19. Nested functions: encapsulation, closures, factory functions

Let us look at **3 common reasons for writing inner (nested) functions**:

1 Encapsulation

You use inner functions to protect them from everything happening outside of the function, meaning that they are **hidden from the global scope**.

Illustartive example:

```
def factorial(number):
    # Error handling:
    if not isinstance(number, int):  # type check
       raise TypeError("Sorry. 'number' must be an integer.")
    if not number >= 0:
       raise ValueError("Sorry. 'number' must be zero or positive.")

    def inner_factorial(number):
        if number <= 1:
           return 1
        return number*inner_factorial(number-1)

    return inner_factorial(number)

# Call the outer function:
print(factorial(4))
> 24
```

An advantage of using this design pattern is that by performing all argument checking in the outer function, **you can safely skip error checking altogether in the inner function**.

2 Keeping it DRY

DRY: Do Not Repeat Yourself.

Perhaps you have **a giant function that performs the same chunk of code in numerous places**. E.g. you might write a function that processes a file, and you want to accept either an open file object or a file name:

```
def process(file_name):
    def do_stuff(file_process):
        for line in file_process:
            print(line)

    if isinstance(file_name, str):
        with open(file_name, 'r') as f:
            do_stuff(f)
    else:
        do_stuff(file_name)
```

More practical example: A number of Wi-Fi hotspots in New York City from a CSV:

```
import itertools

def process(file_name):

    def do_stuff(file_process):
        wifi_locations = {}
        for line in file_process:
            values = line.split(',')
            # Build the dict and increment values:
            try:
                wifi_locations[values[4]] = wifi_locations.get(values[4], 0) + 1
            except IndexError as name:
                print(name, values)
         max_key = 0
         for name, key in wifi_locations.items():
             all_locations = sum(wifi_locations.values())
             if key > max_key:
                 max_key = key
                 business = name
         print(f'There are {all_locations} Wi-Fi hotspots in NYC, '
               f'and {business} has the most with {max_key}.')

     if isinstance(file_name, str):
         with open(file_name, 'r') as f:
             do_stuff(f)
     else:
         do_stuff(file_name)
```

```
process('NYC.csv')
> There are 3348 Wi-Fi hotspots in NYC, and  has the most with 257.
```

3 Closures and factory functions

Now we come to the most important reason of using inner functions. All considered above were ordinary functions nested inside another one.

3.1 What is a closure

A closure **causes the inner function to remember the state of its environment when called**. The closure “closes” the local variable on the stack, and this stays after the stack creation has finished:

Example: ***factory function***

```
def generate_power(power):  # 'power' is now closed\pythoninline{}
    '''Factory function'''
    def nth_power(number):
        return number ** power  # ... that is returned by the factory function.
    return nth_power # returning a function

raise_two   = generate_power(2)  # create one function
raise_three = generate_power(3)  # create another function

raise_two(2) -> 4
raise_three(2) -> 8
```

4 Conclusion

The use of closures and factory functions is the most common and powerful use for inner functions.

In most cases, when you see a decorated function, the decorator is a factory function that takes a function as argument and returns a new function that includes the old function inside the closure. To put it another way, a decorator is just syntactic sugar for implementing the process outlined in the `generate_power()` example.

```python
def generate_power(exponent):
    def decorator(f):
        def inner(*args):
            result = f(*args)
            return exponent**result
        return inner
    return decorator

@generate_power(2)
def raise_two(n):
    return n

@generate_power(3)
def raise_three(n):
    return n

raise_two(7) -> 128
raise_two(5) -> 32
```