PROBLEMS ACTIVITIES SUBMISSIONS 265_FIJ_MUV2_ROSARIO_JUSTIN JACOB (202503304).

# [CS 11 25.1] Lab 7d – Decorators

**Cheatsheet is available here:** https://oj.dcs.upd.edu.ph/cs11cheatsheet/

Submit solution
[CS 11 25.1] Lab Exercise 7
My submissions

**Points:** 150 (partial)
**Time limit:** 6.0s
**Memory limit:** 2G

**Problem type**
**Allowed languages**
py3

## Problem Statement

In CS 11, you were taught that decorators "decorate" existing functions. Let's take that literally!

Make a decorator that "decorates" the output of a function by framing it with characters.

## Task Details

Your task is to make a decorator **factory** named `decorate`. A decorator factory is basically a decorator that can be configured; that is, you can pass in arguments when using the decorator. Implementation-wise, a *decorator factory* is a function that takes in some parameters and returns a *decorator*, which in turn is a function that takes in a function and returns a function. Got it?

For this problem, `decorate` should take in a `str` as an **optional** parameter; this is a length-$1$ string indicating the character that should be used to decorate the output of the decorated function. If no value is passed into `decorate`, the `str` parameter's value should default to `#`.

When we say "decorate", we mean drawing a rectangle around the decorated function's output with a "margin" of one character. For example, decorating the following output:

```
11
```

with the `#` character gives us:

```
######
#    #
# 11 #
#    #
######
```

Name your file `hope3c.py` and your testing file `test_hope3c.py`.

## Restrictions

- The following symbols are allowed: `iter`, `next`, `map`, `filter`
- The following imports are allowed:
  - `count`, `islice`, `chain`, `takewhile`, `starmap` and `zip_longest` from `itertools`.
  - `cache`, `lru_cache`, `total_ordering`, `partial`, `reduce` and `wraps` from `functools`.
  - `randint`, `randrange and choice` from `random`.
  - `Fraction` from `fractions`.
  - `dataclass` from `dataclasses`.
  - `contextmanager` from `contextlib`.
  - `Enum`, `auto` from `enum`.
- Anonymous functions are allowed.
- Inner functions are allowed.
- Classes, dataclasses, and enums are allowed.
- Recursion is allowed.
- Loops are allowed.
- Generators and comprehensions are allowed.
- Your source code must have at most $1200$ bytes.

## Example Testing

Here's an example testing file.

```
# pyright: strict

from hope3c import decorate


@decorate()
def fun() -> int:
    return 11


assert fun() == """\
######
#    #
# 11 #
#    #
######
"""


@decorate("*")
def fun2(x) -> str:
    return f"What haffen, {x}?"


assert fun2("Vella") == """\
***********************
*                     *
* What haffen, Vella? *
*                     *
***********************
"""

# TODO add more tests here
```

## Constraints

- `decorate` will be used at most $100$ times.

## Scoring

**Note:** New tests may be added and all submissions may be rejudged at a later time. (All future tests will satisfy the constraints.)

- You get $15$ ❤️ points if you solve all test cases where:
  - The function to be decorated requires no keyword arguments.
  - `decorate` will always be used **without** a parameter.
- You get $15$ ❤️ points if you solve all test cases where:
  - The function to be decorated requires no keyword arguments.
  - `decorate` will always be used **with** a parameter.
- You get $30$ 🔴 points if you solve all test cases where:
  - The function to be decorated requires no keyword arguments.
- You get $15$ ❤️ points if you solve all test cases where:
  - `decorate` will always be used **without** a parameter.
- You get $15$ ❤️ points if you solve all test cases where:
  - `decorate` will always be used **with** a parameter.
- You get $60$ 🔴 points if you solve all test cases.

## Clarifications

Report an issue

No clarifications have been made at this time.