators
3. OPC:
    1. Timer
    2. HTML decorating

Νίκος Θ.

Χρυσός Χορηγός Μαθήματος

Πέτρος Γ.

Σμαραγδένιος Χορηγός Μαθήματος

ΜΑΘΗΜΑ 4: Decorators για Συναρτήσεις 1. Συναρτήσεις ως Αντικείμενα

- Έχουμε ήδη δει ότι **οι συναρτήσεις είναι αντικείμενα**
- και ότι δικές μας κλάσεις, μπορούν να συμπεριφέρονται ως συναρτήσεις, αρκεί να τους έχουμε ορίσει τη dunder method __call__ (βλ. και μάθημα 17).

- Έτσι **μία συνάρτηση μπορεί να διοχετευτεί ως όρισμα σε άλλη συνάρτηση.**
- Λέμε μάλιστα ότι μία τέτοια συνάρτηση είναι υψηλότερης τάξεως (**higher-order function**)

Παράδειγμα 1: higher.order.func.py

```
def leibniz_term(n):
    return (-1)**n * 1/(2*(n+1)-1)

def sum_func_terms(leibniz_term, n):
    sum=0
    for i in range(n):
        sum+=leibniz_term(i)

    return 4*sum

for i in range(10,200,10):
    print(sum_func_terms(leibniz_term, i))

print(sum_func_terms(leibniz_term,100000))
```

- Και ισχύει ότι **μία συνάρτηση μπορεί να επιστραφεί και από συνάρτηση** (και είδαμε στο μάθημα 14 μια εφαρμογή αυτού, τα εργοστάσια συναρτήσεων), π.χ.:

```
def factory_power(power):
    def nth_power(number):
        return number ** power

    return nth_power

square = factory_power(2)
print(square(4))

cube = factory_power(3)
print(cube(4))
```

- Η τεχνική του να “δένονται” (**bind**) στην παραγόμενη συνάρτηση δεδομένα (εδώ π.χ. Ο εκθέτης 2), λέγεται “κλειστότητα” (**closure**).
- Κι έχουμε τη δυνατότητα να έχουμε πρόσβαση στις τιμές αυτές μέσω του χαρακτηριστικού closure της παραγόμενης συνάρτησης

Παράδειγμα 2: closure.py

```
square = factory_power(2)
print(square.__closure__[0].cell_contents)
square = factory_power(3)
print(cube.__closure__[0].cell_contents)
```

- Οι **decorators** είναι συναρτήσεις, οι οποίες εμπλουτίζουν τη λειτουργία άλλων συναρτήσεων, χωρίς όμως να τις τροποποιήσουν.
- Υλοποιούνται μέσω εργοστασίων συναρτήσεων.

Παρέχεται η **συντακτική διευκόλυνση** να εφαρμόζεται ο decorator πάνω στη συνάρτηση:

- Απλά θέτοντας **το @ ακολουθούμενο από το όνομα του decorator**, πριν τον ορισμό της συνάρτησης

Παράδειγμα 3: decorator.py

```
def decorate_with_lines(func):
    def dec():
        print("=" * 20)
        func()
        print("=" * 20)

    return dec

def some_func():
    print("I did many things.. ")


decorated_func = decorate_with_lines(some_func)

decorated_func()
```

Παράδειγμα 4: decorator.syntax.py

```
def decorate_with_lines(func):
    def dec():
        print("=" * 20)
        func()
        print("=" * 20)
    return dec

@decorate_with_lines
def some_func():
    print("I did many things.. ")

some_func()
```

Άσκηση 1:

Μπορούμε να έχουμε πολλούς decorators που να “κοσμούν” μία συνάρτηση.
Πειραματιστείτε, ορίζοντας έναν ακόμη decorator στο παράδειγμα 4 (ο οποίος να δημιουργεί ακόμη μία γραμμή, αυτή τη φορά με αστέρια) και έπειτα εφαρμόστε και τους δύο decorators (όπως πριν, απλά θα προηγούνται και οι δύο του ορισμού της συνάρτησης.

- Για να κατασκευάσουμε **decorators για συναρτήσεις που δέχονται παραμέτρους**:
  - Το περίβλημα (εσωτερική συνάρτηση) πρέπει να έχει τα ίδια ορίσματα με την συνάρτηση που διακοσμεί.

Παράδειγμα 5: decorator.func.with.params.py

```
def decorate_gt_zero(func):
    def dec(a,b):
        if a<=0 or b<=0:
            print("Error: both args must be positive!")
        else:
            func(a,b)

    return dec

@decorate_gt_zero
def some_calc(a,b):
    print(a*b)

some_calc(1,4)
some_calc(-1,3)
```

Ενώ εντελώς αντίστοιχα μπορούμε να κατασκευάσουμε και έναν γενικό decorator που δουλεύει ανεξάρτητα από το πλήθος των ορισμάτων της συνάρτησης:

- Χρησιμοποιώντας μεταβλητό αριθμό ορισμάτων (Μάθημα 13)

Παράδειγμα 6: decorator.arbitrary.args.py

```
def decorate_with_lines(func):
    def dec(*args, **kwargs):
        print("=" * 20)
        func(*args, **kwargs)
        print("=" * 20)

    return dec

@decorate_with_lines
def some_func(a,b):
    print(a*b)

some_func(5,10)
```

Άσκηση 2:

Κατασκευάστε έναν decorator που ελέγχει ότι όλα τα ορίσματα είναι ακέραιοι αριθμοί, διαφορετικοί από το μηδέν.
Έπειτα κατασκευάστε μία συνάρτηση που υπολογίζει και επιστρέφει το γινόμενο των ορισμάτων της, αφού πρώτα τη διακοσμήσετε με τον παραπάνω decorator.
[Υπενθύμιση: Με την isinstance() ελέγχουμε τον τύπο δεδομένων]

```python
# opc.timing.py
# source (mod): https://realpython.com/primer-on-python-decorators/

import time

def timer(func):
    """Print the runtime of the decorated function"""
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()    # 1
        value = func(*args, **kwargs)
        end_time = time.perf_counter()      # 2
        run_time = end_time - start_time    # 3
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])

for i in range(0,200,20):
    print(i)
    waste_some_time(i)
```

```python
# opc.html.decorating.py
# source (mod): https://gist.github.com/Zearin/2f40b7b9cfc51132851a

# The decorator to make it bold
def makebold(fn):
    # The new function the decorator returns
    def wrapper():
        # Insertion of some code before and after
        return '<b>' + fn() + '</b>'
    return wrapper

# The decorator to make it italic
def makeitalic(fn):
    # The new function the decorator returns
    def wrapper():
        # Insertion of some code before and after
        return '<i>' + fn() + '</i>'
    return wrapper

@makebold
@makeitalic
def say():
    return 'hello'

print(say())
```