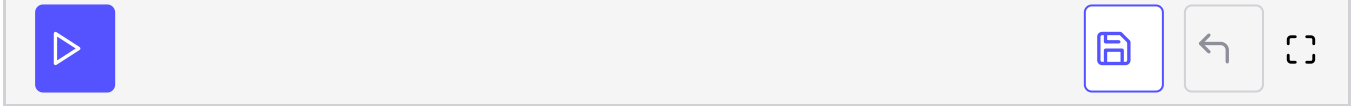
## Wraps to the Rescue!

How do we fix this little mess? The Python developers have given us the solution in functools.wraps! Let's check it out:

```
from functools import wraps


def another_function(func):
    """
    A function that accepts another function
    """

    @wraps(func)
    def wrapper():
        """
        A wrapping function
        """
        val = "The result of %s is %s" % (func(),
                                          eval(func())
                                          )

        return val
    return wrapper


@another_function
def a_function():
    """A pretty useless function"""
    return "1+1"


if __name__ == "__main__":
    #a_function()
    print(a_function.__name__)
    print(a_function.__doc__)
```



Here we import wraps from the functools module and use it as a decorator for the nested wrapper function inside of another_function. If you run it this time, the output will have changed:

```
a_function
A pretty useless function
```

```
A pretty useless function
```

Now we have the right name and docstring once more. If you go into your Python interpreter, the help function will now work correctly as well. I'll skip putting its output here and leave that for you to try.

## Wrapping Up

Let's review. In this chapter, you learned some basic caching using **lru_cache**. Then we moved onto **partials** which lets you "freeze" a portion of your function's arguments and/or keywords allowing you to create a new object that you can call. Next we used **singledispatch** to overload functions with Python. While it only allows function overloading based on the first argument, this is still a handy tool to add to your arsenal! Finally we looked at **wraps** which had a very narrow focus: namely it fixes docstrings and function names that have been decorated such that they don't have the decorator's docstring or name any more.