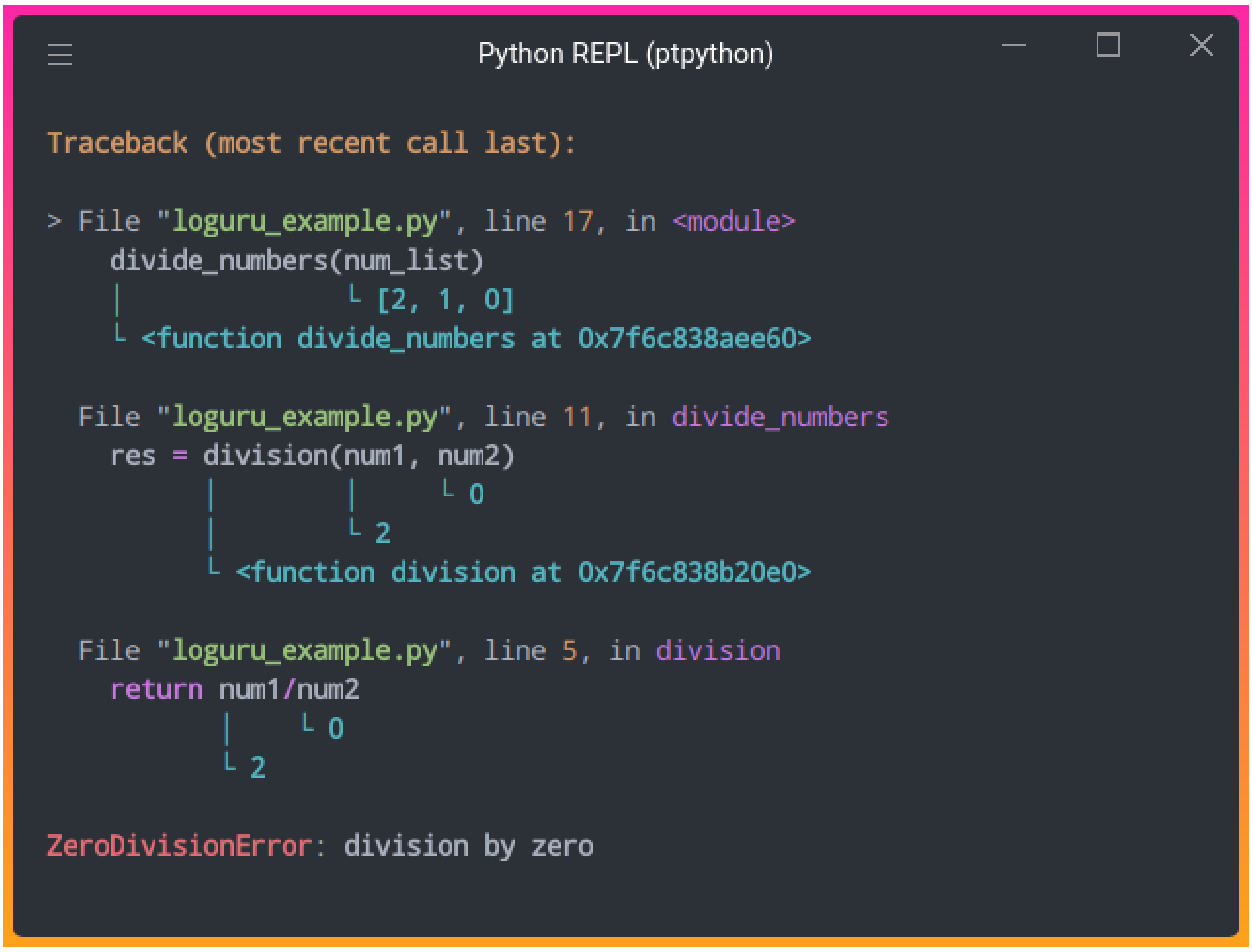
3 Tools to Track and Visualize the Execution of your Python Code - By Khuyen Tran



```
Traceback (most recent call last):

> File "loguru_example.py", line 17, in <module>
    divide_numbers(num_list)
    |              └ [2, 1, 0]
    └ <function divide_numbers at 0x7f6c838aee60>

  File "loguru_example.py", line 11, in divide_numbers
    res = division(num1, num2)
          |        |     └ 0
          |        └ 2
          └ <function division at 0x7f6c838b20e0>

  File "loguru_example.py", line 5, in division
    return num1/num2
           |    └ 0
           └ 2

ZeroDivisionError: division by zero
```

Loguru — Print Better Exceptions

Loguru is a library that aims to make logging in Python enjoyable. Loguru provides many interesting functionalities, but one functionality that I found to be the most helpful is the ability to **catch unexpected errors** and **display which value of a variable causes your code to fail**.

To install Loguru, type

```
pip install loguru
```

To understand how Loguru can be useful, imagine that you have 2 functions `division` and `divide_numbers`and the function `divide_numbers` is executed.

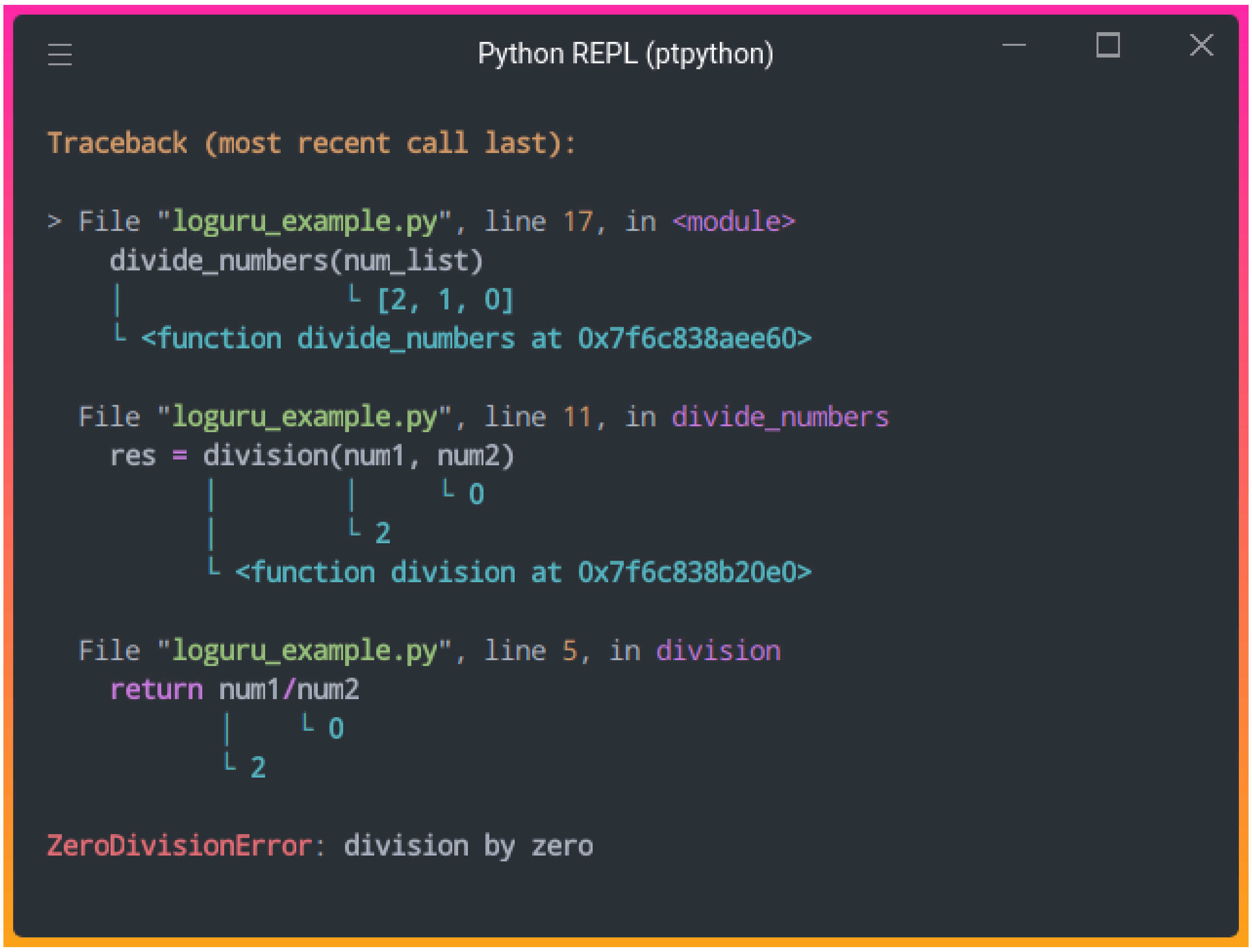
Note that `combinations([2,1,0], 2)` returns `[(2, 1), (2, 0), (1, 0)]` . After running the code above, we get this error:

```
2 divided by 1 is equal to 2.0.
Traceback (most recent call last):
  File "loguru\_example.py", line 17, in
<module>
    divide\_numbers(num\_list)
  File "loguru\_example.py", line 11, in
divide\_numbers
    res = division(num1, num2)
  File "loguru\_example.py", line 5, in
division
    return num1/num2
ZeroDivisionError: division by zero
```

From the output, we know that the line `return num1/num2` is where the error occurs, but we don't know which values of `num1` and `num2` cause the error. Luckily, this can be easily tracked by adding Loguru's `logger.catch` decorator:

```python
from loguru import logger
from itertools import combinations

def division(num1: int, num2: int):
    return num1/num2

@logger.catch # Add this to track errors
def divide_numbers(num_list: list):
    for comb in combinations(num_list, 2):
        num1, num2 = comb
        res = division(num1, num2)
        print(f"{num1} divided by {num2} is
equal to {res}.")


if __name__ =='__main__':
    num_list = [2, 1, 0]
    divide_numbers(num_list)
```

Output:



```
Traceback (most recent call last):

> File "loguru_example.py", line 17, in <module>
    divide_numbers(num_list)
    |              └ [2, 1, 0]
    └ <function divide_numbers at 0x7f6c838aee60>

  File "loguru_example.py", line 11, in divide_numbers
    res = division(num1, num2)
          |        |     └ 0
          |        └ 2
          └ <function division at 0x7f6c838b20e0>

  File "loguru_example.py", line 5, in division
    return num1/num2
           |    └ 0
           └ 2

ZeroDivisionError: division by zero
```

By adding `logger.catch`, the exceptions are much easier to understand! It turns out that the error occurs when dividing `2` by `0`.

snoop — Print the Lines of Code being Executed in a Function

What if there is no error in the code, but we want to figure out what is going on in the code? That is when snoop comes in handy.

snoop is a Python package that prints the lines of code being executed along with the values of each variable by adding only one decorator.

To install snoop, type:

```
pip install snoop
```

Let's imagine we have a function called`factorial` that finds the factorial of an integer.

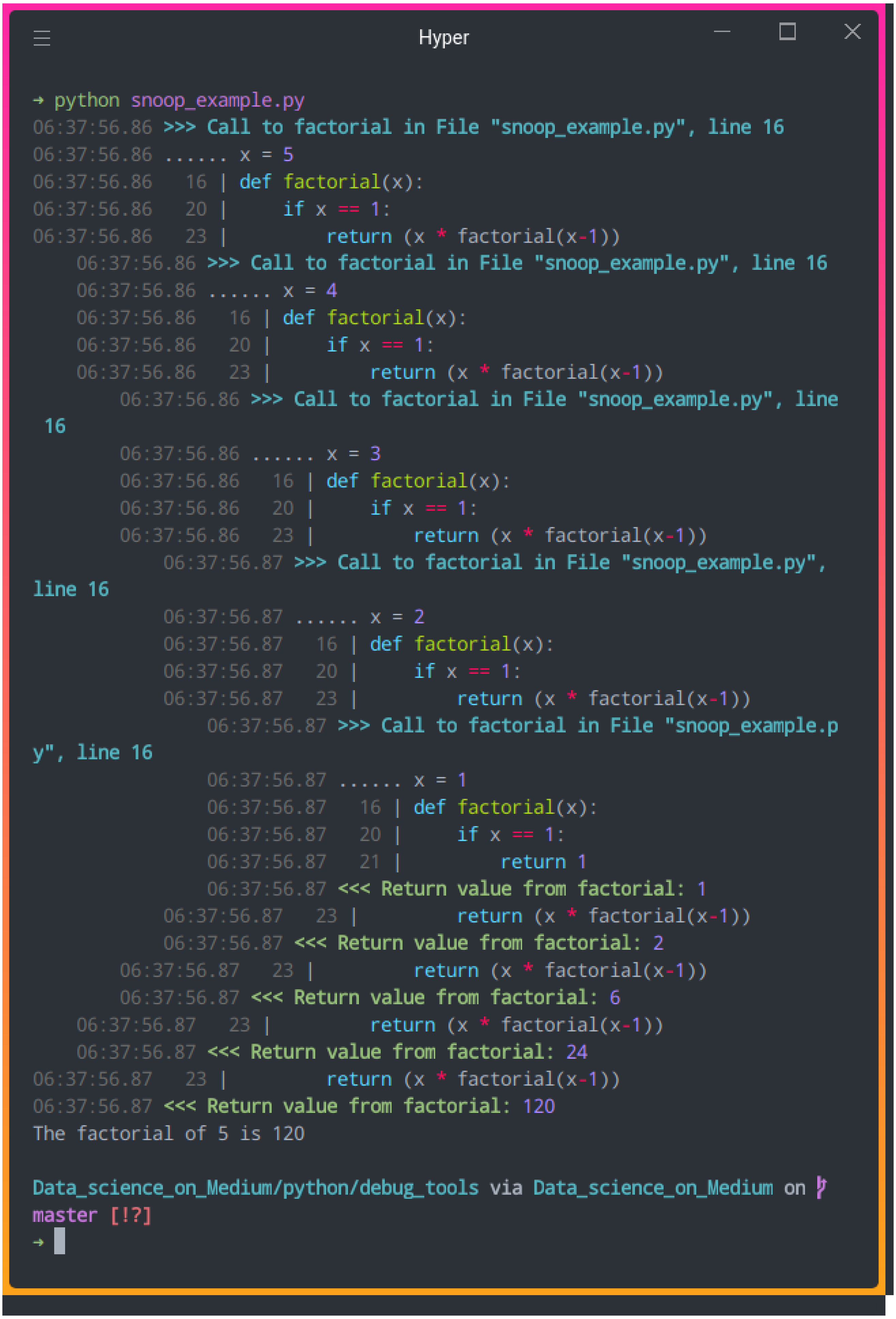
```python
import snoop

def factorial(x: int):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))

if __name__ == "__main__":
    num = 5
    print(f"The factorial of {num} is 
{factorial(num)}")
```

Output:

```
The factorial of 5 is 120
```

To understand why the output of `factorial(5)` is `20` , we can add `snoop` decorator to the function `factorial` .

Output:



```
→ python snoop_example.py
06:37:56.86 >>> Call to factorial in File "snoop_example.py", line 16
06:37:56.86 ...... x = 5
06:37:56.86   16 | def factorial(x):
06:37:56.86   20 |     if x == 1:
06:37:56.86   23 |         return (x * factorial(x-1))
    06:37:56.86 >>> Call to factorial in File "snoop_example.py", line 16
    06:37:56.86 ...... x = 4
    06:37:56.86   16 | def factorial(x):
    06:37:56.86   20 |     if x == 1:
    06:37:56.86   23 |         return (x * factorial(x-1))
        06:37:56.86 >>> Call to factorial in File "snoop_example.py", line
 16
        06:37:56.86 ...... x = 3
        06:37:56.86   16 | def factorial(x):
        06:37:56.86   20 |     if x == 1:
        06:37:56.86   23 |         return (x * factorial(x-1))
            06:37:56.87 >>> Call to factorial in File "snoop_example.py",
line 16
            06:37:56.87 ...... x = 2
            06:37:56.87   16 | def factorial(x):
            06:37:56.87   20 |     if x == 1:
            06:37:56.87   23 |         return (x * factorial(x-1))
                06:37:56.87 >>> Call to factorial in File "snoop_example.p
y", line 16
                06:37:56.87 ...... x = 1
                06:37:56.87   16 | def factorial(x):
                06:37:56.87   20 |     if x == 1:
                06:37:56.87   21 |         return 1
                06:37:56.87 <<< Return value from factorial: 1
            06:37:56.87   23 |         return (x * factorial(x-1))
            06:37:56.87 <<< Return value from factorial: 2
        06:37:56.87   23 |         return (x * factorial(x-1))
        06:37:56.87 <<< Return value from factorial: 6
    06:37:56.87   23 |         return (x * factorial(x-1))
    06:37:56.87 <<< Return value from factorial: 24
06:37:56.87   23 |         return (x * factorial(x-1))
06:37:56.87 <<< Return value from factorial: 120
The factorial of 5 is 120

Data_science_on_Medium/python/debug_tools via Data_science_on_Medium on 
master [!?]
→
```

In the output above, we can view the values of the variables and which lines of code are executed. Now we can understand how recursion works much better!

heartrate — Visualize the Execution of a Python Program in Real-Time

If you want to visualize which lines are executed and how many times they are executed, try heartrate.

heartrate is also created by the creator of snoop. To install heartrate, type:

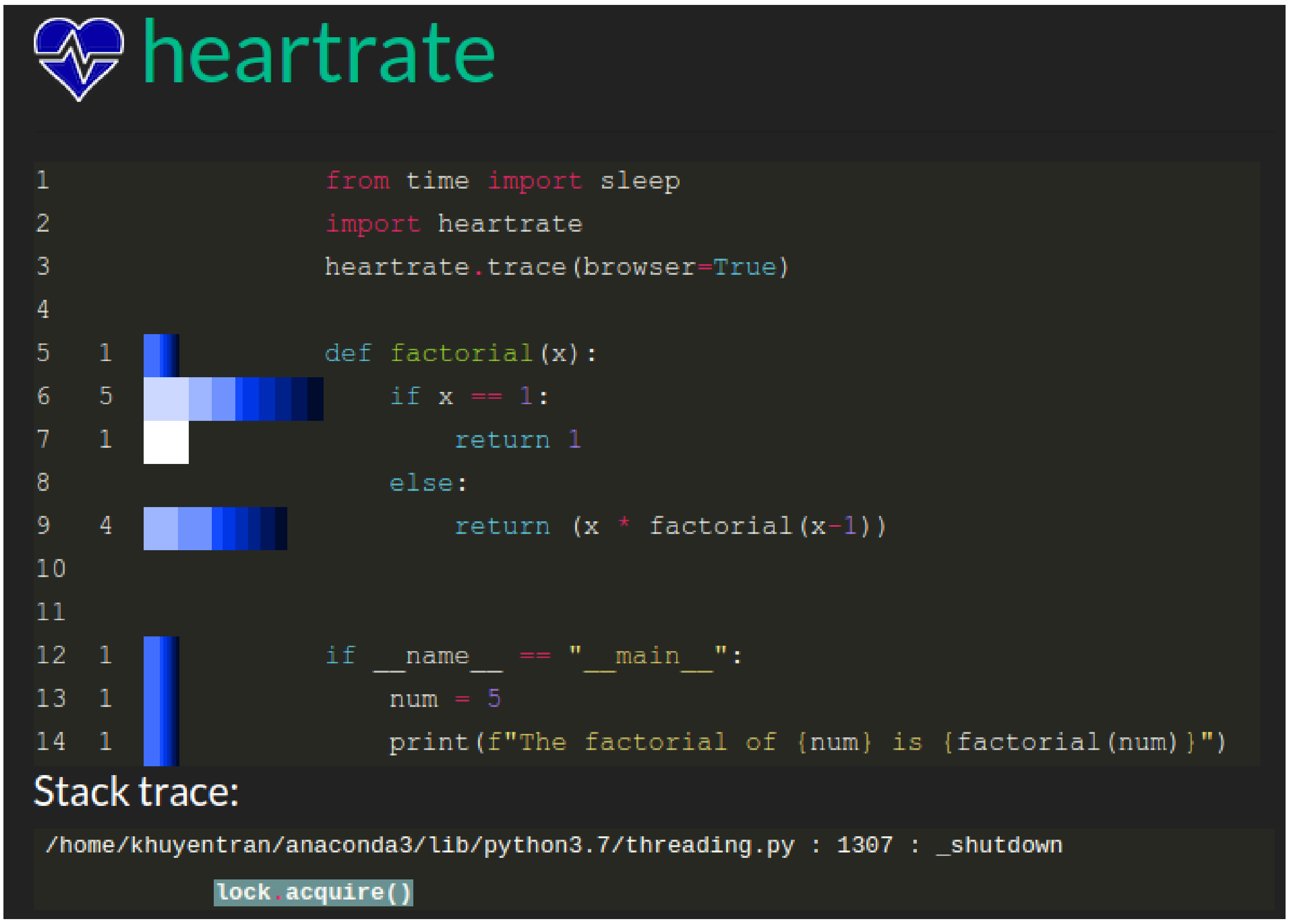
```
pip install heartrate
```

Now let's add `heartrate.trace(browser=True)` to our previous code. This will open a browser window displaying the visualization of the file where `trace()` was called.

```
import heartrate
heartrate.trace(browser=True)

def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
```

```
if __name__ == "__main__":
    num = 5
    print(f"The factorial of {num} is
{factorial(num)}")
```

A new browser should pop up when you run the code above. If not, go to http://localhost:9999. You should see the output like below:



```
1               from time import sleep
2               import heartrate
3               heartrate.trace(browser=True)
4
5   1           def factorial(x):
6   5               if x == 1:
7   1                   return 1
8                   else:
9   4                   return (x * factorial(x-1))
10
11
12  1           if __name__ == "__main__":
13  1               num = 5
14  1               print(f"The factorial of {num} is {factorial(num)}")
```

Stack trace:

```
/home/khuyentran/anaconda3/lib/python3.7/threading.py : 1307 : _shutdown
        lock.acquire()
```

Cool! The bars show the lines that have been hit. The longer bars mean more hits, lighter colors mean more recent.

From the output above, we can see that the program executes:

- `if x==1` 5 times
- `return 1` once
- `return (x * factorial(x-1))` 4 times

The output makes sense since the initial value of `x` is 5 and the function is called repetitively until `x` equals to `1` .