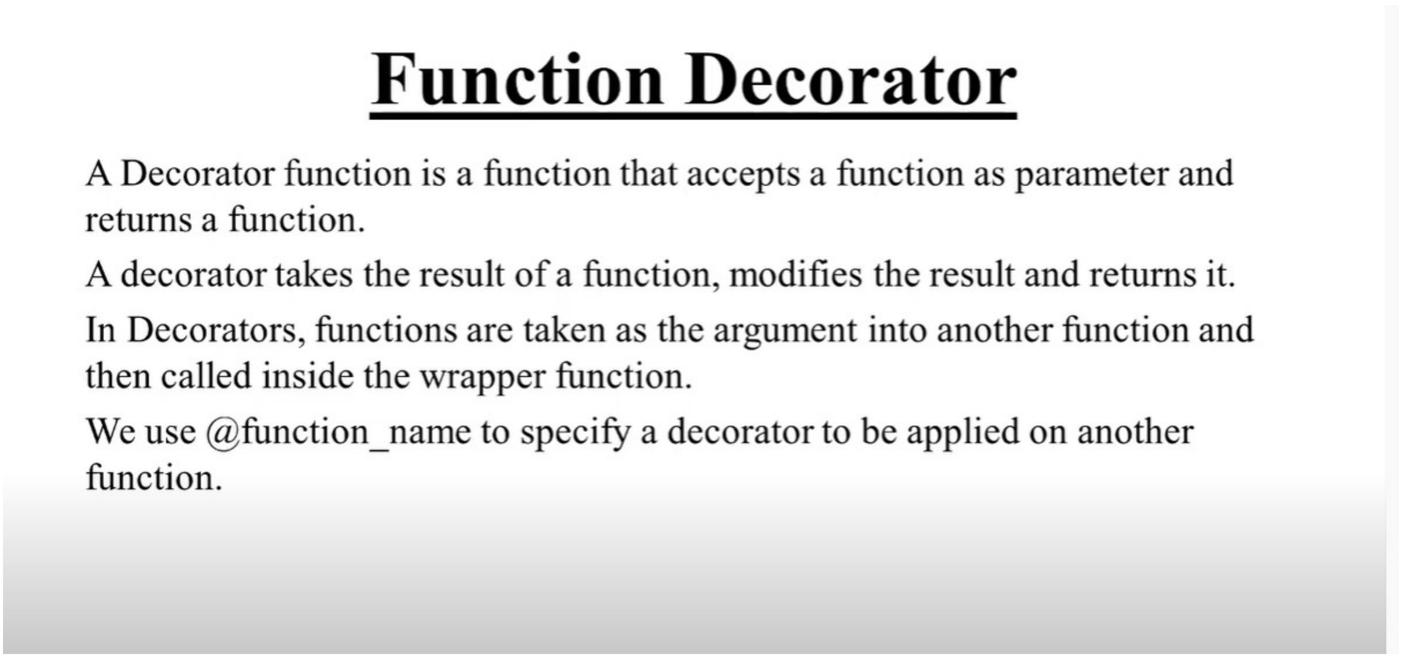
PYTHON NOTES-4

ADVANCED PYTHON DECORATORS

- Decorators enhances the functionality of other function.
- In Python, functions are the first-class objects, which means that –
  Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.
- Functions can be defined inside another function and can also be passed as argument to another function.
- Decorators are very powerful and useful tool in Python since it allows programmers to modify the behaviour of function or class. Decorators allow us to wrap another function in order to extend the behaviour of wrapped function, without permanently modifying it.
- In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.



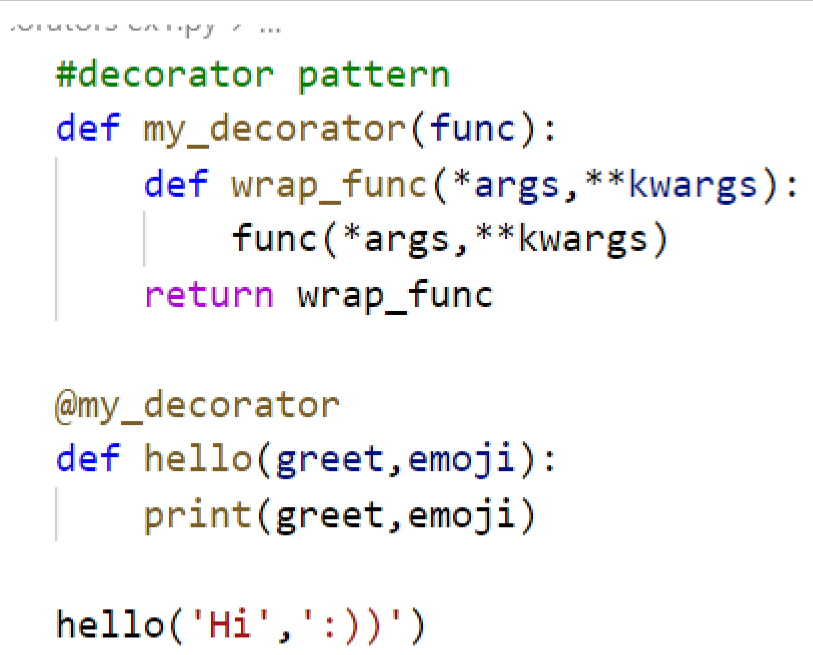
Function Decorator

A Decorator function is a function that accepts a function as parameter and returns a function.

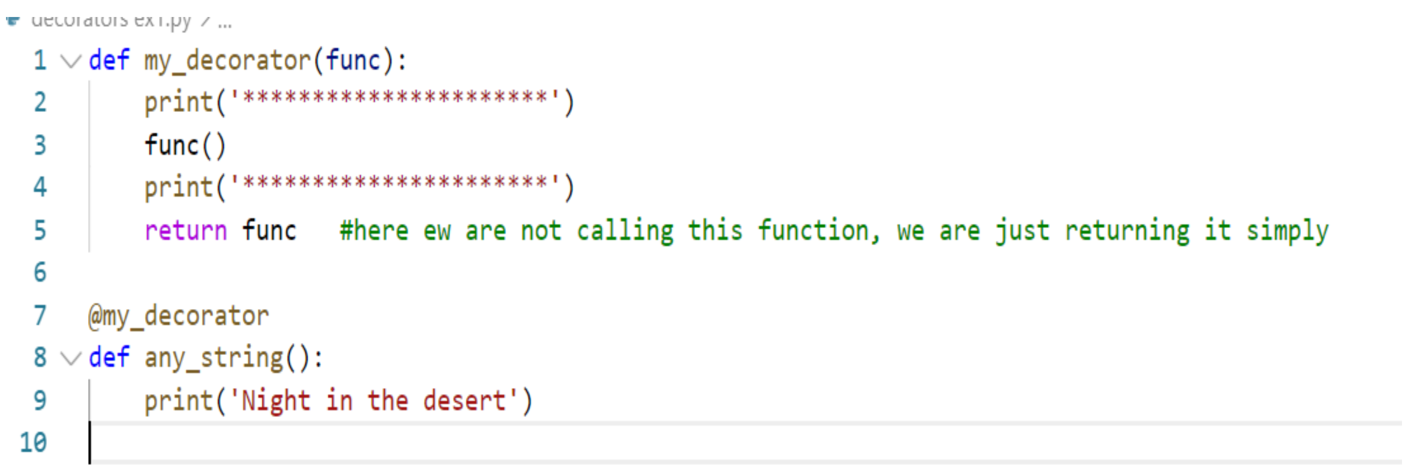
A decorator takes the result of a function, modifies the result and returns it.

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

We use @function_name to specify a decorator to be applied on another function.



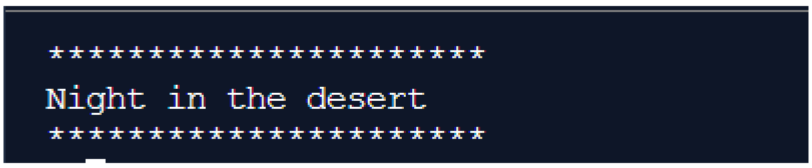
```
#decorator pattern
def my_decorator(func):
    def wrap_func(*args,**kwargs):
        func(*args,**kwargs)
    return wrap_func

@my_decorator
def hello(greet,emoji):
    print(greet,emoji)

hello('Hi',':))')
```



decorators ex1.py > ...

```python
def my_decorator(func):
    print('**********************')
    func()
    print('**********************')
    return func   #here ew are not calling this function, we are just returning it simply

@my_decorator
def any_string():
    print('Night in the desert')

```

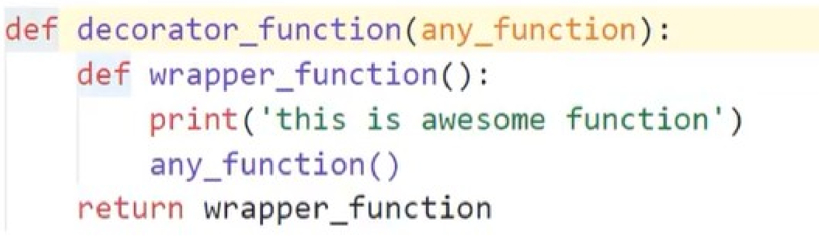
OUTPUT:



```
**********************
Night in the desert
**********************
```

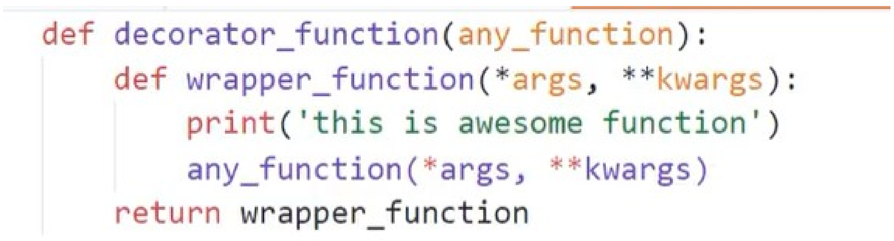
Note: structure of any decorator used code:

# if only one argument is given



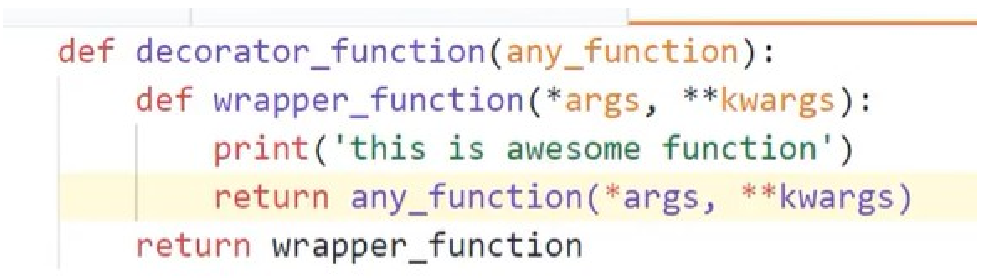
```python
def decorator_function(any_function):
    def wrapper_function():
        print('this is awesome function')
        any_function()
    return wrapper_function
```

#If more than one argument is given in the function



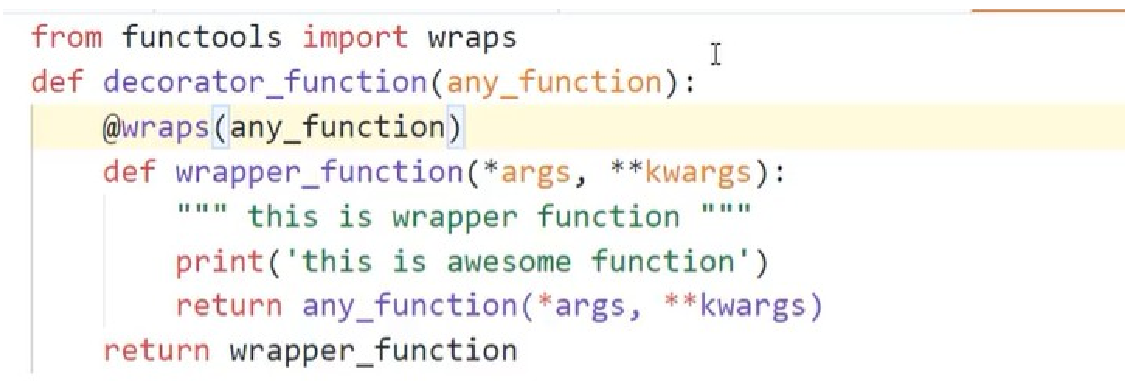
```python
def decorator_function(any_function):
    def wrapper_function(*args, **kwargs):
        print('this is awesome function')
        any_function(*args, **kwargs)
    return wrapper_function
```

#if we want that any_function given by us return some value then we have to give the return statement



```python
def decorator_function(any_function):
    def wrapper_function(*args, **kwargs):
        print('this is awesome function')
        return any_function(*args, **kwargs)
    return wrapper_function
```

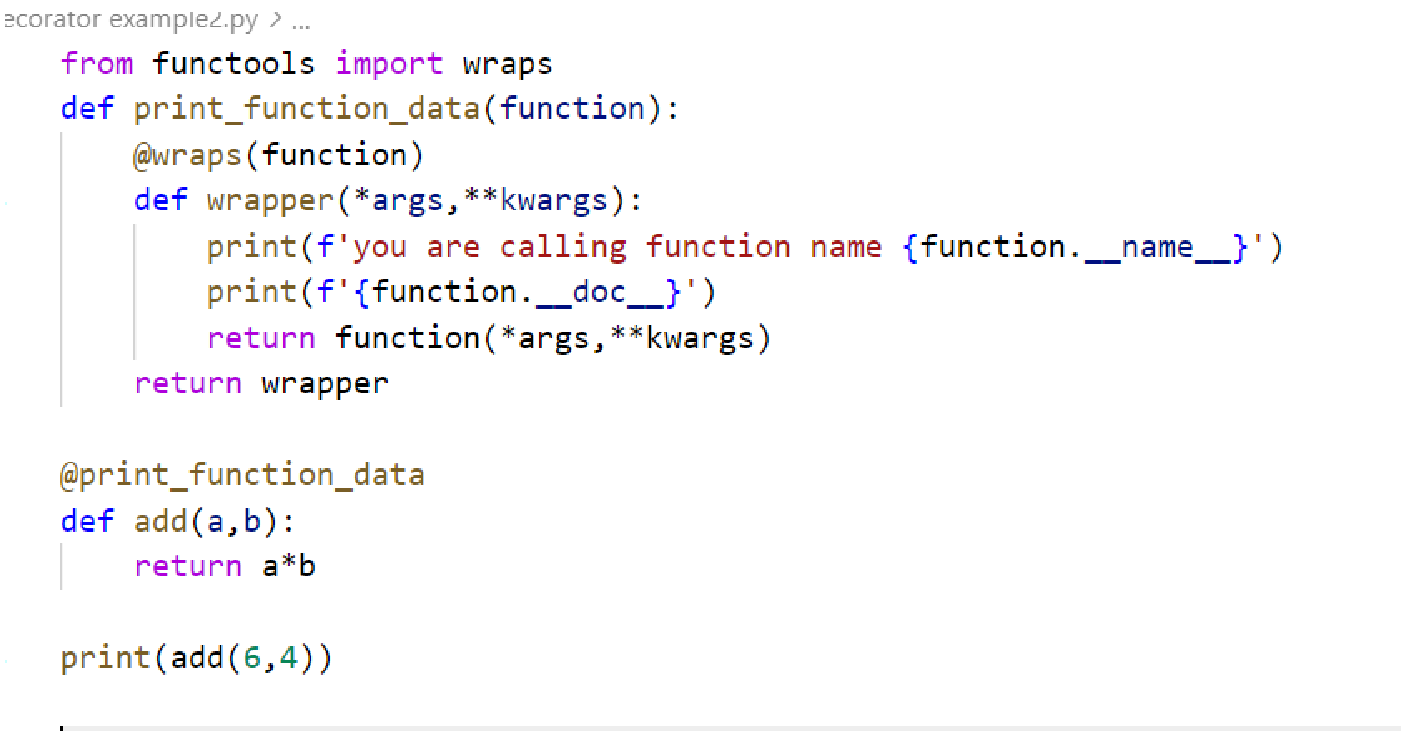
#remedy for doc string issue-



```
from functools import wraps
def decorator_function(any_function):
    @wraps(any_function)
    def wrapper_function(*args, **kwargs):
        """ this is wrapper function """
        print('this is awesome function')
        return any_function(*args, **kwargs)
    return wrapper_function
```

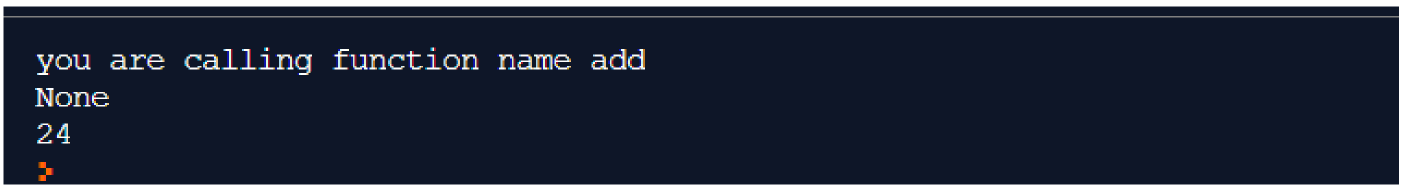
Code Exercise:

-import wraps module to solve the issue for doc string
- define a decorator function name as print_function_data
- def wrapper function which will print certain lines
- then define function named as add and give arguments into it and use decorator function.



ecorator example2.py > ...

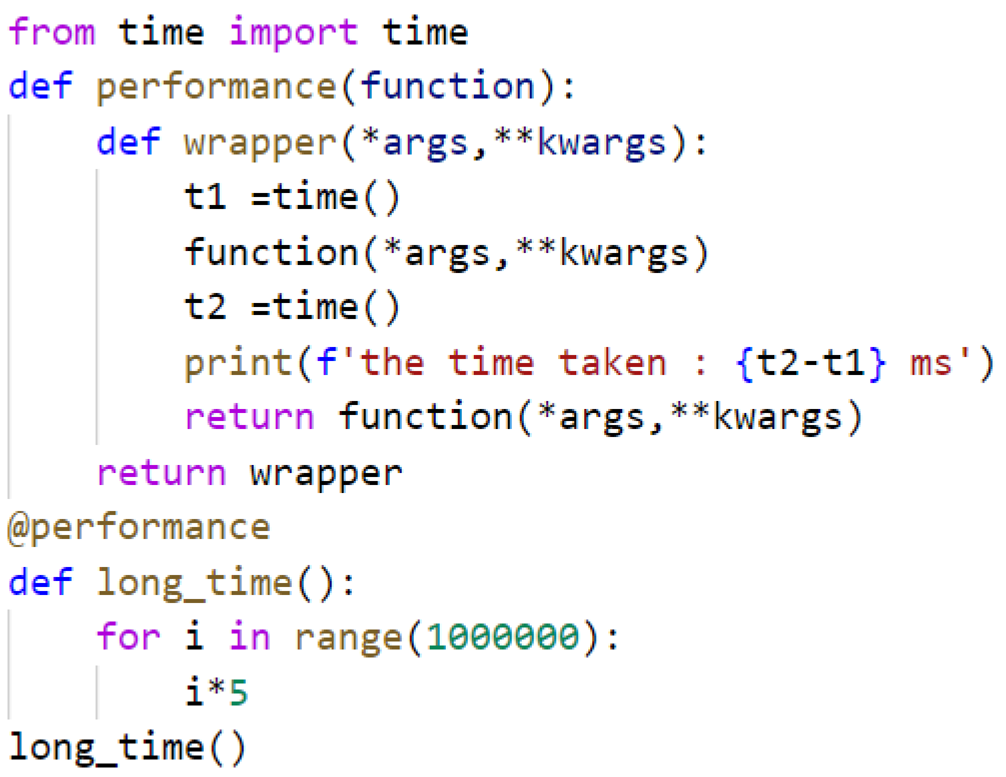
```
from functools import wraps
def print_function_data(function):
    @wraps(function)
    def wrapper(*args,**kwargs):
        print(f'you are calling function name {function.__name__}')
        print(f'{function.__doc__}')
        return function(*args,**kwargs)
    return wrapper

@print_function_data
def add(a,b):
    return a*b

print(add(6,4))
```



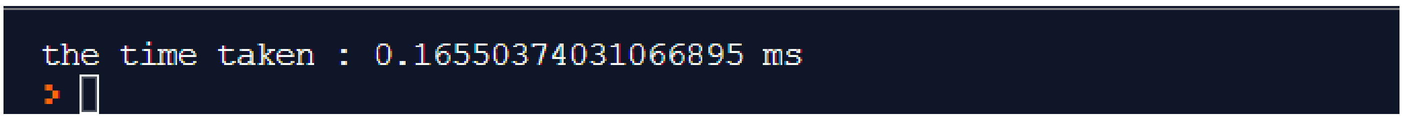
```
you are calling function name add
None
24
```

Code Exercise: define a decorator performance which calculate the initial and final time taken by any function to run and give result.

- We need a module ‘time’ for this decorator to work.



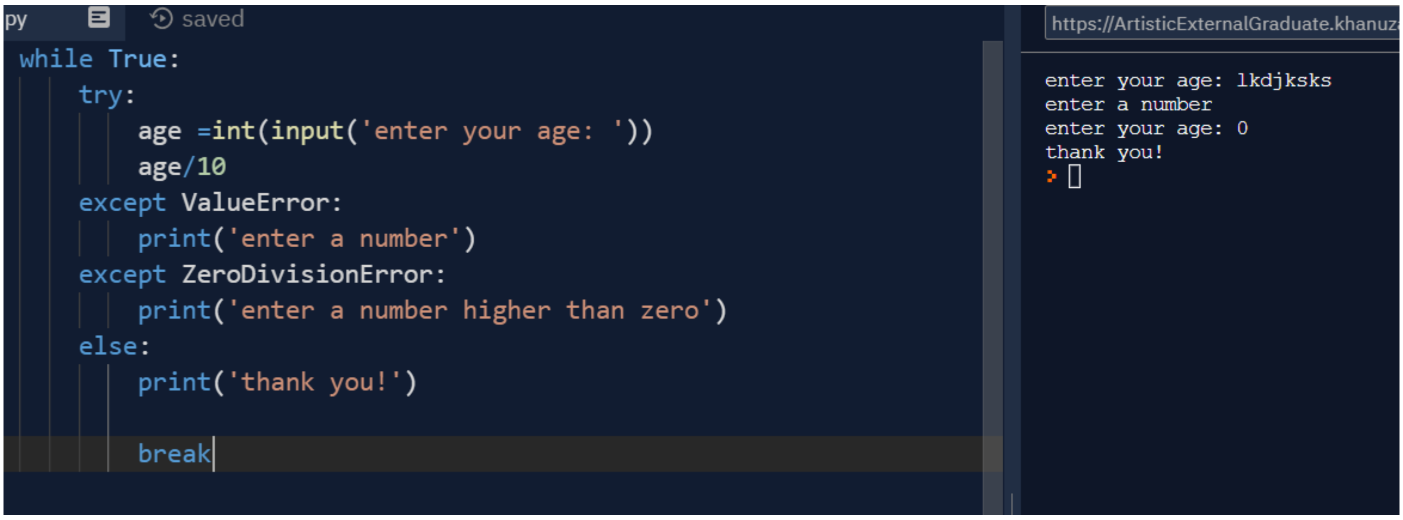
```python
from time import time
def performance(function):
    def wrapper(*args,**kwargs):
        t1 =time()
        function(*args,**kwargs)
        t2 =time()
        print(f'the time taken : {t2-t1} ms')
        return function(*args,**kwargs)
    return wrapper
@performance
def long_time():
    for i in range(1000000):
        i*5
long_time()
```



```
the time taken : 0.16550374031066895 ms
>
```

ERROR HANDLING

https://docs.python.org/3/library/exceptions.html



```
while True:
    try:
        age =int(input('enter your age: '))
        age/10
    except ValueError:
        print('enter a number')
    except ZeroDivisionError:
        print('enter a number higher than zero')
    else:
        print('thank you!')

        break
```

```
enter your age: lkdjksks
enter a number
enter your age: 0
thank you!
>
```

- The `try` block lets you test a block of code for errors.
- The `except` block lets you handle the error.
- The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling

- When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

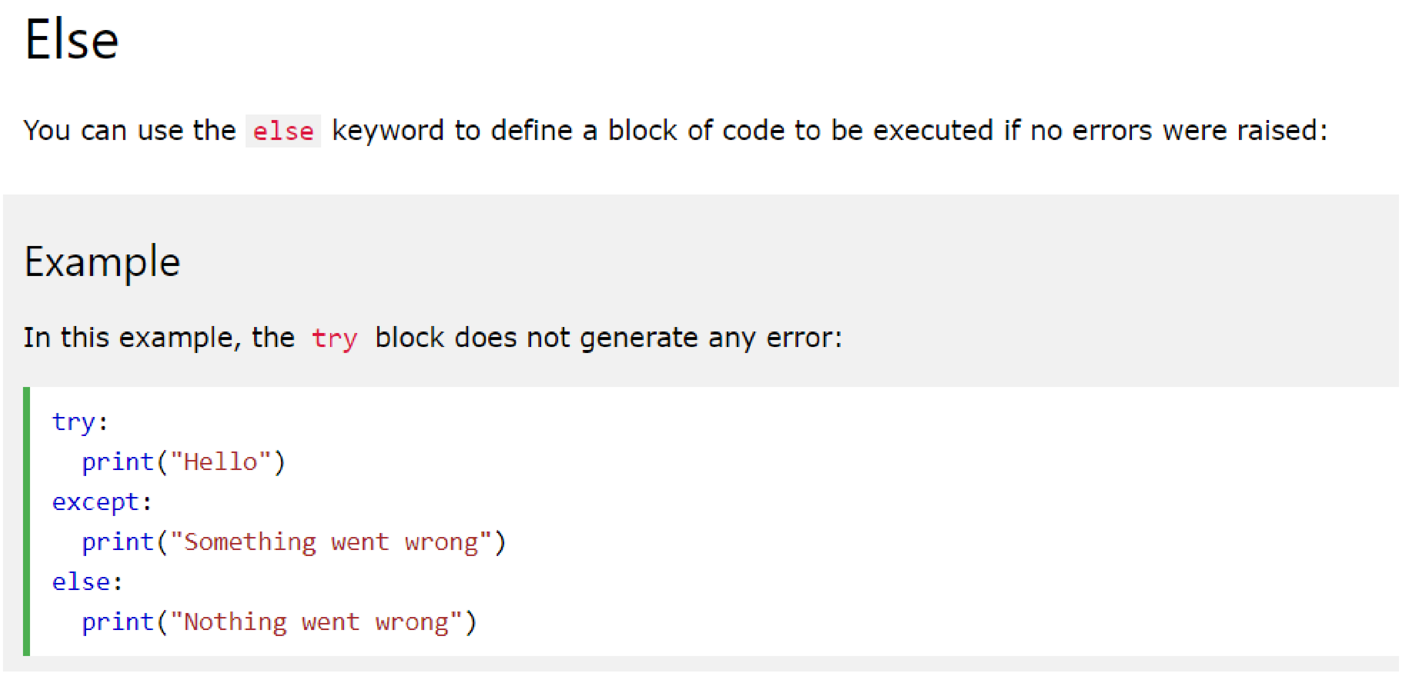
These exceptions can be handled using the **`try`** statement:

The `try` block will generate an exception, because `x` is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Print one message if the try block raises a `NameError` and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```



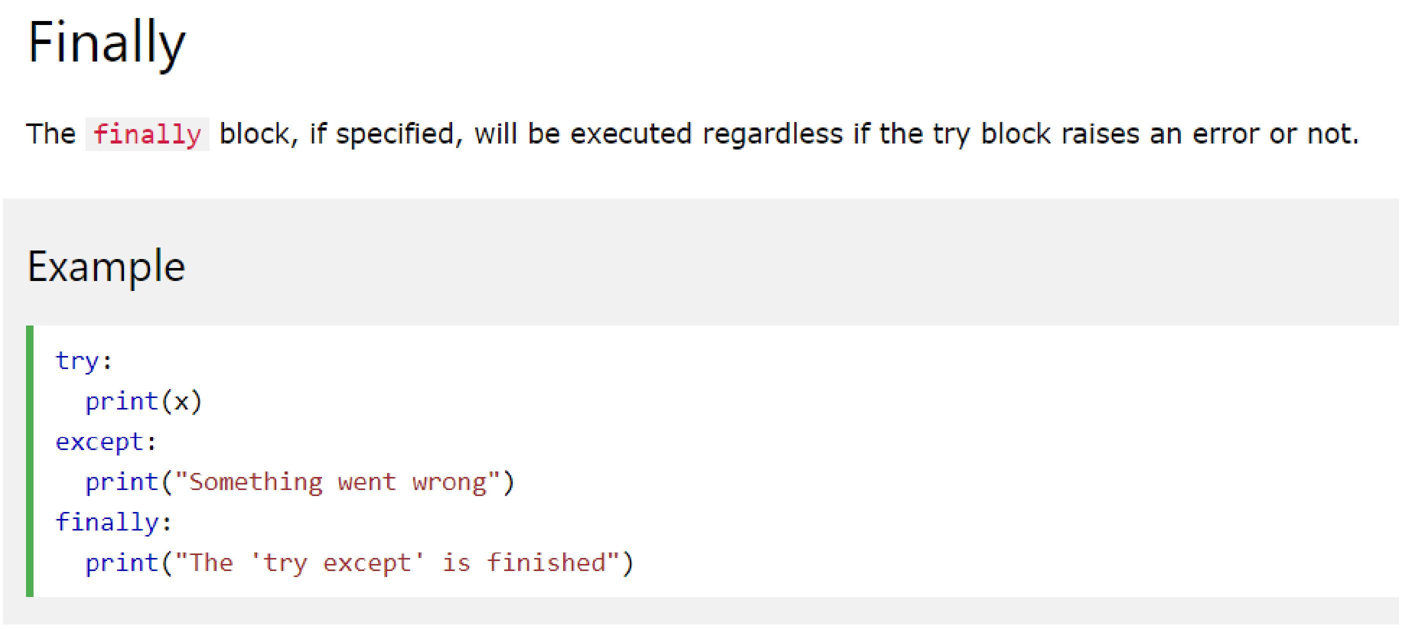
Else

You can use the `else` keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the `try` block does not generate any error:

```python
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```



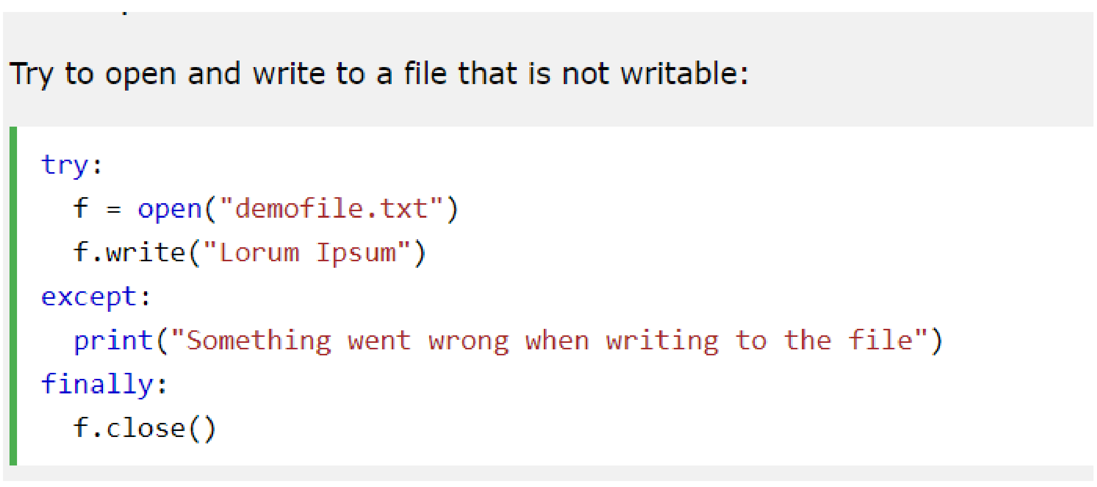
Finally

The `finally` block, if specified, will be executed regardless if the try block raises an error or not.

Example

```python
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

- This keyword 'FINALLY' can be used to close the object and clean-up the resource.



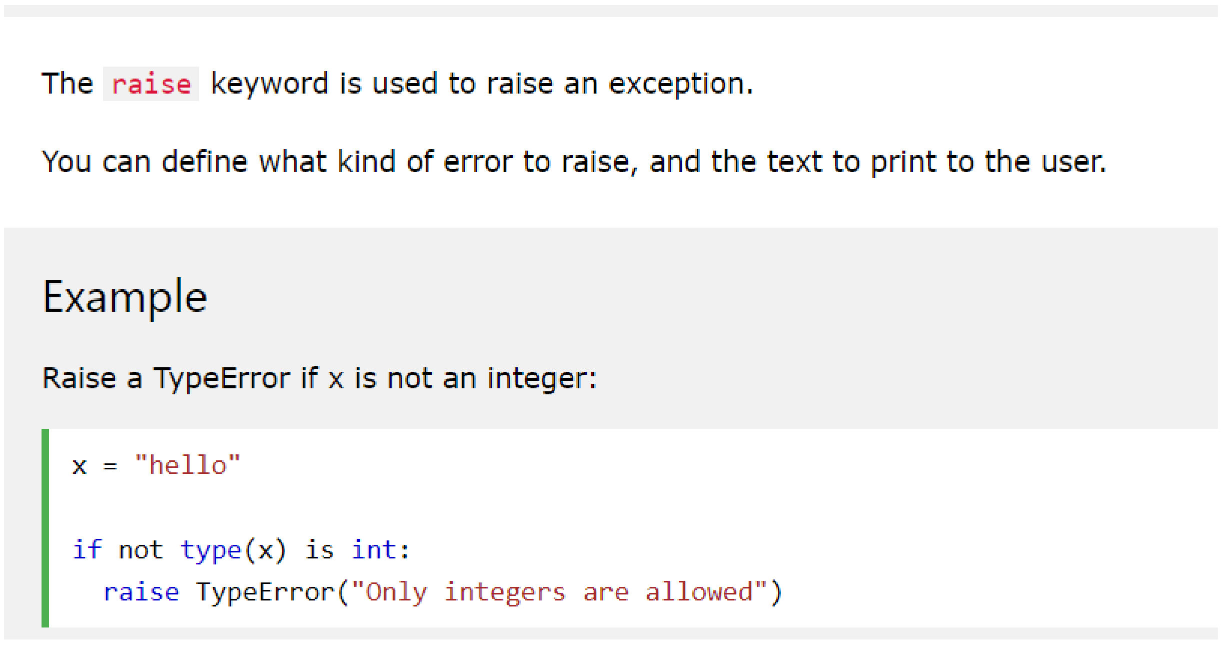
Try to open and write to a file that is not writable:

```python
try:
  f = open("demofile.txt")
  f.write("Lorum Ipsum")
except:
  print("Something went wrong when writing to the file")
finally:
  f.close()
```

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the `raise` keyword.



The `raise` keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

Example

Raise a TypeError if x is not an integer:

```python
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```