COL1000
Introduction to Programming

Priyanka Golia

Most (if not all) of the content is borrowed from Prof. Subodh Kumar's slides

Quiz

```
x = "global"
def outer():
    x = "outer"
    def inner():
        global x
        x = "inner"
    inner()
    print(x)
outer()
print(x)
```

```
def f():
    a = 10
    def g():
        print(a)
    a = 20
    g()
f()
```

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)

inner()
```

Quiz

```
x = "global"
def outer():
    x = "outer"
    def inner():
        global x
        x = "inner"
    inner()
    print(x)
outer()
print(x)
```

outer
inner

```
def f():
    a = 10
    def g():
        print(a)
    a = 20
    g()
f()
```

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)

inner()
```

Quiz

```
x = "global"
def outer():
    x = "outer"
    def inner():
        global x
        x = "inner"
    inner()
    print(x)
outer()
print(x)
```

outer
inner

```
def f():
    a = 10
    def g():
        print(a)
    a = 20
    g()
f()
```

20

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)

inner()
```

Quiz

```
x = "global"
def outer():
    x = "outer"
    def inner():
        global x
        x = "inner"
    inner()
    print(x)
outer()
print(x)
```

outer
inner

```
def f():
    a = 10
    def g():
        print(a)
    a = 20
    g()
f()
```

20

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)

inner()
```

NameError: name 'inner' is not defined

Because inner is **local to outer** — it’s not visible at the global level.

Recursion!

What happens when a function calls itself?

Recursion means that a function calls **itself** as part of its execution.

Every recursive function needs two parts:

Base Case – the situation where the function stops calling itself.

Recursive Case – where the function calls itself on a smaller/simpler input.

Without a base case, recursion will continue forever and cause an error.

Recursion!

```
def fact(n):
    print(f"inside function, n: {n}")
    if n == 1:
        return 1
    return n * fact(n-1)

n = 3
print(f"factorial of {n} is {fact(3)}")
```

Base case

Recursive case

Each function call waits for the next one to finish.

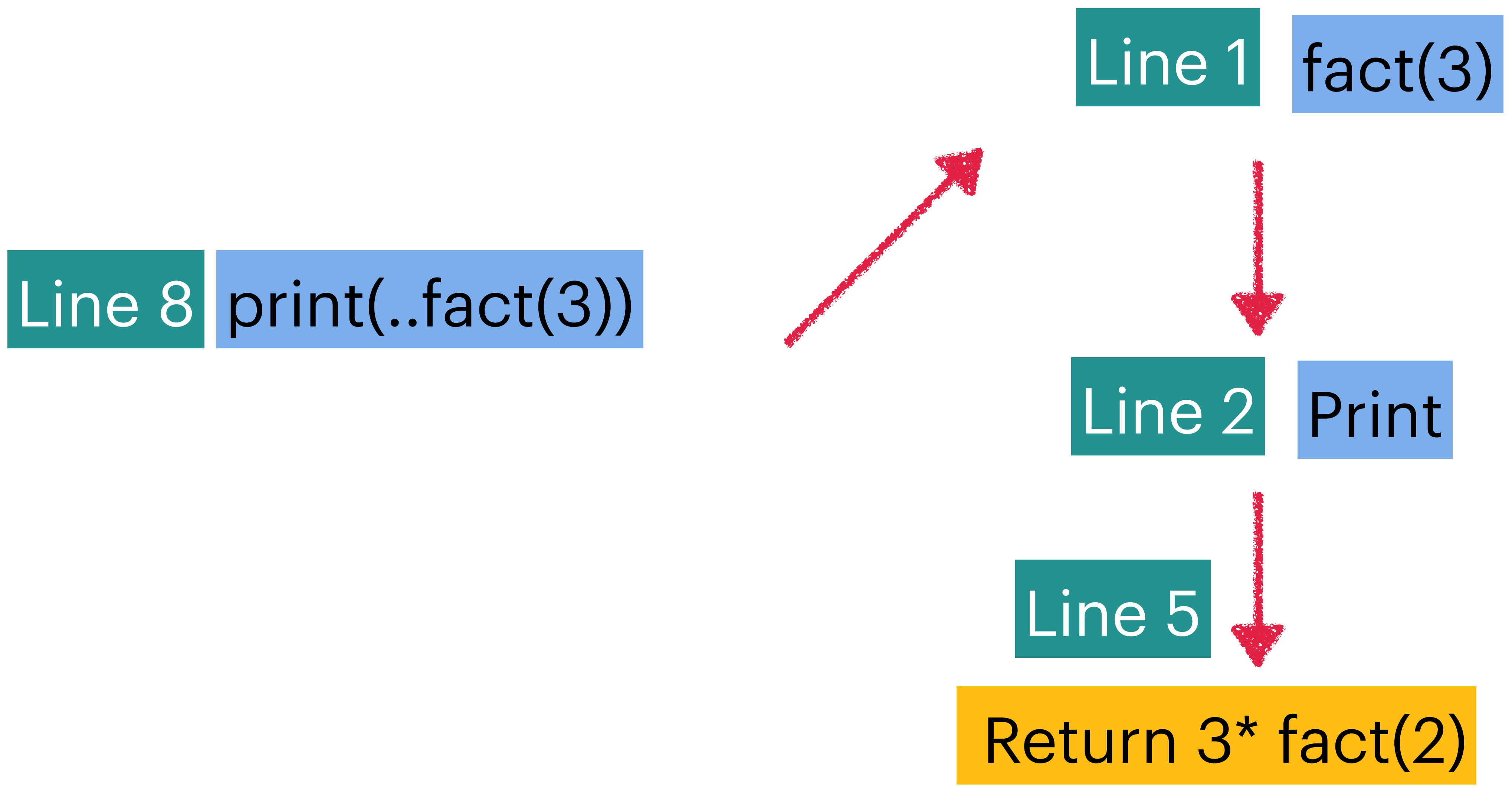
Recursion!

```
1 def fact(n):
2     print(f"inside function, n: {n}")
3     if n == 1:
4         return 1
5     return n * fact(n-1)
6
7 n = 3
8 print(f"factorial of {n} is {fact(3)}")
```

Base case

Recursive case

Each function call waits for the next one to finish.



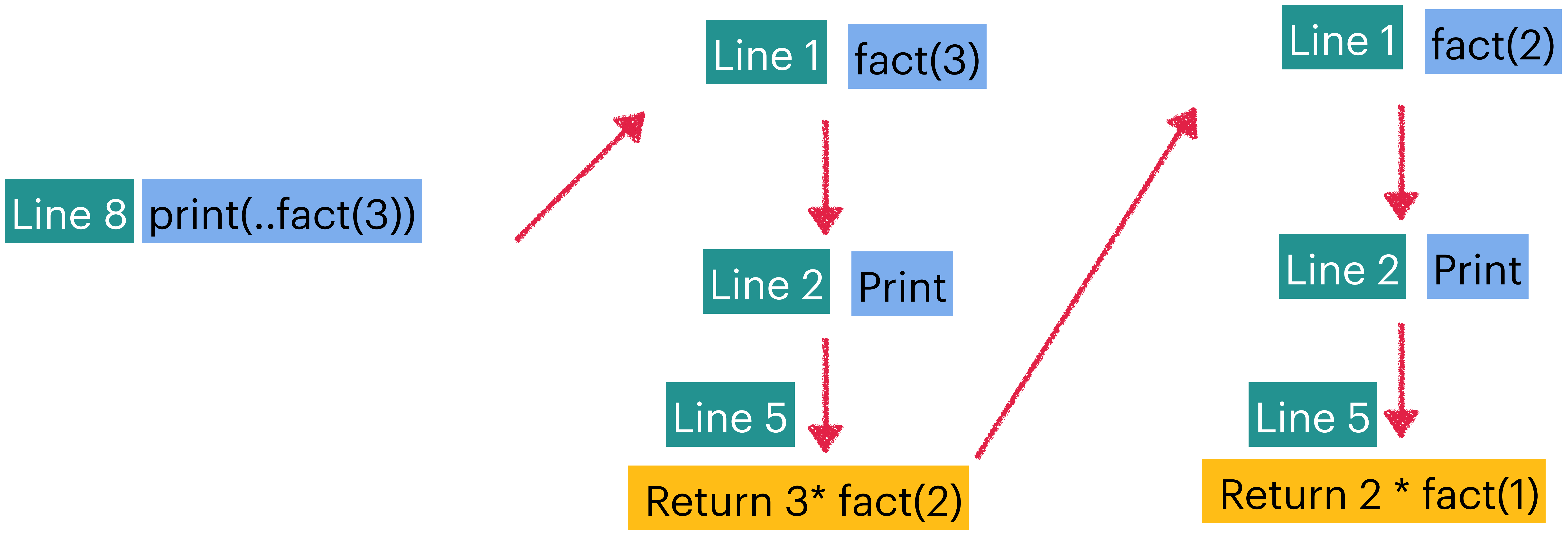
Recursion!

```
1 def fact(n):
2     print(f"inside function, n: {n}")
3     if n == 1:
4         return 1
5     return n * fact(n-1)
6
7 n = 3
8 print(f"factorial of {n} is {fact(3)}")
```

Base case

Recursive case

Each function call waits for the next one to finish.



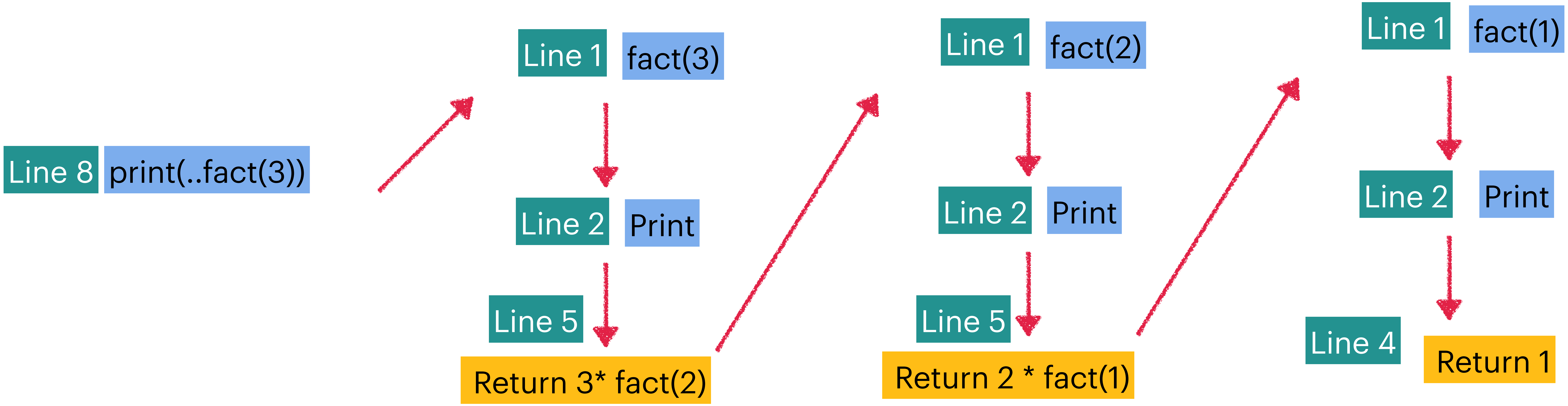
Recursion!

```
1 def fact(n):
2     print(f"inside function, n: {n}")
3     if n == 1:
4         return 1
5     return n * fact(n-1)
6
7 n = 3
8 print(f"factorial of {n} is {fact(3)}")
```

Base case

Recursive case

Each function call waits for the next one to finish.



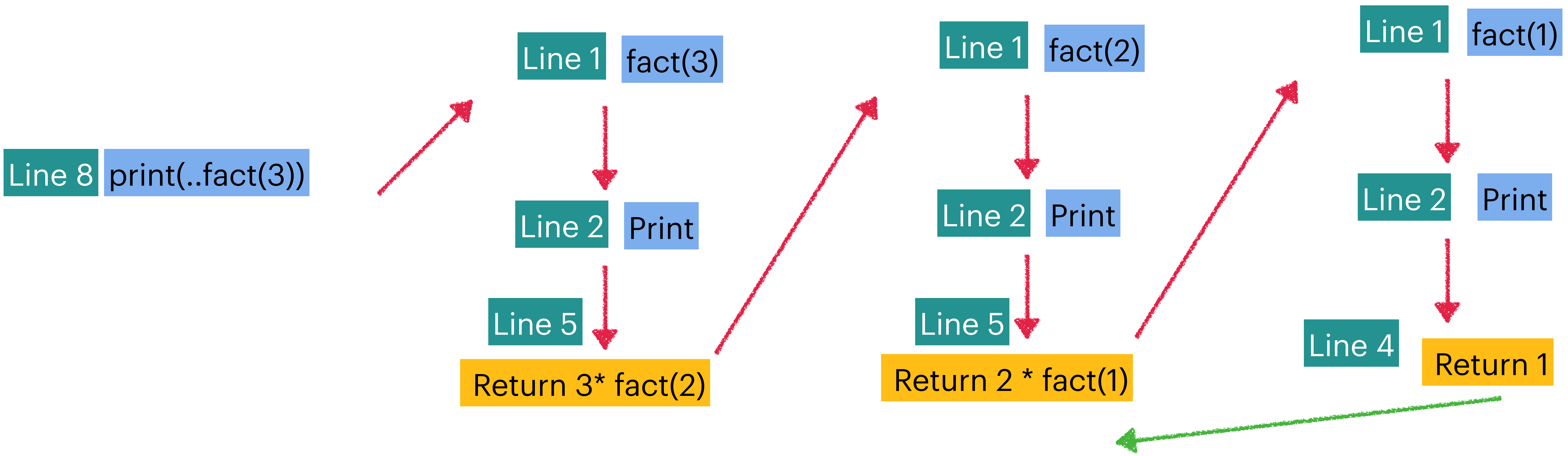
Recursion!

```python
1 def fact(n):
2     print(f"inside function, n: {n}")
3     if n == 1:
4         return 1
5     return n * fact(n-1)
6 
7 n = 3
8 print(f"factorial of {n} is {fact(3)}")
```

Base case

Recursive case

Each function call waits for the next one to finish.



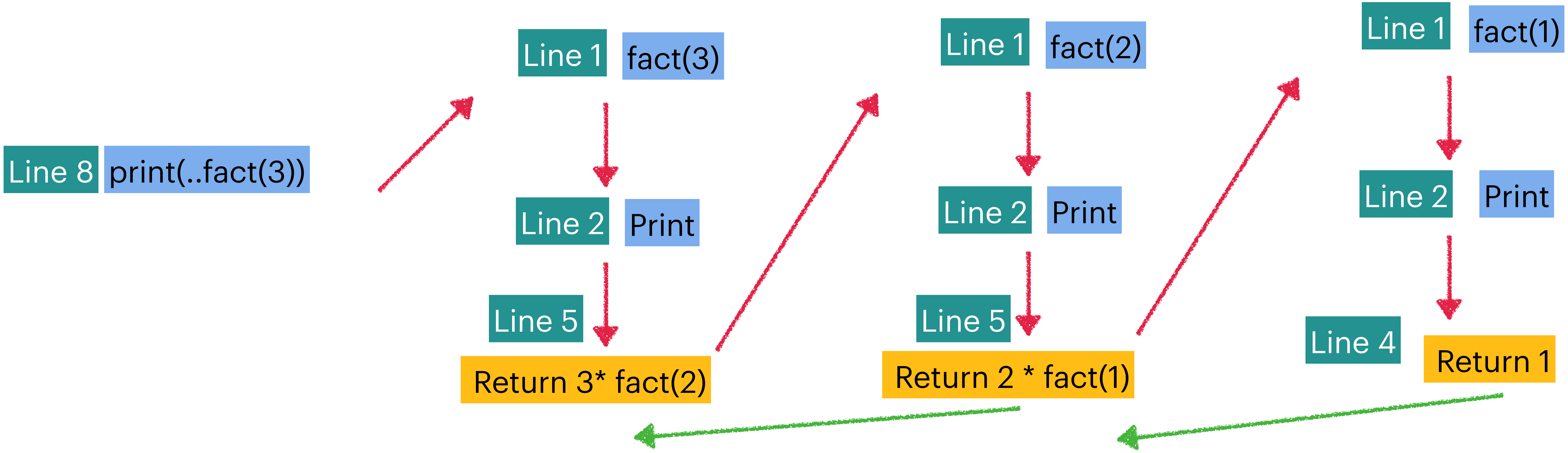
Recursion!

```python
1 def fact(n):
2     print(f"inside function, n: {n}")
3     if n == 1:
4         return 1
5     return n * fact(n-1)
6
7 n = 3
8 print(f"factorial of {n} is {fact(3)}")
```

Base case

Recursive case

Each function call waits for the next one to finish.



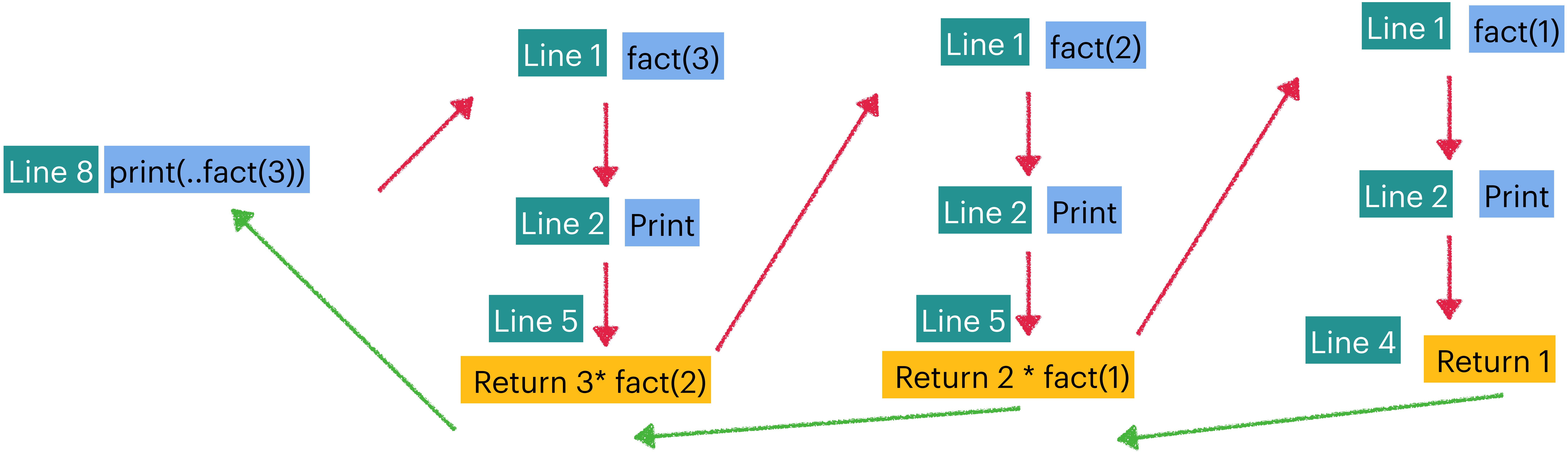
Recursion!

```
1 def fact(n):
2     print(f"inside function, n: {n}")
3     if n == 1:
4         return 1
5     return n * fact(n-1)
6
7 n = 3
8 print(f"factorial of {n} is {fact(3)}")
```

Base case

Recursive case

Each function call waits for the next one to finish.



Recursion!

```python
def fact(n):
    print(f"inside function, n: {n}")
    if n == 1:
        return 1
    return n * fact(n-1)

n = 3
print(f"factorial of {n} is {fact(3)}")
```

```
fact(3)
 └── 3 * fact(2)
           └── 2 * fact(1)
                     └── 1    (base case, returns)
```

Recursion!

```python
def fact(n):
    print(f"inside function, n: {n}")
    if n == 1:
        return 1
    return n * fact(n-1)

n = 3
print(f"factorial of {n} is {fact(3)}")
```

```
fact(3)
 └── 3 * fact(2)
         └── 2 * fact(1)
                 └── 1    (base case, returns)
```

Forgetting the base case → infinite recursion.

Making the recursive call without reducing the problem → still infinite recursion.

Recursion!

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)

n = 3
print(f" {n}th numnber in fibonacci sequence is {fib(3)}")
```

Base cases

Recursive cases

Recursion!

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)

n = 3
print(f" {n}th numnber in fibonacci sequence is {fib(3)}")
```

Base cases

Recursive cases

```
fib(3)                          # not base case → go to recursive case
 ├── fib(2)                     # first evaluate fib(n-1)
 │    ├── fib(1) → 1            # base case → return 1
 │    └── fib(0) → 0            # base case → return 0
 │   = 1 + 0 = 1                # so fib(2) = 1
 └── fib(1) → 1                 # now evaluate fib(n-2), base case
= 1 + 1 = 2                     # so fib(3) = 2
```

Recursion!

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)

n = int(input("enter a number >= 0 "))
seq = []
while n >= 0:
    seq.append(fib(n))
    n -= 1
print("seq is", seq)
```

Recursion!

```
1 def f():
2     return f()
3
4 f()
```

```
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p11169/lec_main.py", line 3, in <module>
    import lec22
  File "/home/p11169/lec22.py", line 4, in <module>
    f()
  File "/home/p11169/lec22.py", line 2, in f
    return f()
  File "/home/p11169/lec22.py", line 2, in f
    return f()
  File "/home/p11169/lec22.py", line 2, in f
    return f()
  [Previous line repeated 982 more times]
RecursionError: maximum recursion depth exceeded
```

Each time a function calls itself, Python adds a new **stack frame** (a memory record of that call) to its *call stack*.
If recursion goes too deep (e.g., an infinite loop or missing base case), the stack keeps growing until Python reaches its **recursion limit** — the maximum depth of the call stack allowed.

Here, there's no base case, so f() keeps calling itself forever.

Tail Recursion

A function is tail-recursive if the recursive call is the last thing the function does before returning — i.e. there is no computation after the recursive call.

```python
def fact(n):
    print(f"inside function, n: {n}")
    if n == 1:
        return 1
    return n * fact(n-1)

n = 3
print(f"factorial of {n} is {fact(3)}")
```

Not tail recursive

```python
def fact(n, acc=1):
    if n == 0:
        return acc
    return fact(n-1, acc * n)
```

tail recursive

Notice fact(n-1, acc * n) is the last statement. There's no multiplication *after* the call returns.

Tail Recursion

A function is tail-recursive if the recursive call is the last thing the function does before returning — i.e. there is no computation after the recursive call.

```python
def fact(n):
    print(f"inside function, n: {n}")
    if n == 1:
        return 1
    return n * fact(n-1)

n = 3
print(f"factorial of {n} is {fact(3)}")
```

Not tail recursive

```python
def fact(n, acc=1):
    if n == 0:
        return acc
    return fact(n-1, acc * n)
```

tail recursive

Notice fact(n-1, acc * n) is the last statement. There's no multiplication *after* the call returns.

In some languages (like Scheme, Haskell, OCaml, Scala), the compiler optimizes tail recursion to reuse the same stack frame — called Tail Call Optimization (TCO).

Recursion!

```python
n = int(input("enter a number"))
e = int(input("enter an exponent"))

def power(n,e):
    if e == 1:
        return n
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

```
enter a number3
enter an exponent4
3^4 is 81
```

Recursion!

```python
n = int(input("enter a number"))
e = int(input("enter an exponent"))

def power(n,e):
    if e == 1:
        return n
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

```
enter a number3
enter an exponent4
3^4 is 81
```

Did it cover all the cases?

Recursion!

```python
n = int(input("enter a number"))
e = int(input("enter an exponent"))

def power(n,e):
    if e == 1:
        return n
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

```
enter a number3
enter an exponent4
3^4 is 81
```

Did it cover all the cases?

What if e is negative ?

Recursion!

```python
n = int(input("enter a number"))
e = int(input("enter an exponent"))

def power(n,e):
    if e == 1:
        return n
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

```
enter a number3
enter an exponent4
3^4 is 81
```

Did it cover all the cases?

What if e is negative ?

What if e is zero?