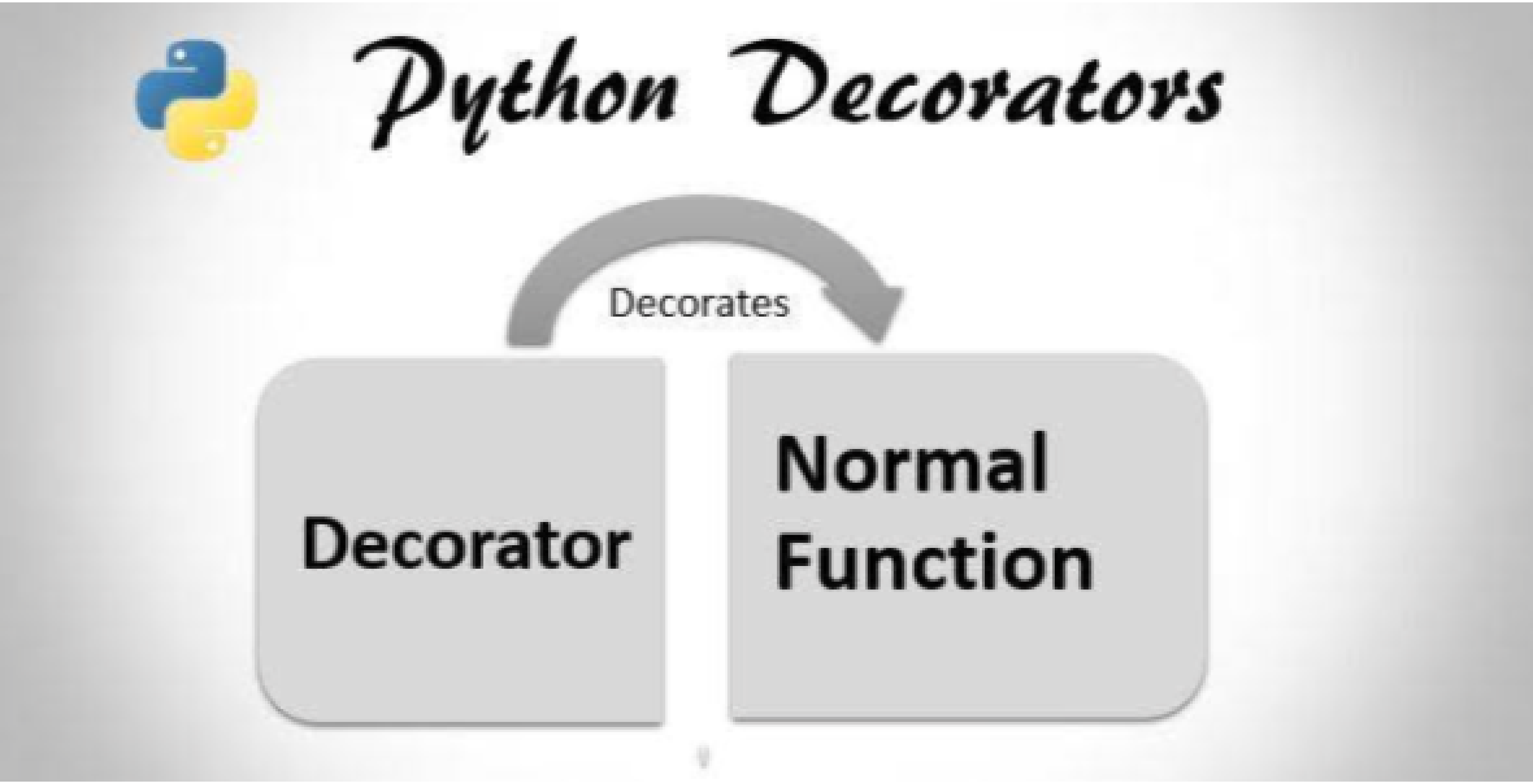
Python


Python Decorators
Decorates
Decorator
Normal Function

What is Python Decorator?

Decorator, as can be noticed by the name, is like a designer that helps to modify a function. The decorator can be said to be a modification to the external layer of function, as it does not change its structure. A decorator takes a function and inserts some new functionality in it without changing the function itself. A reference to a function is passed to a decorator, and the decorator returns a modified function. The modified functions usually contain calls to the original function. This is also known as **metaprogramming** because a part of the program tries to modify and add functionality to another part of the program at compile time. Understanding the definition could be difficult, but you can easily grasp the concept through the video section example. In terms of Python, the other function is also called a wrapper.

Why you should write Decorators?

- **Modularity of Decorators**
  Functionalities can be added or removed easily in defined blocks of code, which refrains the repetition of boilerplate setup.

- **Decorators are Explicit**
  It can be applied to all callable based on the need. This provides readability and hence valuable for debugging.

Where Decorators are used?

- **@classmethod / @staticmethod**
  Creates a method without creating an instance

- **@mock.patch / @mock.patch.object**
  Used for Unit Testing

- **@login_required**
  For setting login privileges

- **@app.route**
  For Function Registry

- **@task**
  To identify function as an asynchronous task

A **wrapper** is a function that provides a wrap-around another function. While using decorator, all the code executed before our function that we passed as a parameter and the code after it is executed belongs to the wrapper function. The purpose of the wrapper function is to assist us. Like if we are dealing with a number of similar statements, the wrapper can provide us with some code that all the functions have in common, and we can use a decorator to call our function along with the wrapper. A function can be decorated many times.

Note that a decorator is called before defining a function.

There are two ways to write a Python decorator:

- We can pass our function to the decorator as an argument, thus defining a function and passing it to our decorator.
- We can simply use the @ symbol before the function we’d like to decorate.

```python
def inner1(func):
    def inner2():
        print("Before function execution");
        func()
        print("After function execution")
    return inner2

@inner1
def function_to_be_used():
    print("This is inside the function")

function_to_be_used()
```

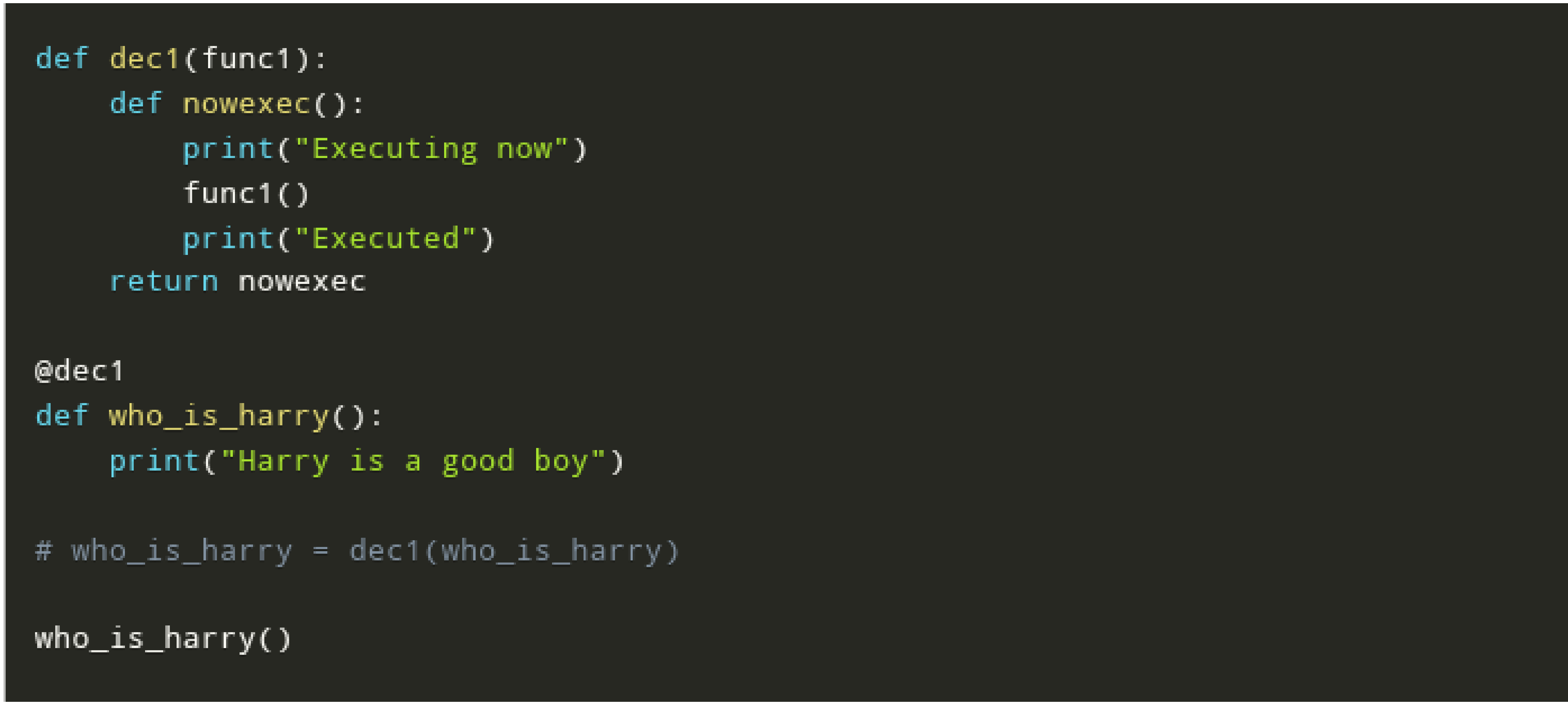
Output:

```
Before function execution
This is inside the function
After function execution
```

Advantages:

- Decorator function can make our work compact because we can pass all the functions to a decorator that requires the same sort of code that the wrapper provides.
- We can get our work done without any alteration in the original code of our function.
- We can apply multiple decorators to a single function.
- We can use decorators in authorization in Python frameworks such as Flask and Django, Logging, and measuring execution time.

We can do a lot with decorators, like Multiple decorators that can be applied to a single function. I hope this tutorial serves as a good introduction to decorators in Python. After understanding the basics of Python decorator, learn more advanced use cases of decorators and how to apply them to classes.

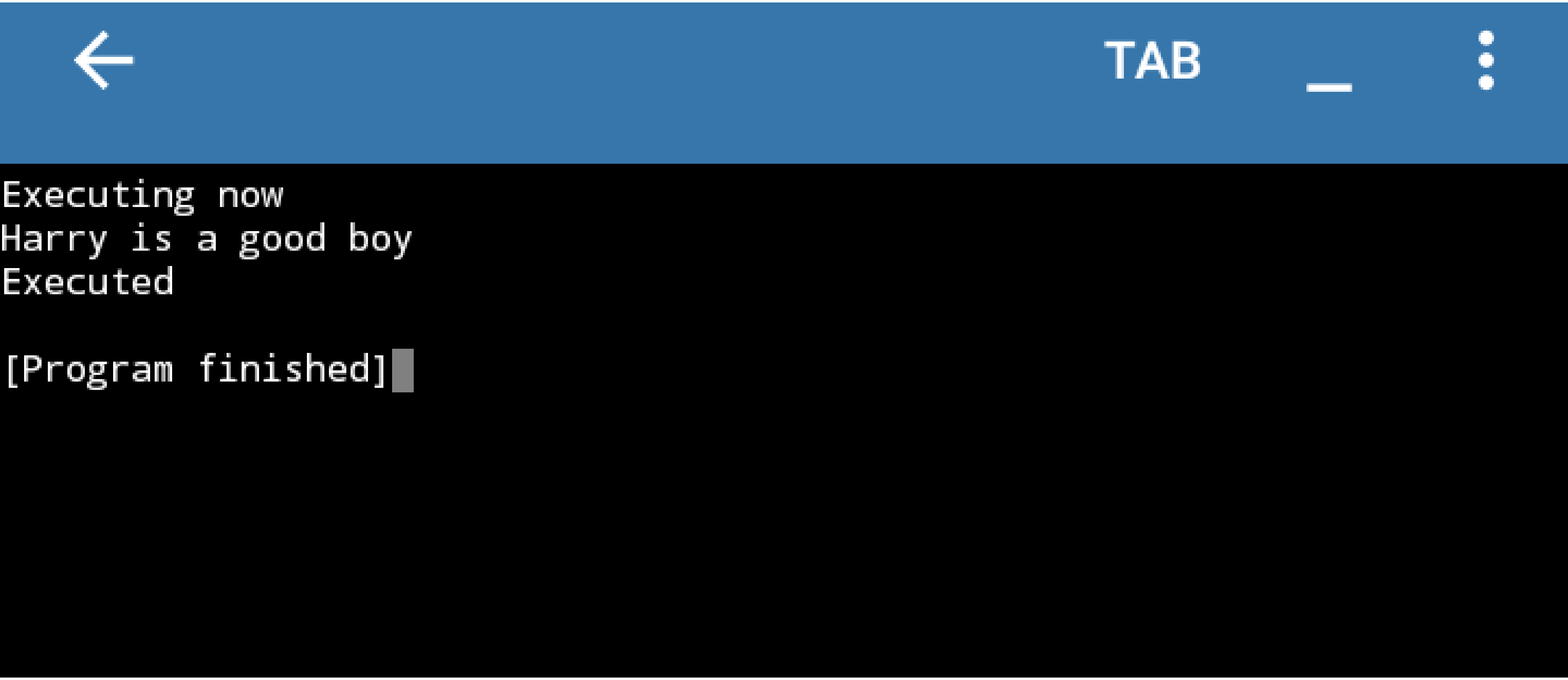


```python
def dec1(func1):
    def nowexec():
        print("Executing now")
        func1()
        print("Executed")
    return nowexec

@dec1
def who_is_harry():
    print("Harry is a good boy")

# who_is_harry = dec1(who_is_harry)

who_is_harry()
```



```
Executing now
Harry is a good boy
Executed

[Program finished]
```