COL1000
Introduction to Programming

Priyanka Golia

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

Quiz

```
1 def add_item(item, lst = []):
2     lst.append(item)
3     return lst
4
5 print(add_item(10))
6 print(add_item(20))
7 print(add_item(30))
```

Quiz

```python
def add_item(item, lst = []):
    lst.append(item)
    return lst

print(add_item(10))
print(add_item(20))
print(add_item(30))
```

```
[10]
[10, 20]
[10, 20, 30]
```

mutable default arguments (like lists/dicts) are shared across calls.

There is **one list object** created when Python defined the function. Every time you don't pass a list, the parameter lst references that same list object. So each call mutates the same object

Quiz

```python
def add_item(item, lst = []):
    lst.append(item)
    return lst

print(add_item(10))
print(add_item(20))
print(add_item(30))
```

```
[10]
[10, 20]
[10, 20, 30]
```

mutable default arguments (like lists/dicts) are shared across calls.

There is **one list object** created when Python defined the function. Every time you don't pass a list, the parameter lst references that same list object. So each call mutates the same object

```python
def lists():
    return [1,2,3]

a = lists()
b = lists()
print(a is b)
```

Quiz

```python
def add_item(item, lst = []):
    lst.append(item)
    return lst

print(add_item(10))
print(add_item(20))
print(add_item(30))
```

```
[10]
[10, 20]
[10, 20, 30]
```

mutable default arguments (like lists/dicts) are shared across calls.

There is **one list object** created when Python defined the function. Every time you don't pass a list, the parameter lst references that same list object. So each call mutates the same object

```python
def lists():
    return [1,2,3]

a = lists()
b = lists()
print(a is b)
```

Don't get confuse. Here, in each call, we are creating a new object.

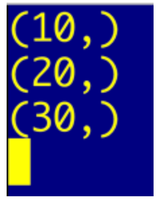
False

Quiz

```
1 def add_item(item, tpl = ()):
2     tpl += (item,)
3     return tpl
4
5 print(add_item(10))
6 print(add_item(20))
7 print(add_item(30))
```

Quiz

```
1 def add_item(item, tpl = ()):
2     tpl += (item,)
3     return tpl
4
5 print(add_item(10))
6 print(add_item(20))
7 print(add_item(30))
```



Quiz

```python
def add_item(item, tpl = ()):
    tpl += (item,)
    return tpl

print(add_item(10))
print(add_item(20))
print(add_item(30))
```

```
(10,)
(20,)
(30,)
```

```python
def cal():
    print(f"x:{x}")
    print(f"y:{y}")
    print(f"z:{z}")

x = 1
y = 2
cal()
```

Quiz

```python
def add_item(item, tpl = ()):
    tpl += (item,)
    return tpl

print(add_item(10))
print(add_item(20))
print(add_item(30))
```

```
(10,)
(20,)
(30,)
```

```python
def cal():
    print(f"x:{x}")
    print(f"y:{y}")
    print(f"z:{z}")

x = 1
y = 2
cal()
```

```
x:1
y:2
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p10290/lec_main.py", line 3, in <module>
    import lec22
  File "/home/p10290/lec22.py", line 8, in <module>
    cal()
  File "/home/p10290/lec22.py", line 4, in cal
    print(f"z:{z}")
NameError: name 'z' is not defined
```

Quiz

```python
def add_item(item, tpl = ()):
    tpl += (item,)
    return tpl

print(add_item(10))
print(add_item(20))
print(add_item(30))
```

```
(10,)
(20,)
(30,)
```

```python
def cal():
    print(f"x:{x}")
    print(f"y:{y}")
    y = x + 1

x = 1
y = 2
cal()
```

```python
def cal():
    print(f"x:{x}")
    print(f"y:{y}")
    print(f"z:{z}")

x = 1
y = 2
cal()
```

```
x:1
y:2
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p10290/lec_main.py", line 3, in <module>
    import lec22
  File "/home/p10290/lec22.py", line 8, in <module>
    cal()
  File "/home/p10290/lec22.py", line 4, in cal
    print(f"z:{z}")
NameError: name 'z' is not defined
```

Quiz

```python
def add_item(item, tpl = ()):
    tpl += (item,)
    return tpl

print(add_item(10))
print(add_item(20))
print(add_item(30))
```

```
(10,)
(20,)
(30,)
```

```python
def cal():
    print(f"x:{x}")
    print(f"y:{y}")

x = 1
y = 2
cal()
```

```
x:1
y:2
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p10290/lec_main.py", line 3, in <module>
    import lec22
  File "/home/p10290/lec22.py", line 8, in <module>
    cal()
  File "/home/p10290/lec22.py", line 4, in cal
    print(f"z:{z}")
NameError: name 'z' is not defined
```

```python
def cal():
    print(f"x:{x}")
    print(f"y:{y}")
    y = x + 1

x = 1
y = 2
cal()
```

```
x:1
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p10619/lec_main.py", line 3, in <module>
    import lec22
  File "/home/p10619/lec22.py", line 8, in <module>
    cal()
  File "/home/p10619/lec22.py", line 3, in cal
    print(f"y:{y}")
UnboundLocalError: local variable 'y' referenced before assignment
```

Quiz

```
1 def cal(a,b):
2     a += a
3     b += b
4     print(f"inside function a:{a} and b:{b}")
5     return a, b
6     a = 3 * a
7     print(f"inside function a:{a} and b:{b}")
8     return a, b
9
10 x, y = cal(3,5)
11 print(f"outside function x:{x}, y:{y}")
```

Quiz

```python
def cal(a,b):
    a += a
    b += b
    print(f"inside function a:{a} and b:{b}")
    return a, b
    a = 3 * a
    print(f"inside function a:{a} and b:{b}")
    return a, b

x, y = cal(3,5)
print(f"outside function x:{x}, y:{y}")
```

```
inside function a:6 and b:10
outside function x:6, y:10
```

Quiz

```python
def cal(a,b):
    a += a
    b += b
    print(f"inside function a:{a} and b:{b}")
    return a, b
    a = 3 * a
    print(f"inside function a:{a} and b:{b}")
    return a, b

x, y = cal(3,5)
print(f"outside function x:{x}, y:{y}")
```

```
inside function a:6 and b:10
outside function x:6, y:10
```

The lines after the first return are **never executed** because a return immediately exits the function.

Quiz

```
1  def cal(a,b):
2      a += a
3      b += b
4      print(f"inside function a:{a} and b:{b}")
5      return a, b
6      a = 3 * a
7      print(f"inside function a:{a} and b:{b}")
8      return a, b
9
10 x, y = cal(3,5)
11 print(f"outside function x:{x}, y:{y}")
```

Unreachable code

```
inside function a:6 and b:10
outside function x:6, y:10
```

The lines after the first return are **never executed** because a return immediately exits the function.

Type Annotations!

Python is **dynamically typed**, but since Python 3.5+, you can add **type hints** (annotations) to improve readability and help tools like *mypy* or IDEs check types.

Type annotations: Provide hints about parameter and return types but are not enforced at runtime. Tools like mypy and IDEs enforce it.

```
def add(x: int, y: int) -> int:
    return x + y
```

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!

```
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.

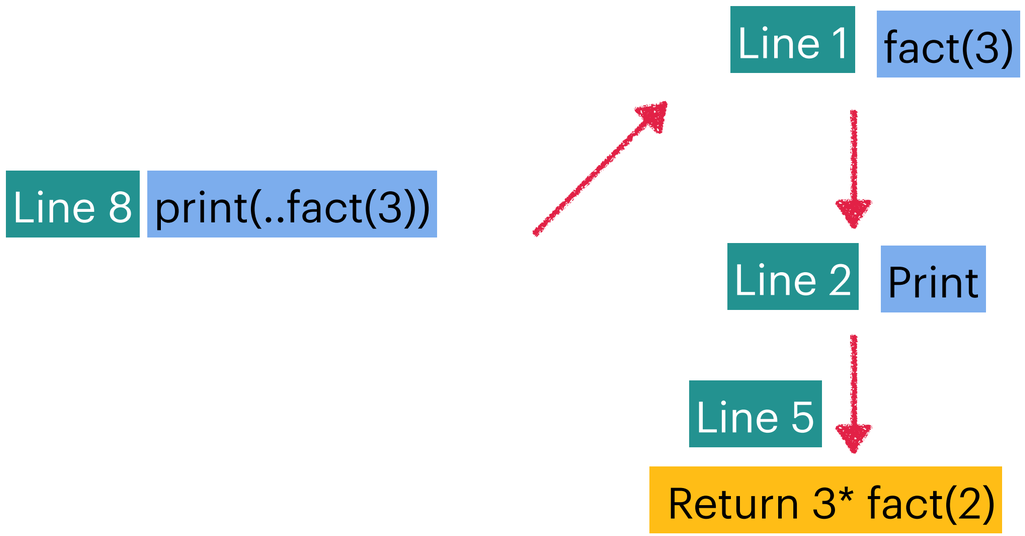
Recursion!

```
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.



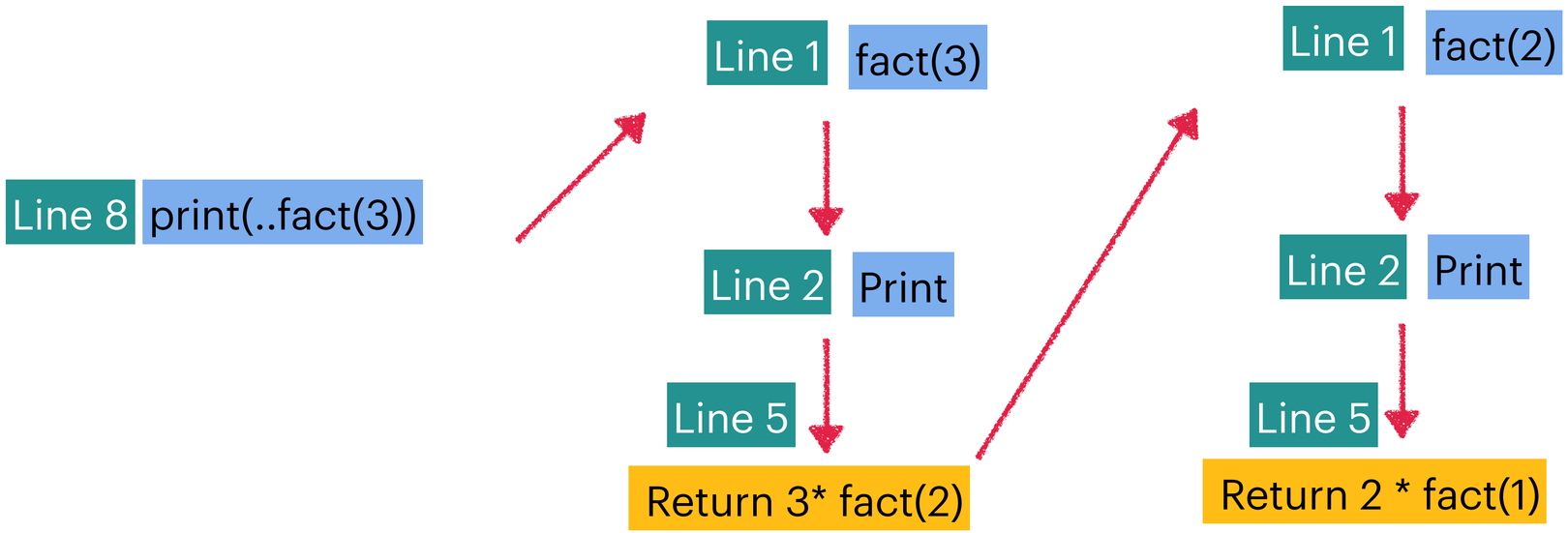
Recursion!

```
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.



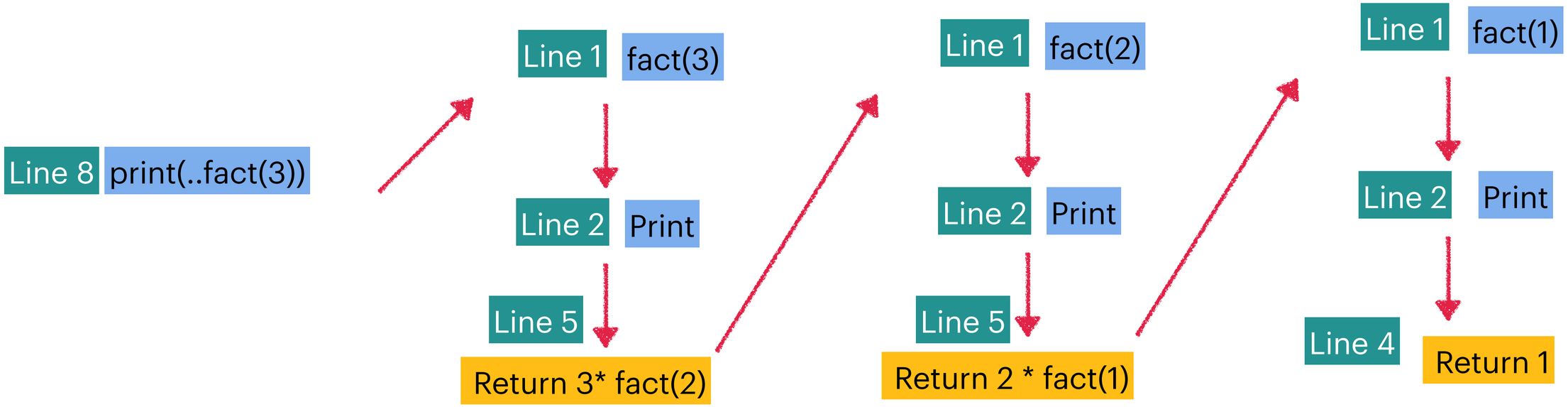
Recursion!

```
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.



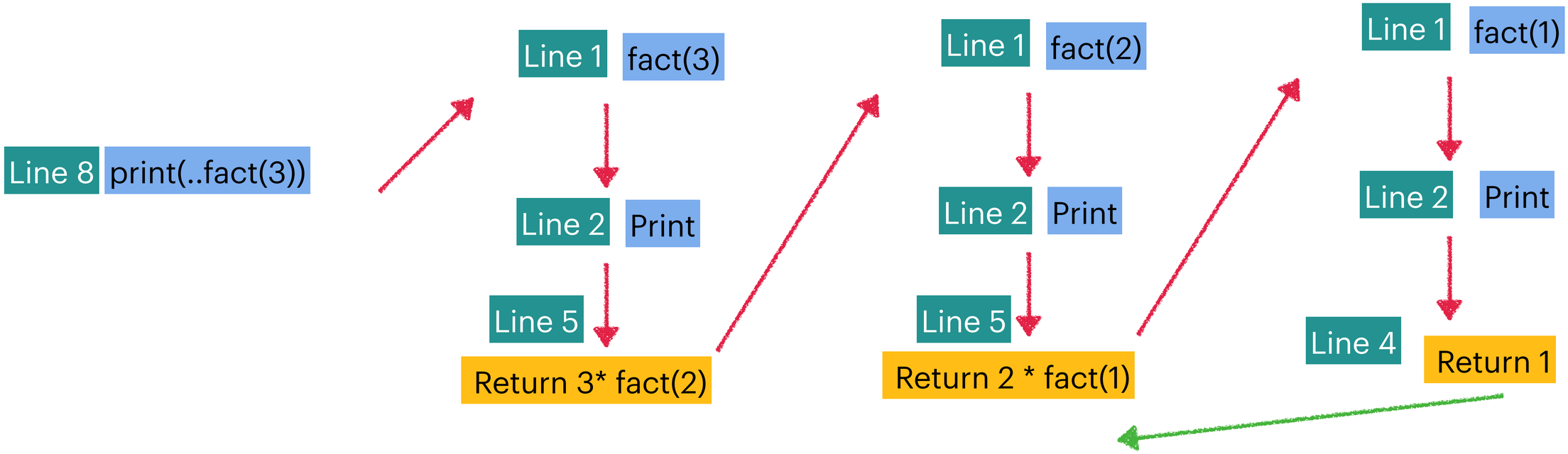
Recursion!

```
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.



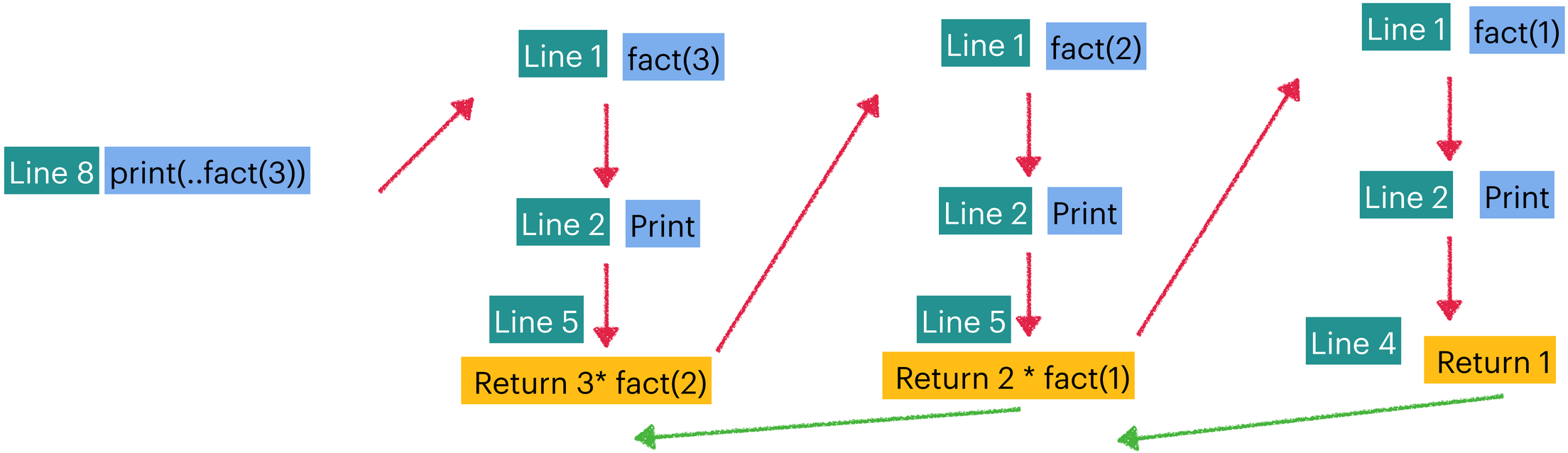
Recursion!

```
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!

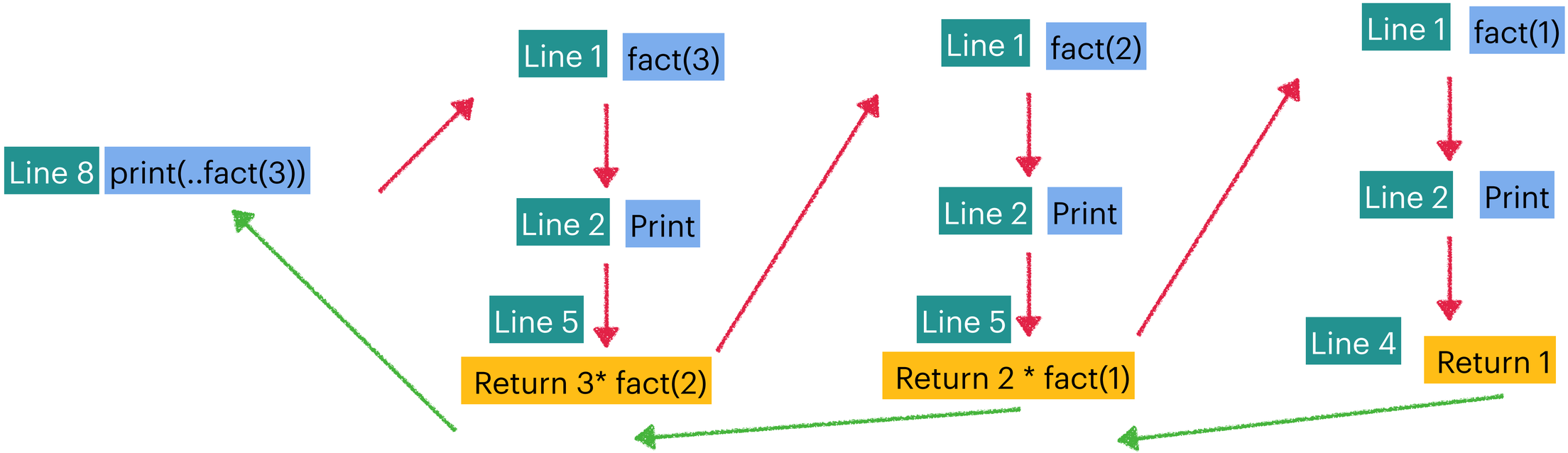
```python
def fact(n):
    print(f"inside function, n: {n}")
    if n == 1:
        return 1
    return n * fact(n-1)

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!

```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.