Decorators

Overview

In this lab you'll define two decorators:

- A decorator to count calls to a function
- A decorator to cache results to a function

Source folders

Student folder : `C:\PythonDev\Student\AppxB-Decorators`

Solution folder: `C:\PythonDev\Solutions\AppxB-Decorators`

Roadmap

There are 3 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1) Defining a decorator to count calls to a function
2) Defining a decorator to cache results to a function
3) Exploring standard Python decorators (if time permits)

Exercise 1: Defining a decorator to count calls to a function

In the *student* folder, take a look at `fib.py`. The `fib()` function returns the n^{th} number in the Fibonacci sequence, using recursion.

Define a decorator function named `countCalls`, which will count the number of times the decorated function is called. Hint – the decorator function must be stateful, i.e. it must preserve a counter and increment it every time the decorated function is called. The easiest way to do this is to pin a property (e.g. named `calls`) to the inner function.

Now decorate the `fib()` function with your `countCalls` decorator. Then enhance the client code to retrieve the `calls` property, to see how many calls occurred to your `fib()` function.

Exercise 2: Defining a decorator to cache results to a function

You might have noticed the number of calls to the `fib()` function is startlingly high. This is because of a lot of (unnecessary) repeated calls to fib(), to calculate results that have already been gotten previously.

You can improve performance dramatically via caching. Generally, the idea is that every time a target function is called with a particular set of arguments (and keyword arguments), stick the result in a cache. Then if the function is called again with the same arguments (and keyword arguments), don't actually call the function but just retrieve the result from cache.

Define a decorator named `cacheResults`, to implement this behavior. Here are some hints:

- The decorator must be stateful. It must hold a dictionary that maps previous function calls to the results of those calls.
- In the dictionary, the key must be the combination of args and kwargs. The best way to do this is to combine args and kwargs into a tuple as follows (make sure you understand what this means):

```
cacheKey = args + tuple(kwargs.items())
```

- Implement the decorator so that it only calls the target function if it hasn't already been called with a particular set of arguments. When you do need to call the function, make sure you cache the result.

Now decorate the `fib()` function with your `cacheResults` decorator. Note that when you chain decorators, the order in which they are stacked is bottom to top, i.e. the bottom decorator in the list is the one that is applied first. This means you must chain the decorators as follows (why is this so?):

```
@countCalls
@cacheResults
def fib(n) :
```

Run the program again and see how the number of calls has reduced.

Exercise 3 (If time permits): Exploring standard Python decorators

Explore the following standard Python decorators in the `functools` module (for more info about these decorators, see https://docs.python.org/3.8/library/functools.html):

- `@wraps`
- `@lru_cache`