Python mock with decorators

Use Case

- create a service that changes its `Class` or behaviour in runtime
- use the actual service in `PROD` but mock service in `DEV`

Goal 🚀

```
@with_mock(cls=ServiceA, mock=ServiceB)
class SomeService:
    """
    Service class that dynamically changes during runtime
    """
```

How to achieve dynamic classes at runtime ?

Maybe use an intermediary class/function that provides the right class.

```
def get_service(condition: bool, cls_a, cls_b):
  """Returns the specified classes acording to provided condition."""
  return cls_a if condition else cls_b
```

Downside of this approach

- need to create an intermidiary function/class to handle the assignment of the `cls` needed.

What I had envisioned

I wanted a method to easily extend this usecase by adding a simple decorator.

> Concise and easily added to other services

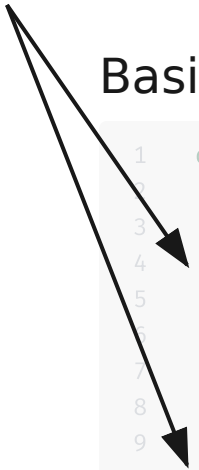
The Solution 🤔

@Decorators

Decorators are used to add additional (meta data) context to `class/functions`.

- add behaviour to classes and functions without directly changing their source code

Basic decorator



```python
def decorator_function(original_function):
    def wrapper_function(*args, **kwargs):
        # Add some extra functionality before the original function is called
        print("Before the function call")

        # Call the original function
        result = original_function(*args, **kwargs)

        # Add some extra functionality after the original function is called
        print("After the function call")

        return result

    return wrapper_function

@decorator_function
def hello():
    print("Hello, world!")

hello()
```

✡ Path to goal

- some way to generate dynamic class
- a condition to switch class

> See the implementation

```python
def changeclass(condition: bool, class_a: T, class_b: T):
    """Decorator to change class according to the condition provided"""

    def decorator(cls: T) -> T:
        new_class = class_a if condition else class_b

        cls_name = getattr(new_class, "__name__")
        # INFO:  attr and behavior of the original class
        bases = getattr(cls, "__bases__", ())

        # Create a new class dynamically with the same name and bases as the original class
        modified_class = type(cls_name, bases, dict(new_class.__dict__))

        return modified_class

    return decorator
```