

IT5001 Software Development Fundamentals

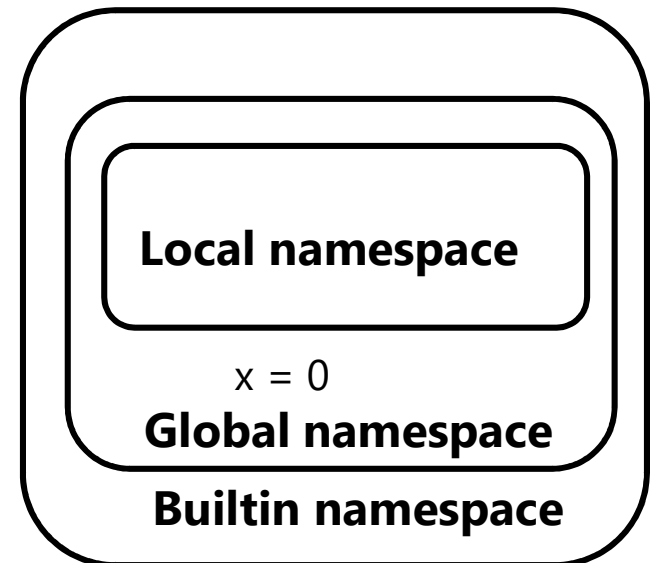
4. Functions, Scope, and Recursion

Sirigina Rajendra Prasad

Scope

Global vs Local Variables

- A variable which is defined in the main body of a file is called a **global** variable. It will be **visible throughout the file**, and also inside any file which imports that file. EXCEPT...
- A variable which is defined inside a function is **local** to that function. It is accessible **from the point at which it is defined until the end of the function**, and exists for as long as the function is executing.
- The parameter names in the function definition behave like local variables, but they contain the values that we pass into the function when we call it.

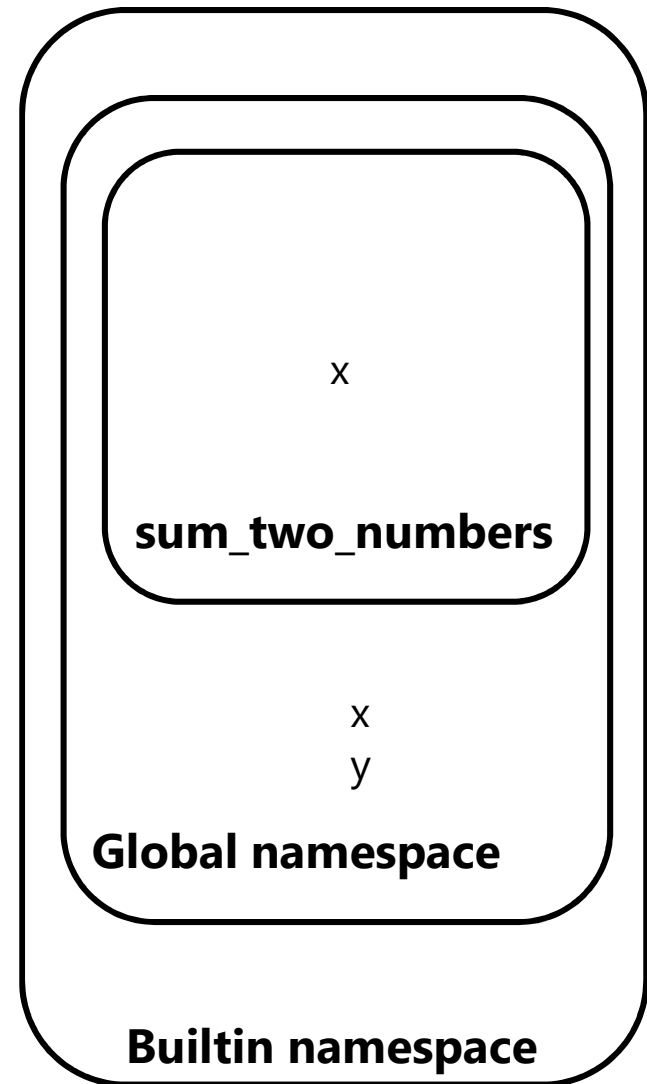
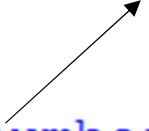


Example

```
x = 2
y = 4
def sum_two_numbers(x):
    return x+y

z = sum_two_numbers(3)
print(z)
```

access global variable y

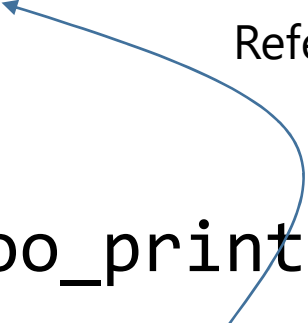


Global Variable

```
x = 0
```

Refers to

```
def foo_printx():  
    print(x)
```

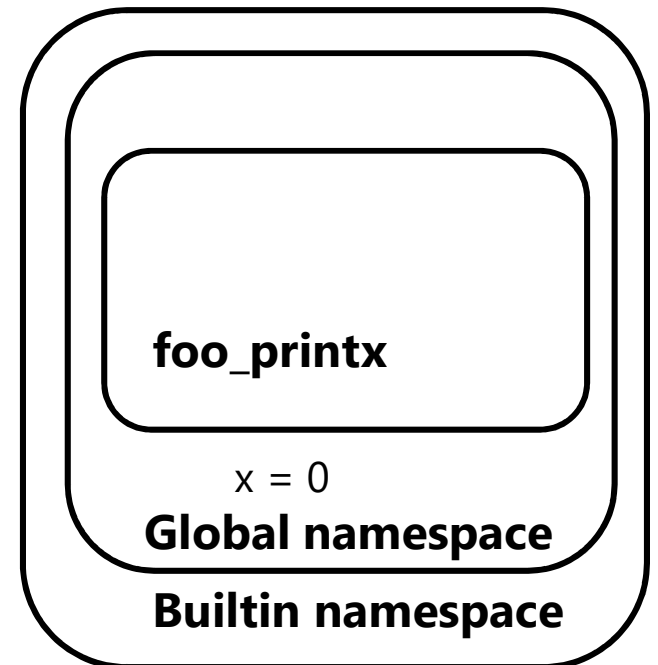


```
foo_printx()  
print(x)
```

- This code will print

0

0



Global vs Local Variables

```
x = 0
```

```
def foo_printx():
```

```
    x = 999
```

```
    print(x)
```

```
foo_printx()
```

```
print(x)
```

Because, a new 'x'
is born here!

- This code will print

999

0

- The first '999' makes sense
- But why the second one is '0'?

Global vs Local Variables

A Global 'x'

```
x = 0
```

- This code will print
999
0

```
def foo_printx():
```

```
    x = 999
```

```
    print(x)
```

```
foo_printx()
```

```
print(x)
```

Scope of the local 'x'

Scope of the global 'x'

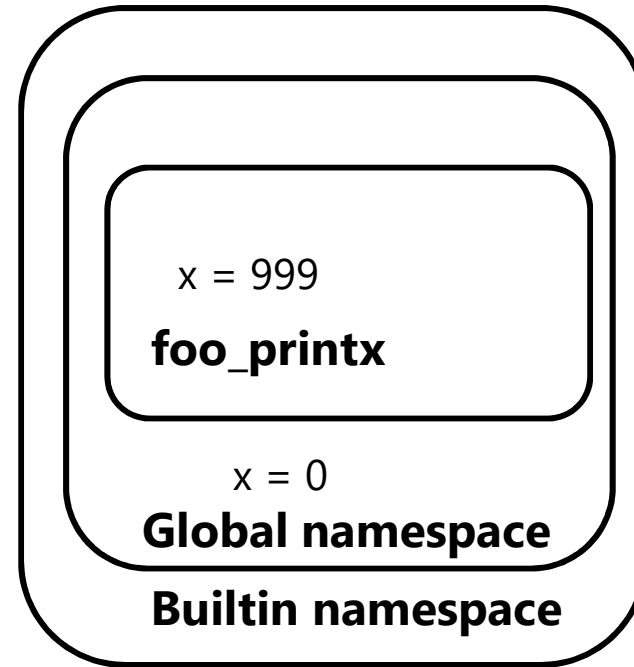
A local 'x' that is created within the function foo_printx() and will 'die' after the function exits

Global vs Local Variables

```
x = 0
```

```
def foo_printx():  
    x = 999  
    print(x)
```

```
foo_printx()  
print(x)
```



Crossing Boundary

- What if we really want to modify a global variable from inside a function?
- Use the "global" keyword
- (No local variable x is created)

```
x = 0
```

```
def foo_printx():  
    global x  
    x = 999  
    print(x)
```

```
foo_printx()  
print(x)
```

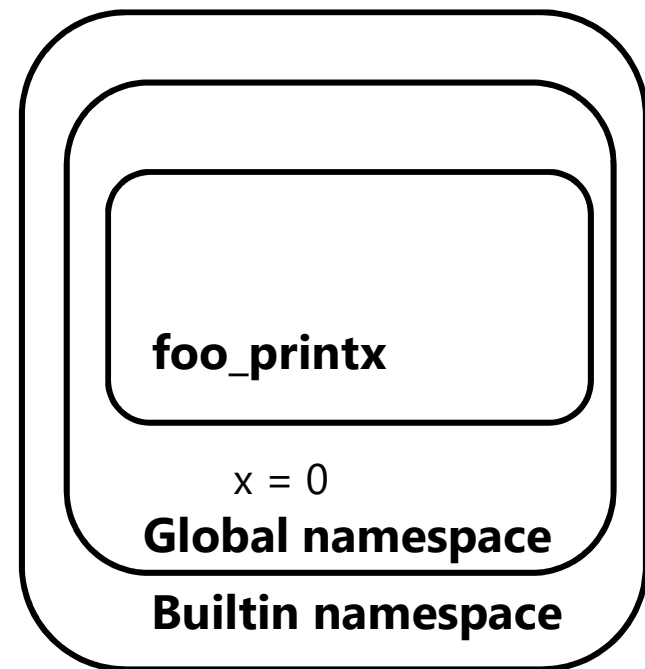
Output:
999
999

Crossing Boundary

```
x = 0
```

```
def foo_printx():  
    global x  
    x = 999  
    print(x)
```

```
foo_printx()  
print(x)
```



Output:
999
999

How about... this?

```
x = 0
```

```
def foo_printx():
```

```
    print(x)
```

```
    x = 999
```

```
    print(x)
```

```
foo_printx()
```

- Local or global?
- Error!
- Because the line "x=999" creates a local version of 'x'
- Then the first print(x) will reference a **local** x that is not assigned with a value
- The line that causes an error

Parameters are LOCAL variables

Scope of x in
p1

```
def p1(x):  
    print('Entering function p1')  
    output = p2(x)  
    print('Line before return in p1')  
    return output
```

Scope of x in
p2

```
def p2(x):  
    print('Entering function p2')  
    output = p3(x)  
    print('Line before return in p2')  
    return output
```

Scope of x in
p3

```
def p3(x):  
    print('Entering function p3')  
    output = x * x  
    print('Line before return in p3')  
    return output
```

Does not refer to

```
print(p1(3))
```

Practices (Convention)

- Global variables are VERY **bad**, especially if modification is allowed
- 99% of time, global variables are used as CONSTANTS
 - Variables that every function could access
 - But not expected to be modified

Convention:
Usually in all CAPs

```
POUNDS_IN_ONE_KG = 2.20462

def kg2pound(w) :
    return w * POUNDS_IN_ONE_KG

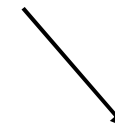
def pound2kg(w) :
    return w / POUNDS_IN_ONE_KG
```

Generator Functions

Generator Functions

```
def function(arg_1, arg_2,..., arg_n):  
    <statement>  
    <statement>  
    .  
    .  
    .  
    return <statement>
```

```
def generator_function(arg_1, arg_2,..., arg_n):  
    <statement>  
    <statement>  
    .  
    .  
    .  
    yield <statement>
```



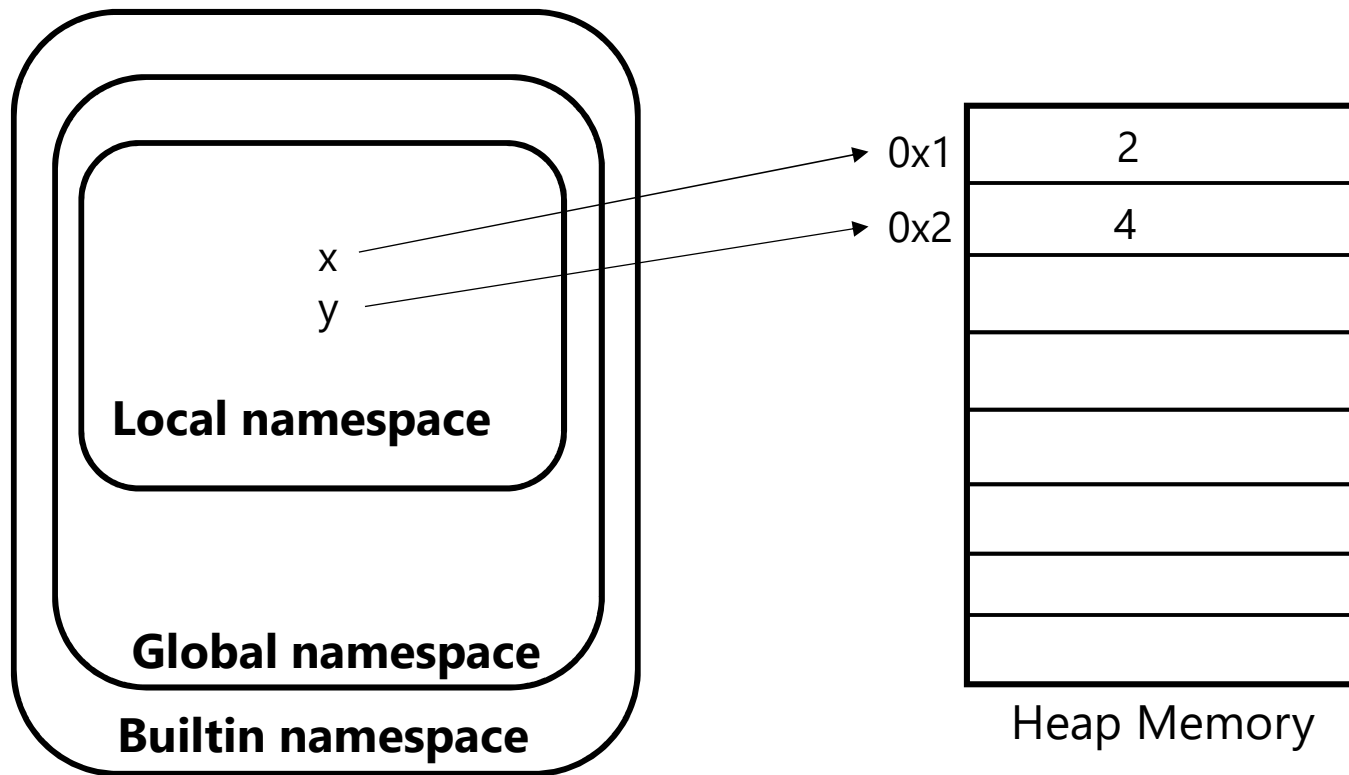
only difference

Return vs Yield

- With **return** statement:
 - State is not retained after the function returns the value
- With **yield** statement:
 - State of the function is retained between the calls
 - Can have many **yield** statements in sequence

What is the state of a function?

- Namespace and the objects that are referred by names



Generator Functions: Examples

```
def my_range(start = 0, stop = None, step = 1):  
    element = start  
    while element <= stop:  
        yield element  
        element = element + step
```

```
from math import inf  
for item in my_range(0, stop = inf):  
    print(item)
```

Generator Functions: Examples

```
def even_range(start = 0, stop = None, step = 1):  
    element = start  
    while element <= stop:  
        if element%2 == 0:  
            yield element  
        element = element + step
```

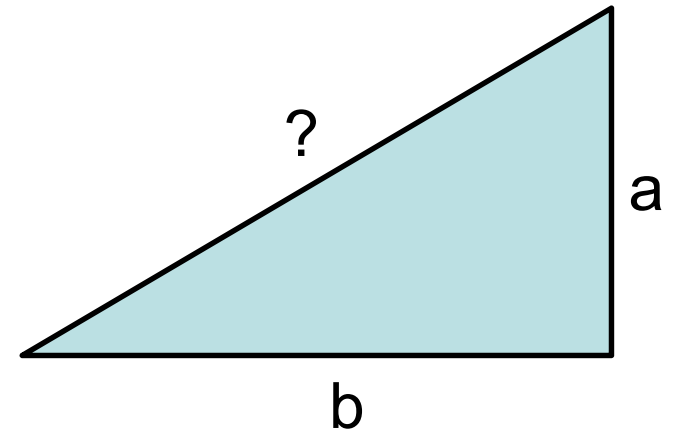
Calling Other Functions

Compare:

```
def hypotenuse(a, b):  
    return sqrt(sum_of_squares(a, b))
```

```
def sum_of_squares(x, y):  
    return square(x) + square(y)
```

```
def square(x):  
    return x * x
```



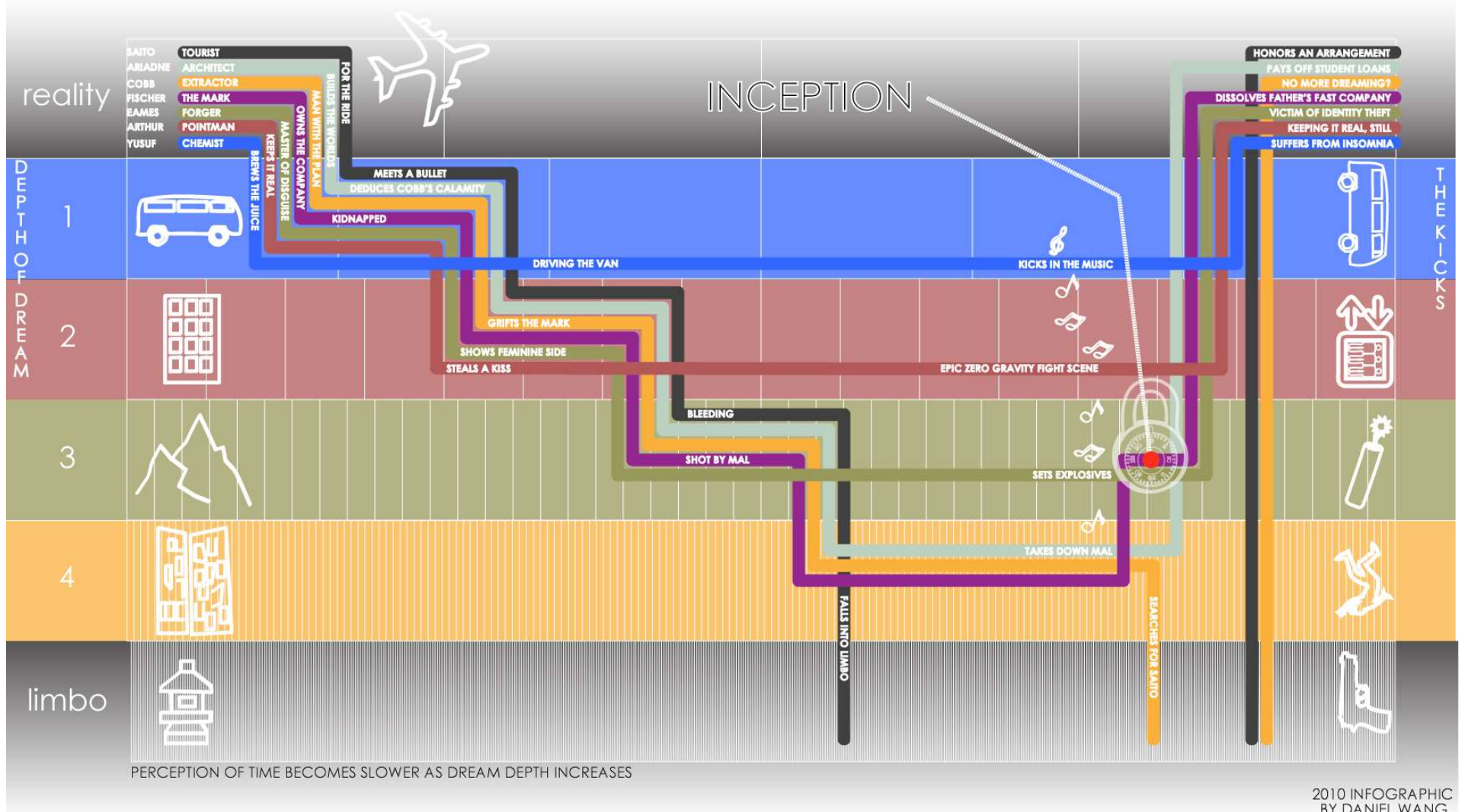
Versus:

```
def hypotenuse(a, b):  
    return sqrt((a*a) + (b*b))
```



The Call Stack

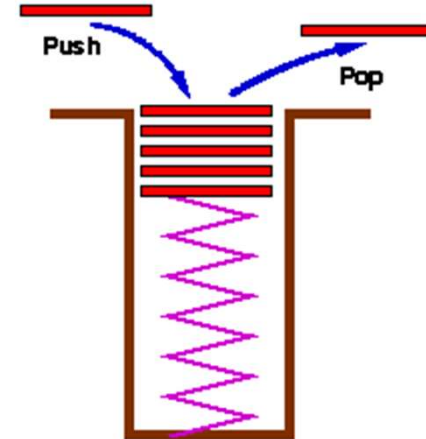




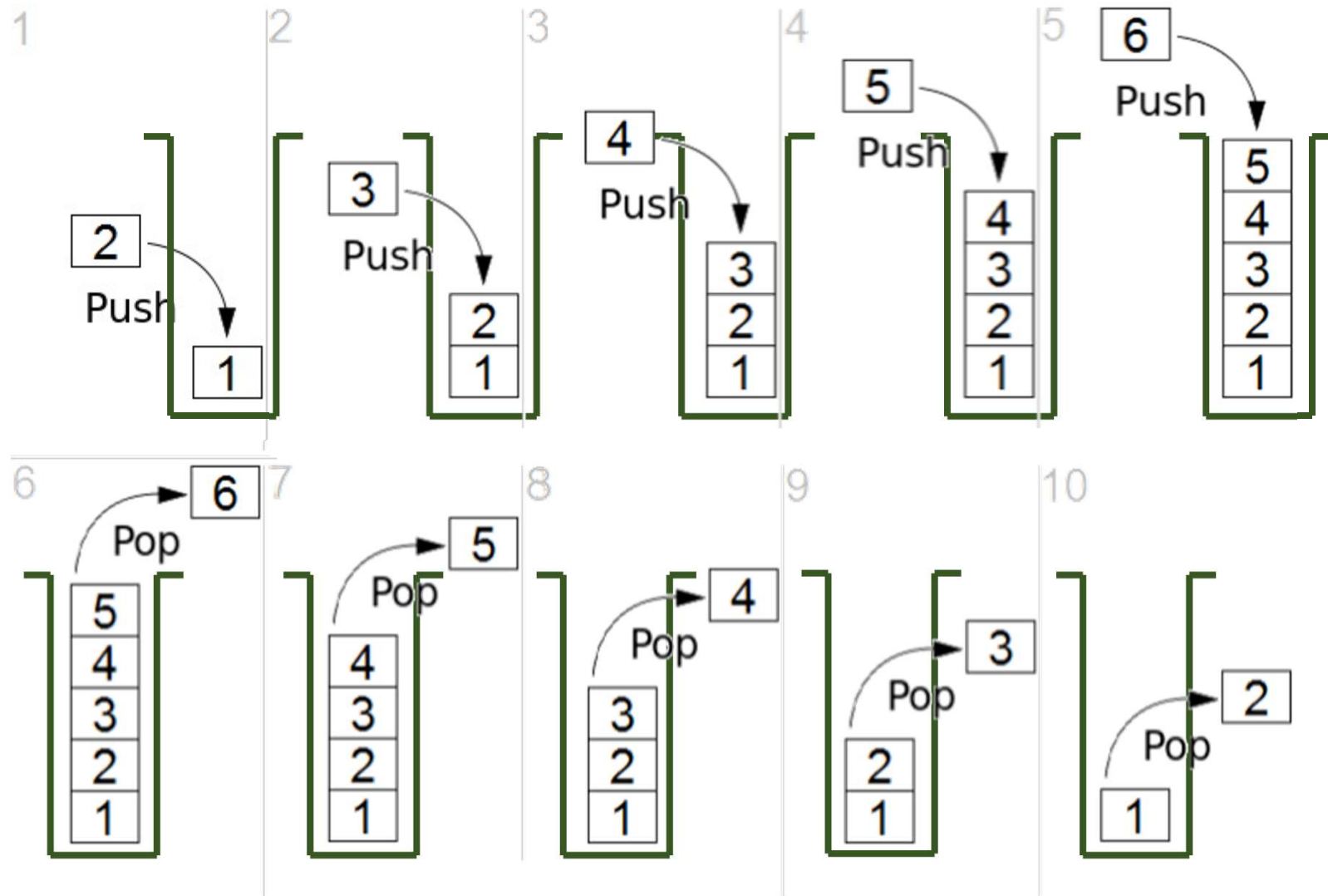
[Michael Caine Just Ended An Eight Year Long Debate Over The Ending Of "Inception"](#)

Stack

- First in last out order



First in Last Out



The Stack (or the Call Stack)

```
def p1(x):  
    print('Entering function p1')  
    output = p2(x)  
    print('Line before return in p1')  
    return output
```

```
def p2(x):  
    print('Entering function p2')  
    output = p3(x)  
    print('Line before return in p2')  
    return output
```

```
def p3(x):  
    print('Entering function p3')  
    output = x * x  
    print('Line before return in p3')  
    return output
```

```
print(p1(3))
```

The Stack (or the Call Stack)

```
>>> p1(3)
```

```
Entering function p1
```

```
Entering function p2
```

```
Entering function p3
```

```
Line before return in p3
```

```
Line before return in p2
```

```
Line before return in p1
```

```
9
```

The text "FILO!" is displayed in a large, bold, black font inside a solid orange rectangular box. This box is positioned in the bottom right area of the slide, visually representing the Last In, First Out (FILO) principle of a stack.

```
print(p1(3))
```

→ Going in
→ Exiting a function

```
def p1(x):
```

```
    print('Entering function p1')
```

```
    output = p2(x)
```

```
    print('Line before return in p1')
```

```
    return output
```

```
def p2(x):
```

```
    print('Entering function p2')
```

```
    output = p3(x)
```

```
    print('Line before return in p2')
```

```
    return output
```

```
def p3(x):
```

```
    print('Entering function p3')
```

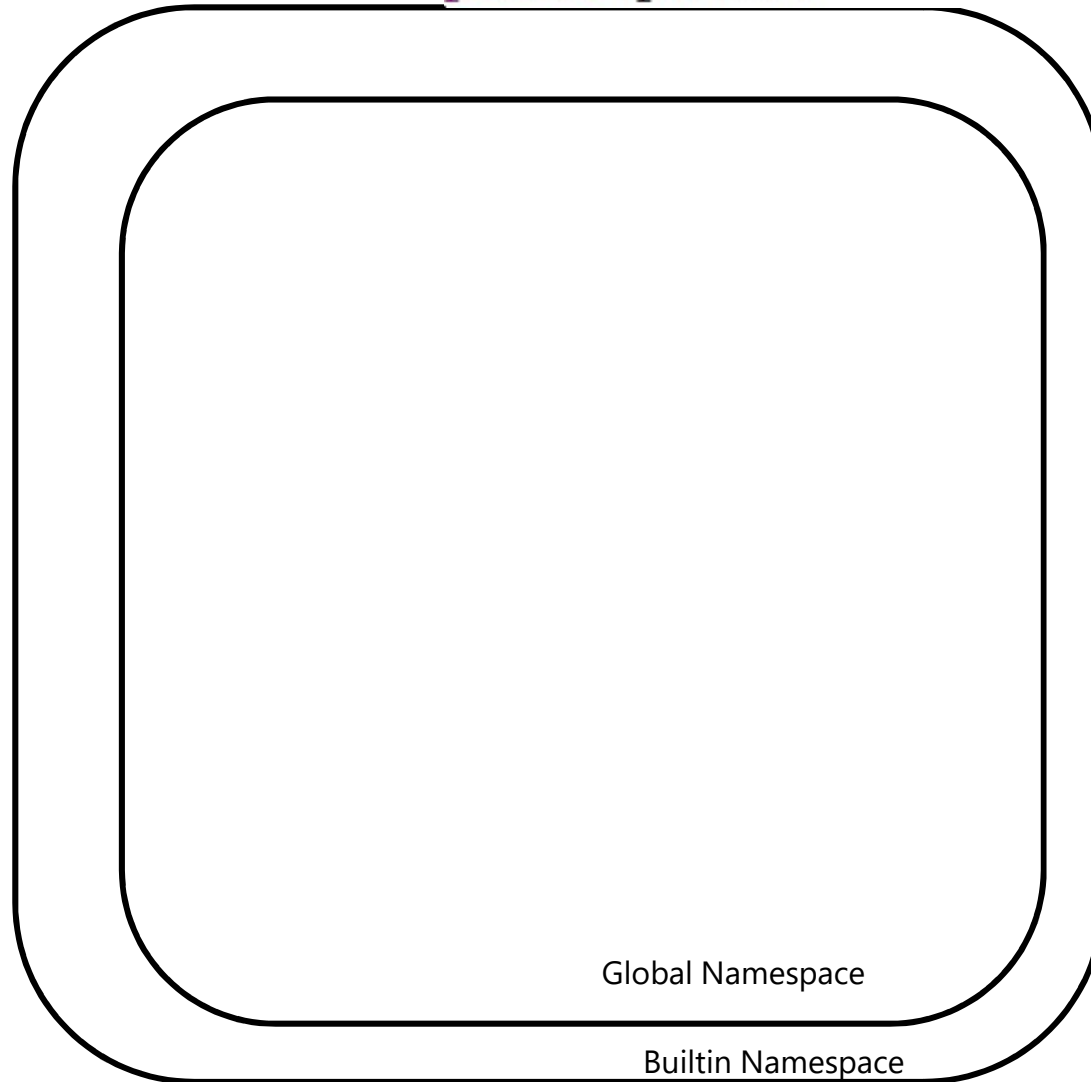
```
    output = x * x
```

```
    print('Line before return in p3')
```

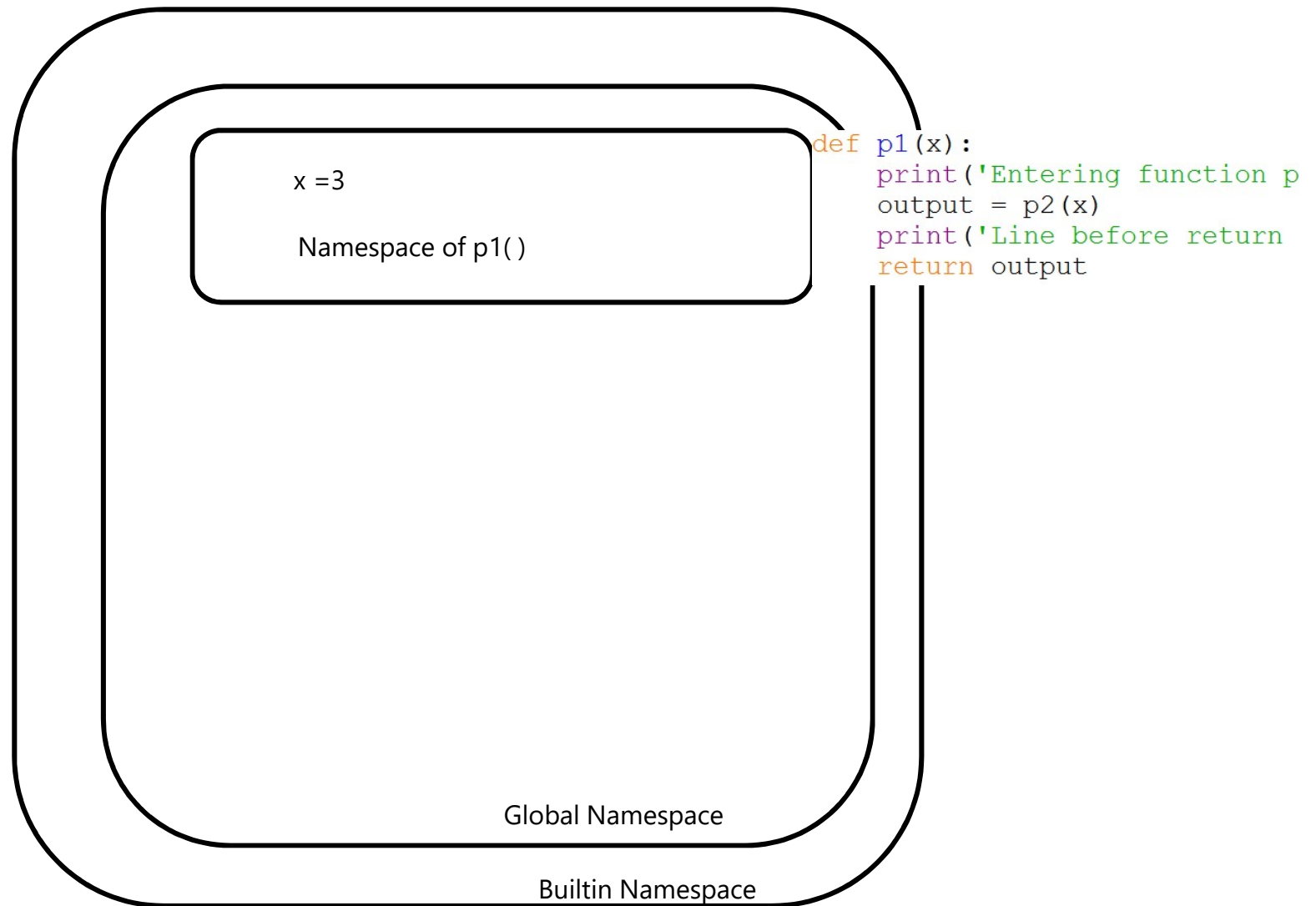
```
    return output
```

Namespaces: Calling Other Functions

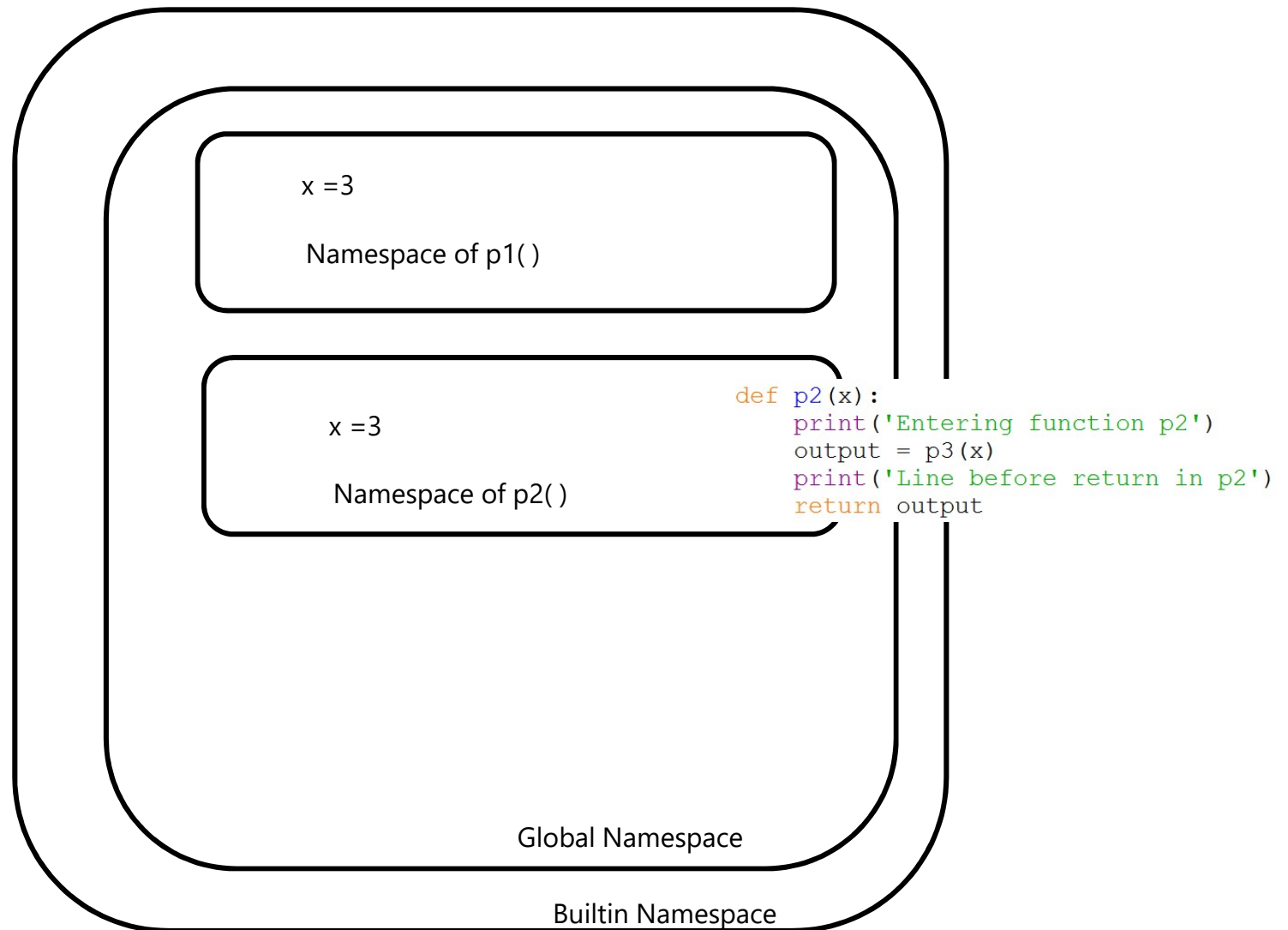
```
print(p1(3))
```



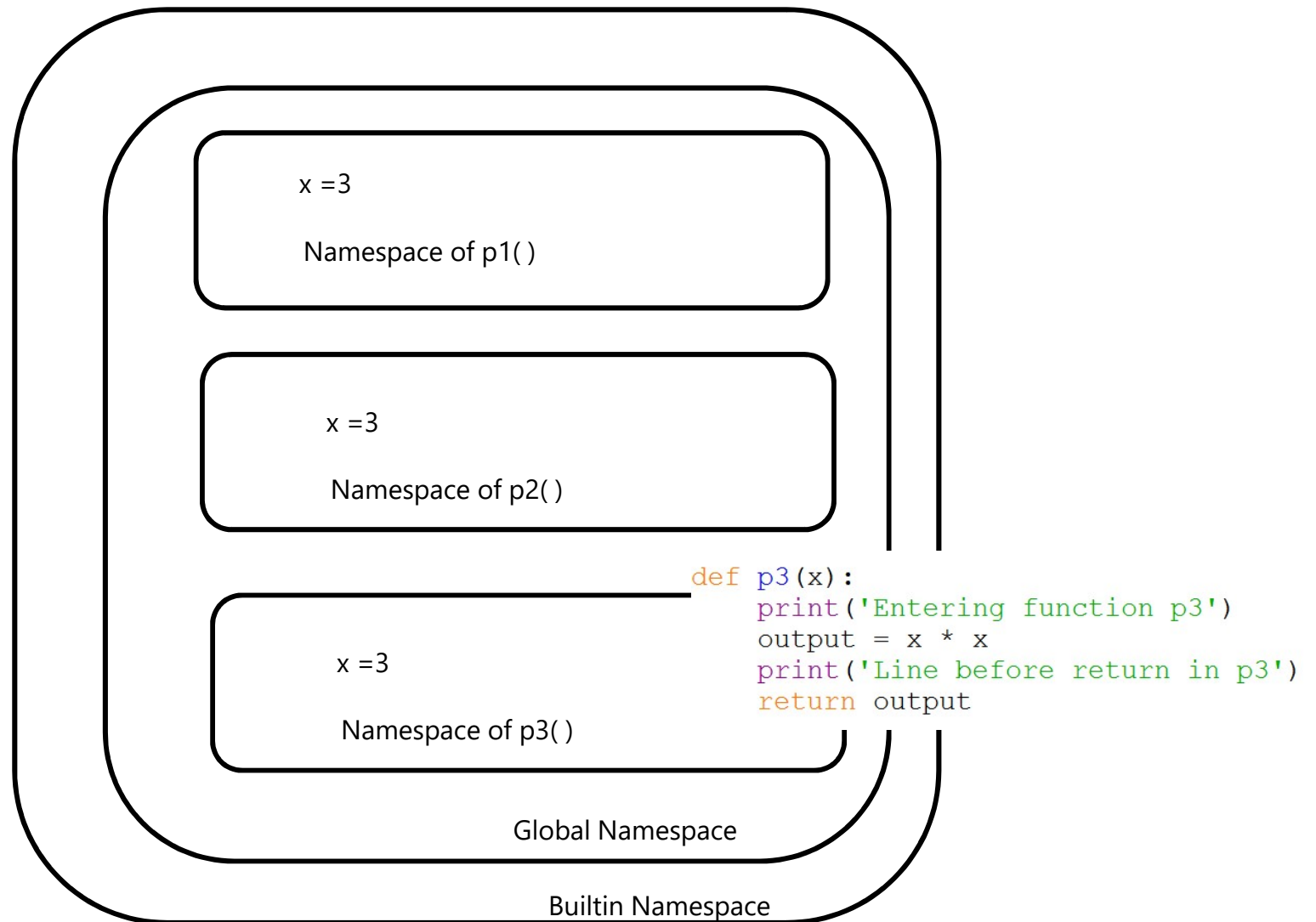
Namespaces: Calling Other Functions



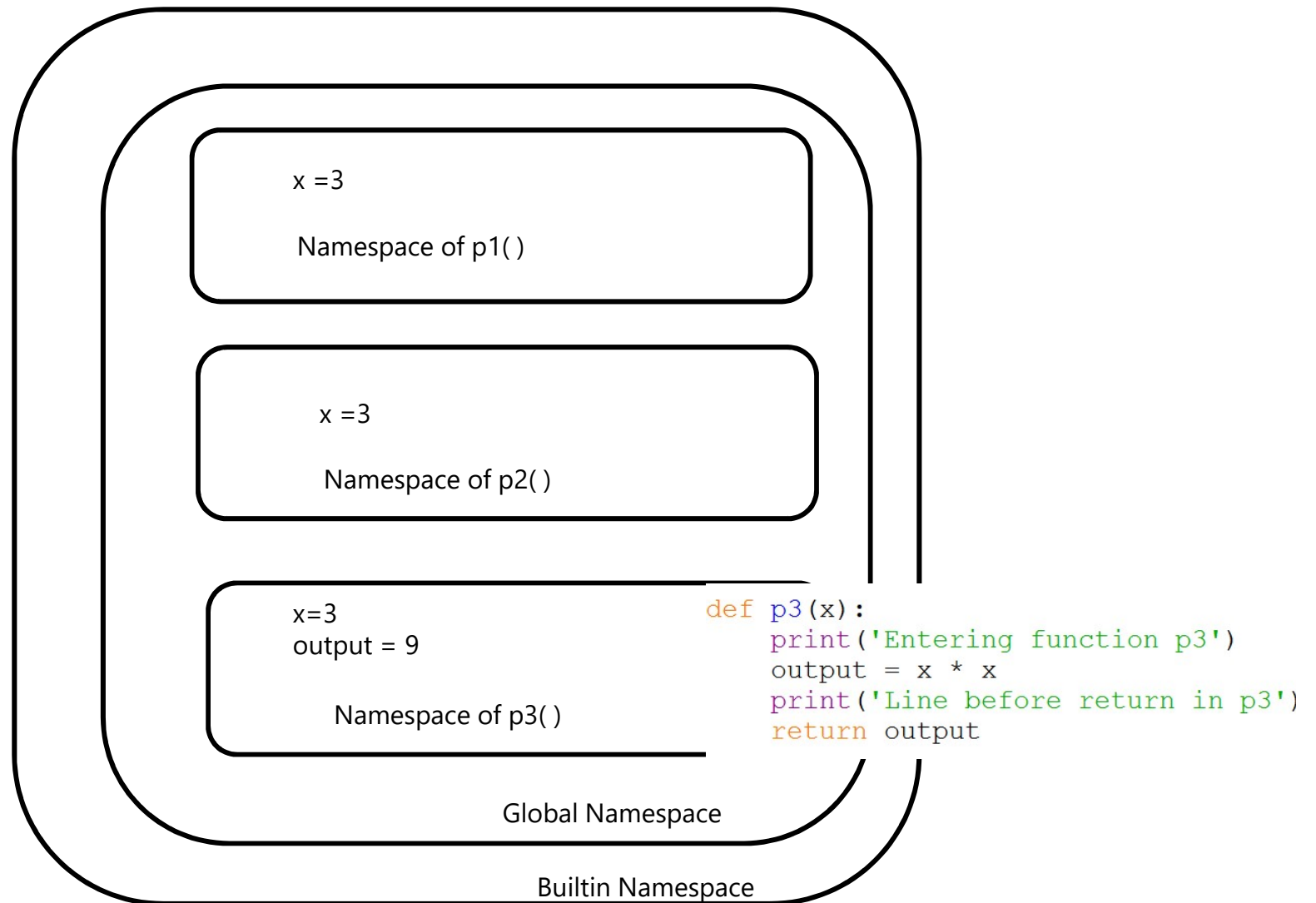
Namespaces: Calling Other Functions



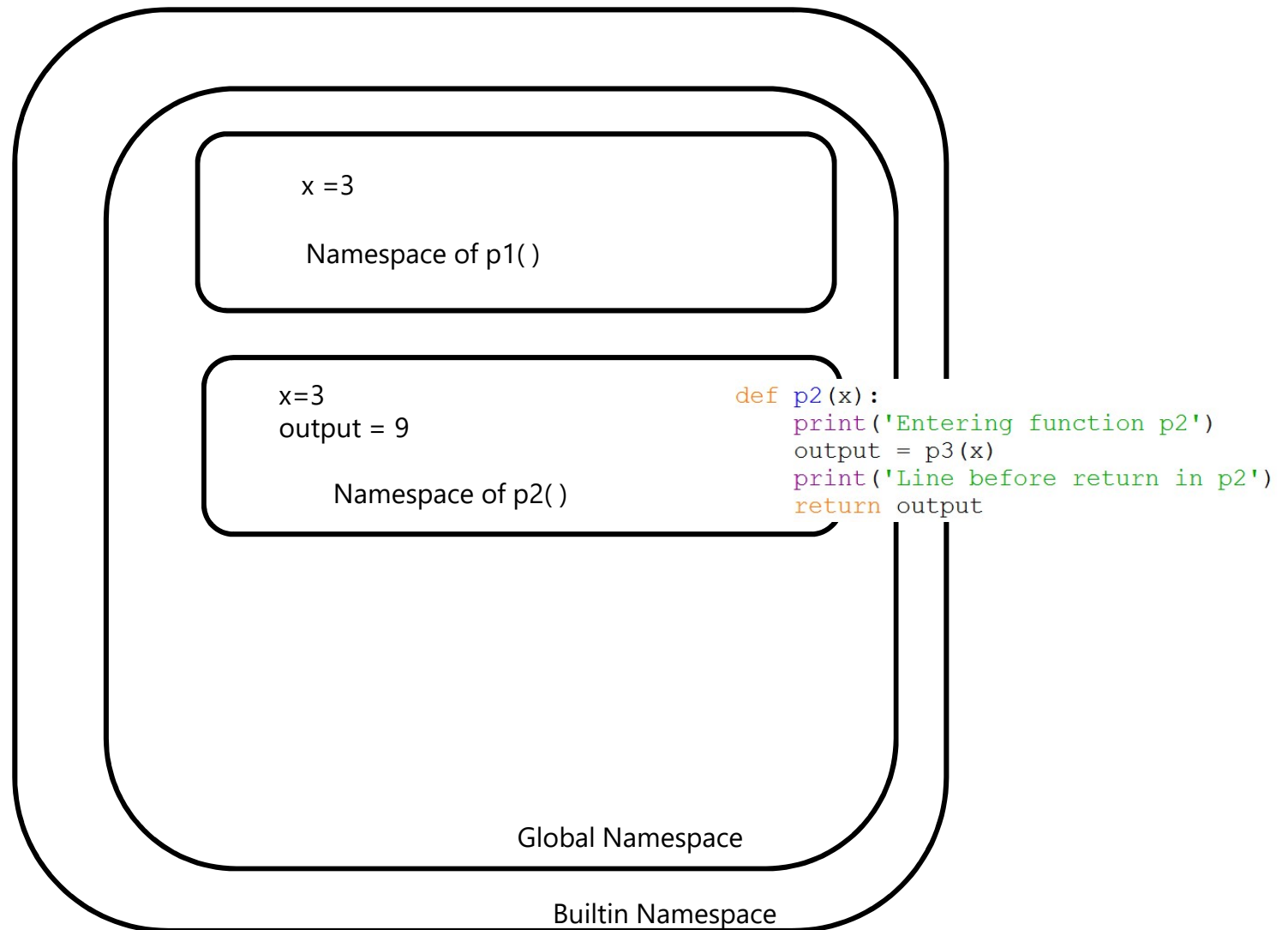
Namespaces: Calling Other Functions



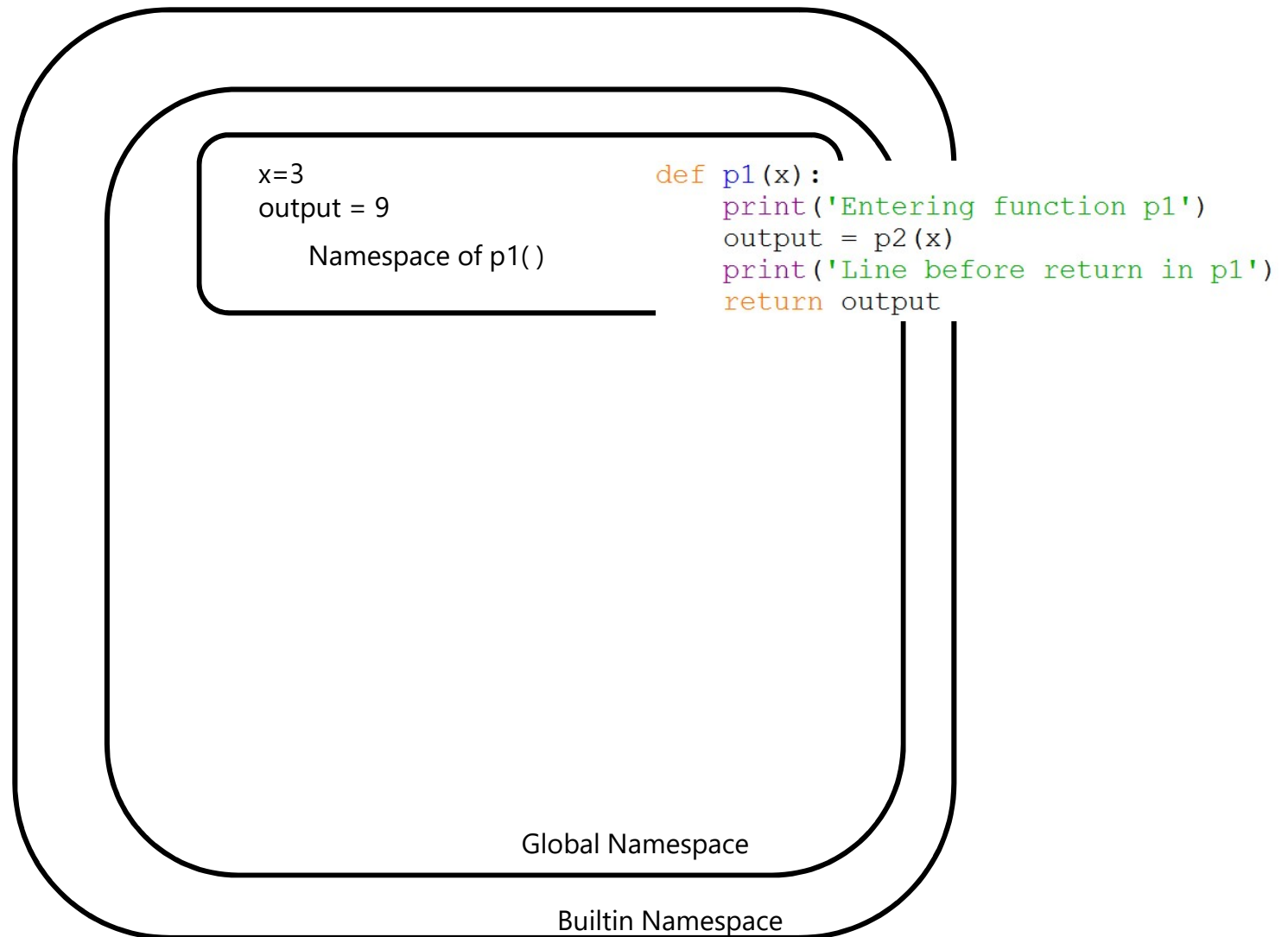
Namespaces: Calling Other Functions



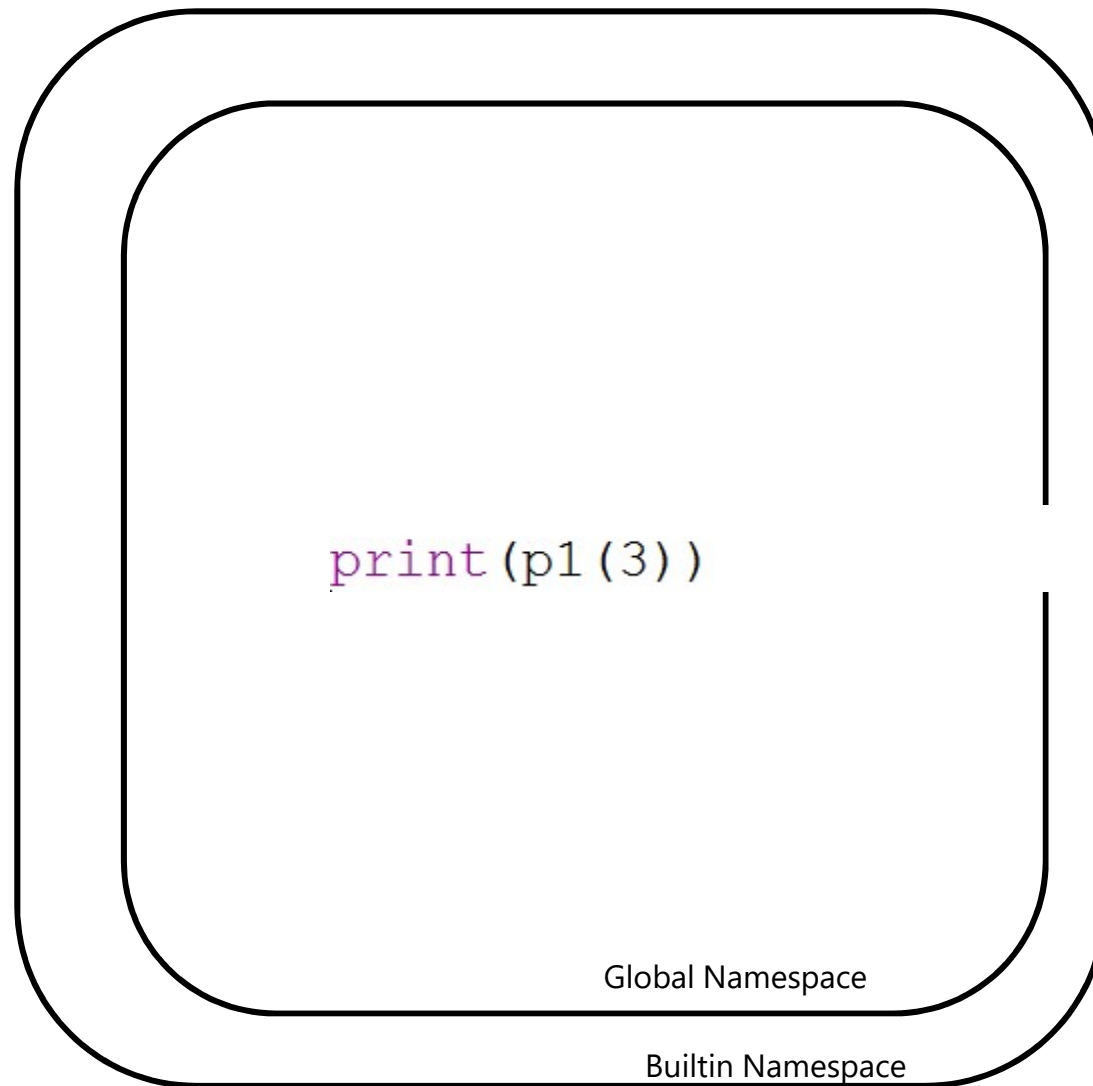
Namespaces: Calling Other Functions




Namespaces: Calling Other Functions



Namespaces: Calling Other Functions



 Debug Control

GoStepOverOutQuit

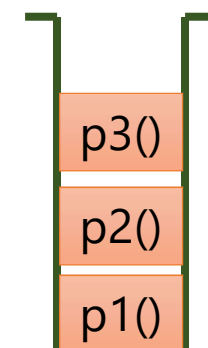
☒ Stack☐ Source☒ Locals☐ Globals

W03a Call Stack.py:16: p3()

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
'__main__'.<module>(), line 1: p1(3)
'__main__'.p1(), line 3: output = p2(x)
'__main__'.p2(), line 10: output = p3(x)
> '__main__'.p3(), line 16: output = x * x
```

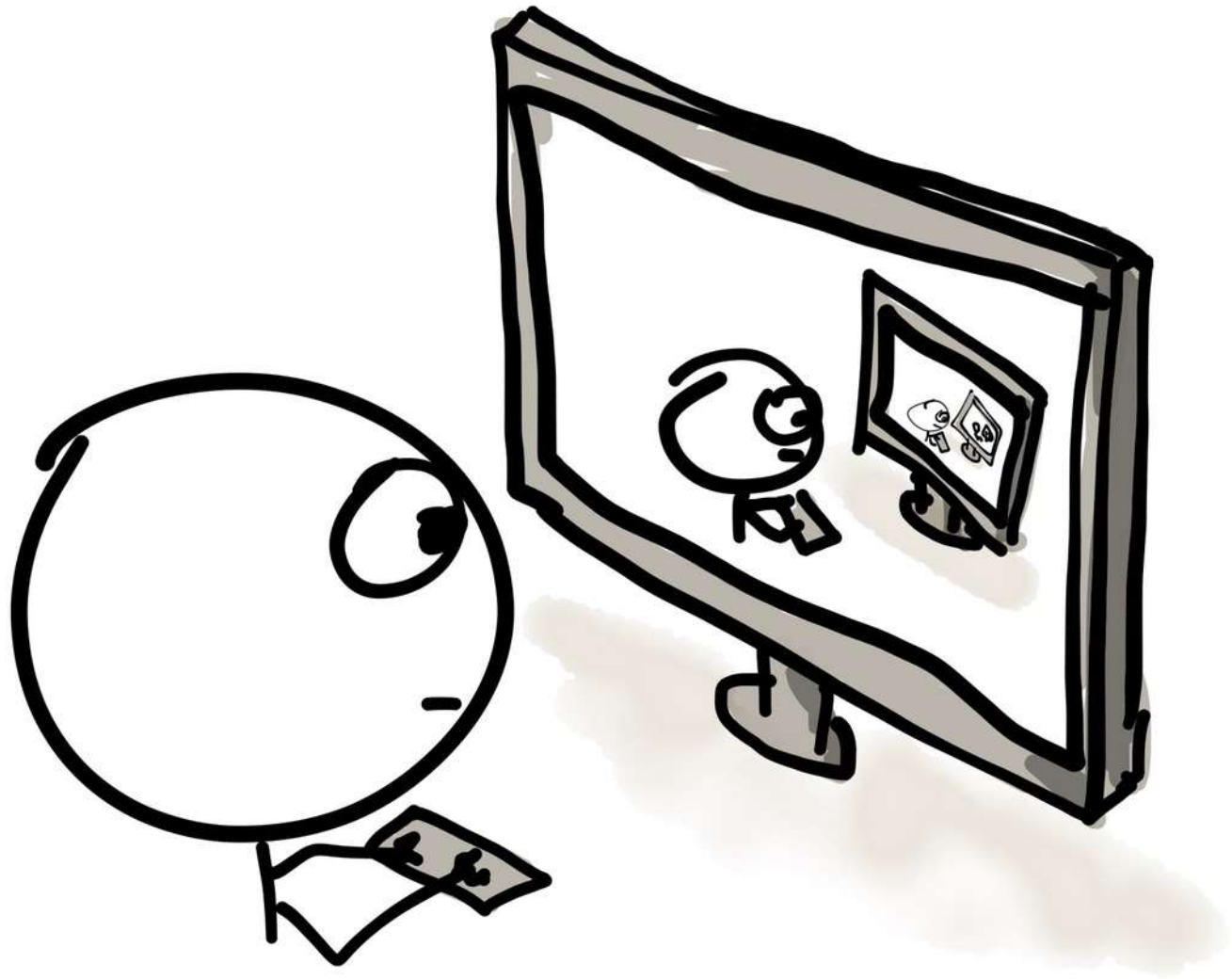
Locals

x 3



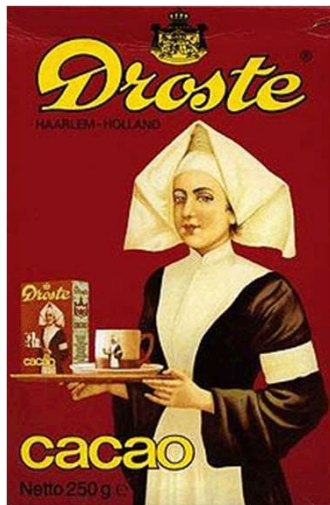
Recursion

Recursion

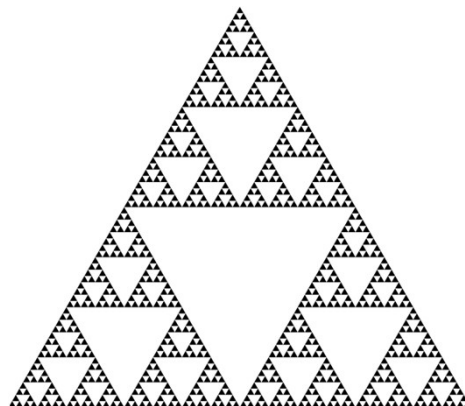


A Central Idea of CS

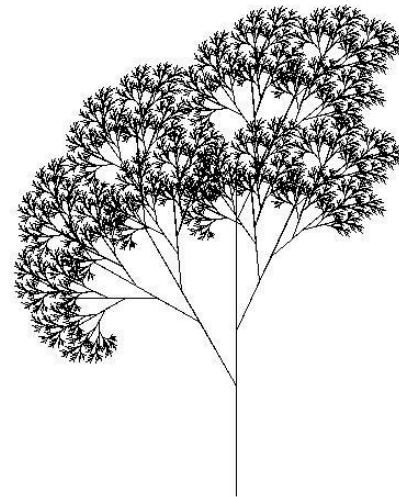
Some examples of recursion (inside and outside CS):



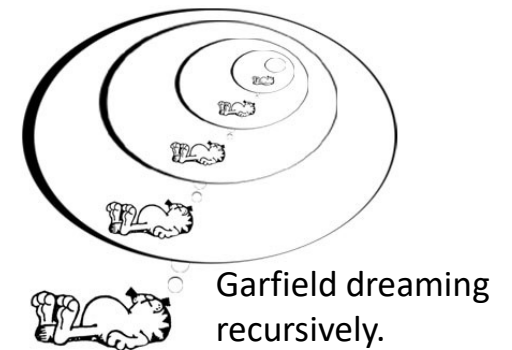
Droste effect



Sierpinski triangle



Recursive tree



[Mandelbrot Fractal Endless Zoom](#)

Recursion

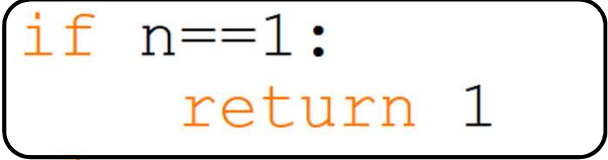
- A function that calls itself
- And extremely powerful technique
- Solve a big problem by solving a smaller version of itself
 - Mini-me



Recursive Functions


- Function calls itself

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



A rectangular box highlights the base case of the function: `if n==1: return 1`. An arrow points from this box to the text "Base Case" located above and to the right of the code.

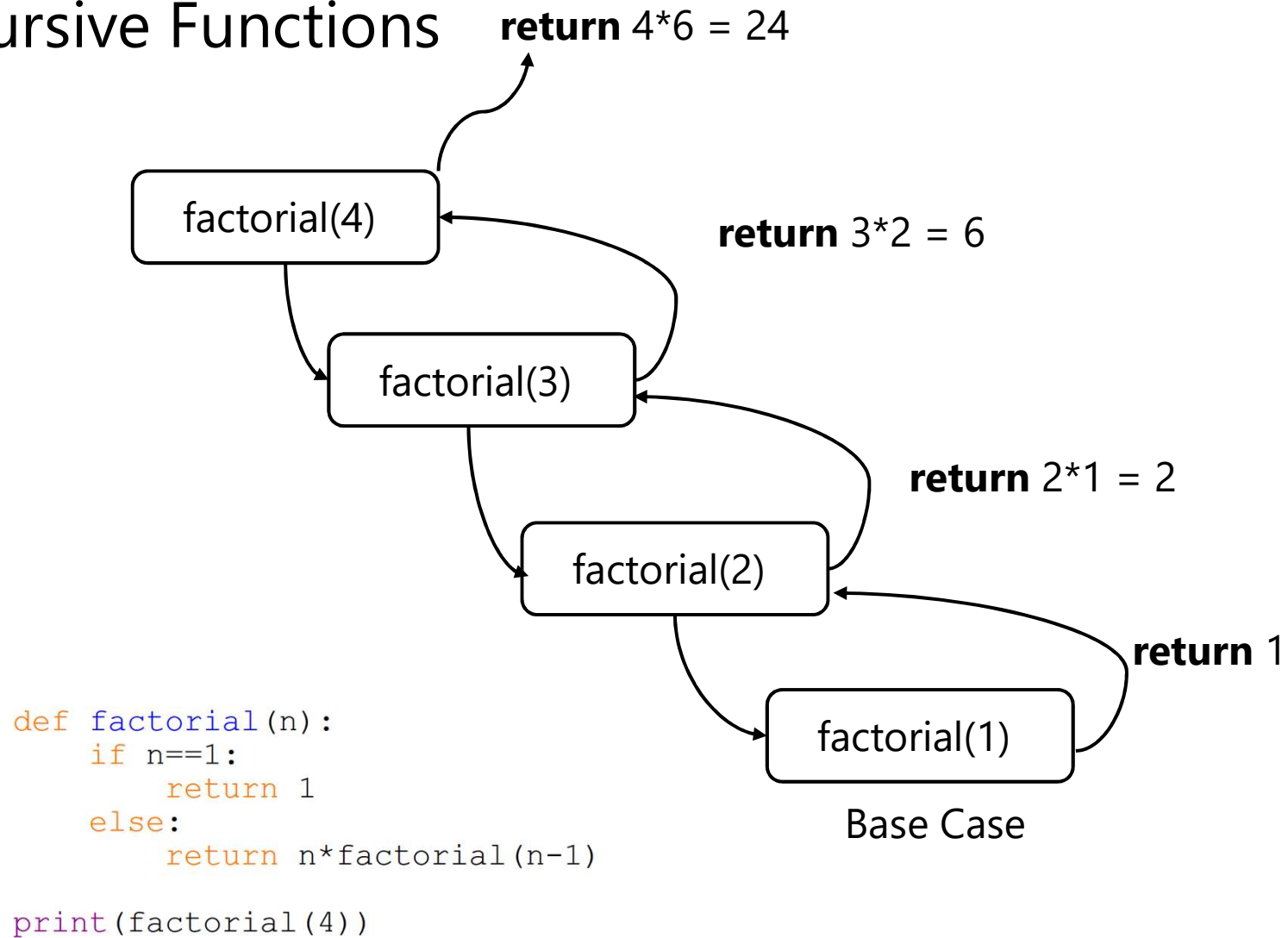
```
print(factorial(4))
```



An arrow points from the recursive call `factorial(n-1)` in the code to the text "Calling itself (smaller problem)" located below and to the right of the code.

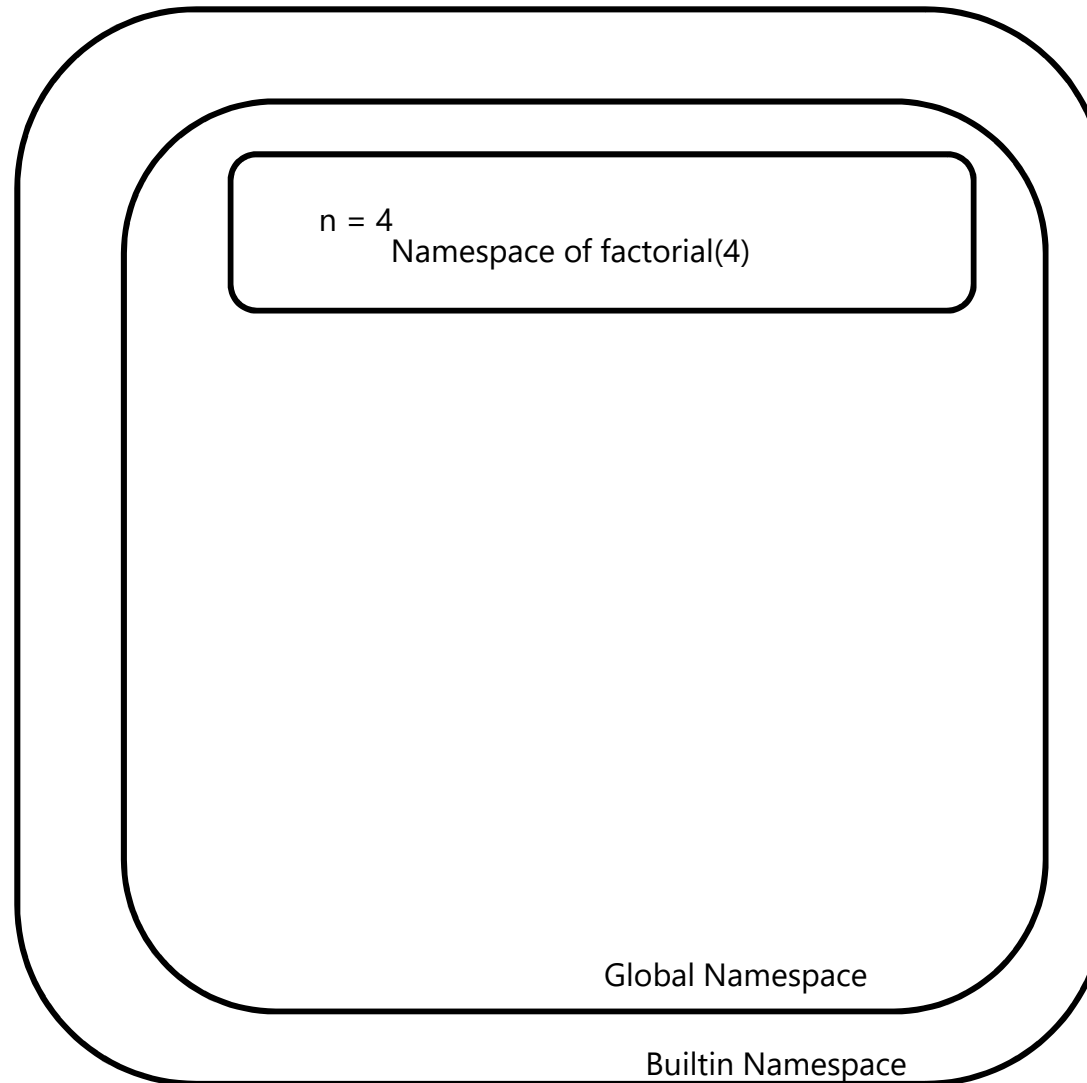
Calling itself
(smaller problem)

Recursive Functions



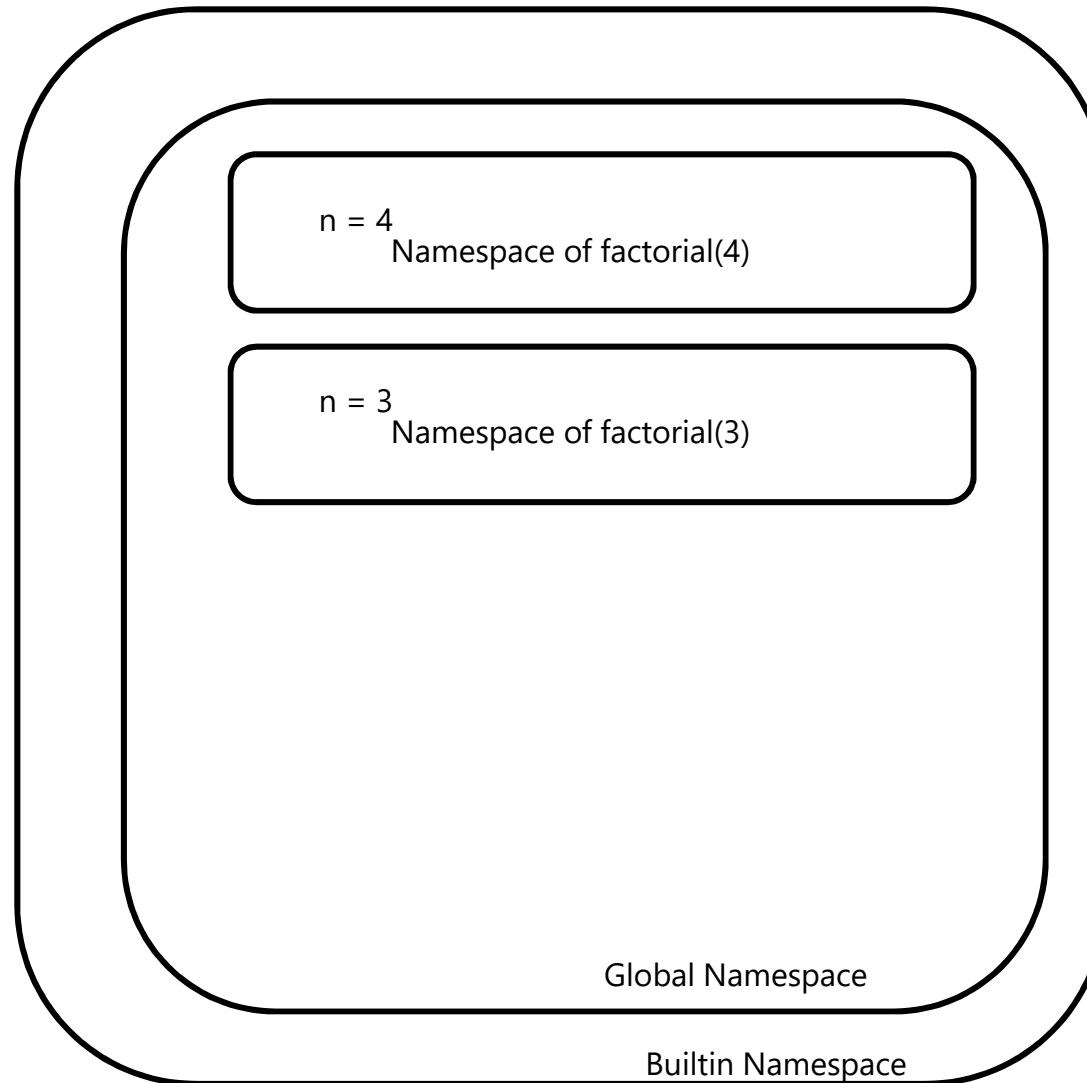
Namespaces: Recursive Function

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



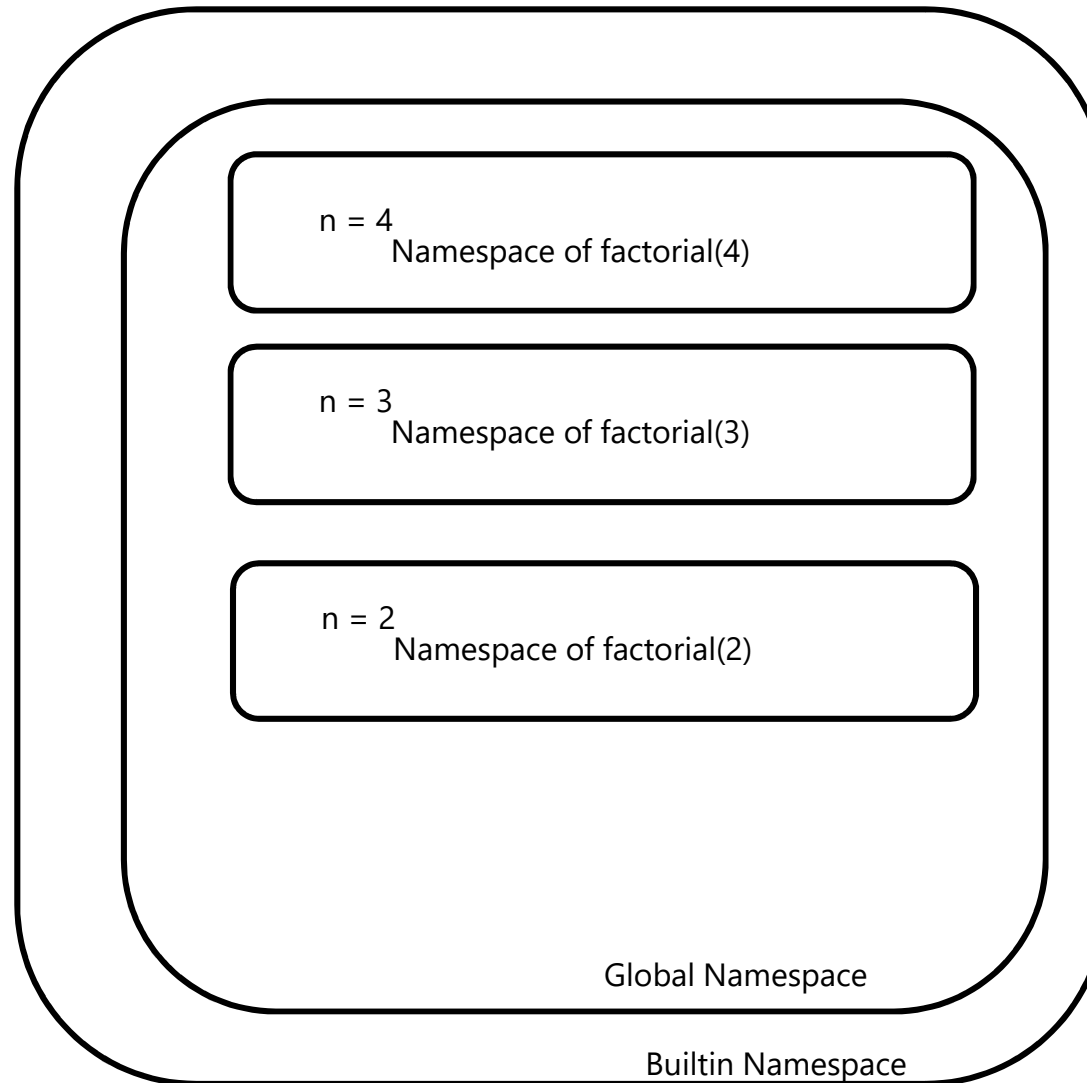
Namespaces: Recursive Function

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



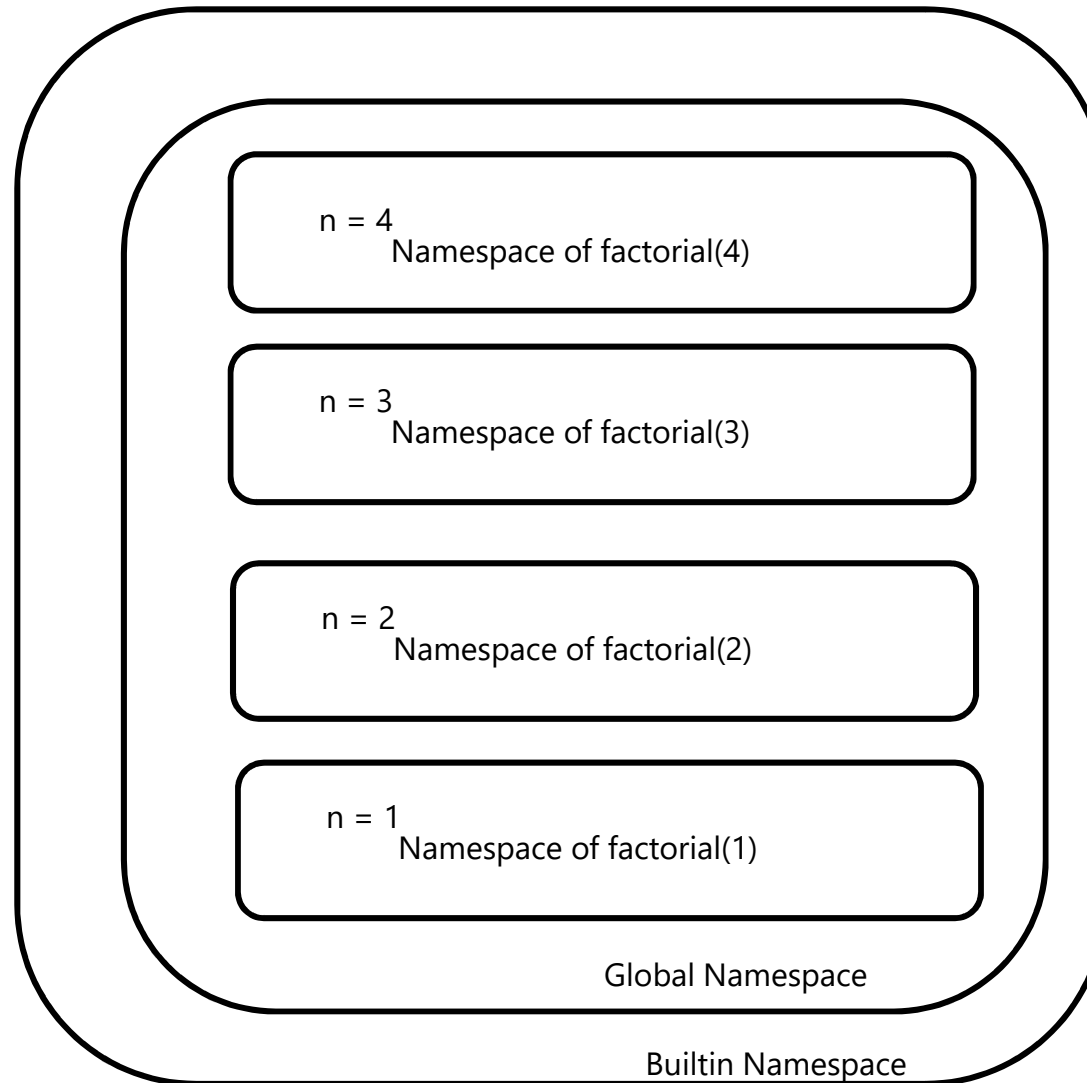
Namespaces: Recursive Function

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



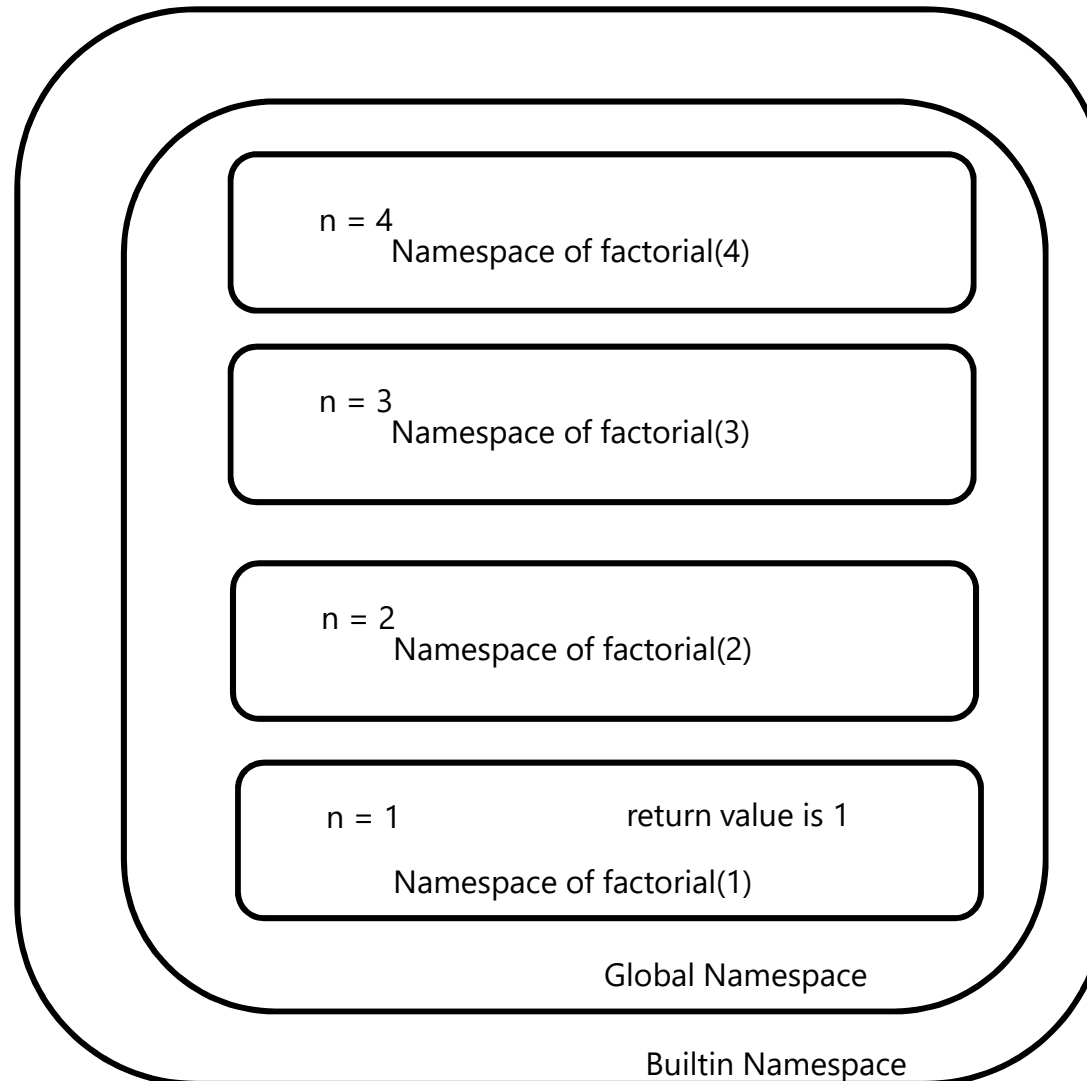
Namespaces: Recursive Function

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



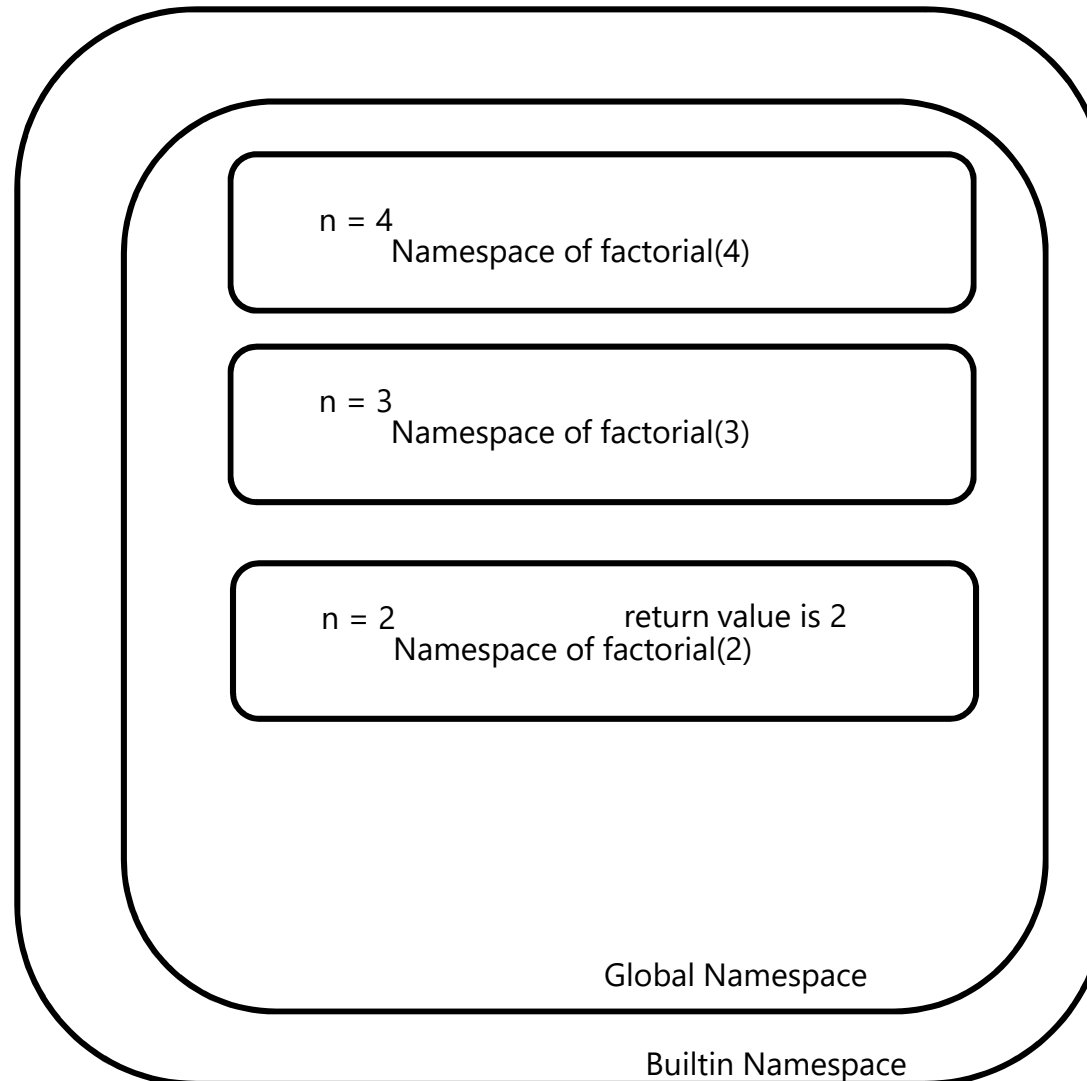
Namespaces: Recursive Function

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



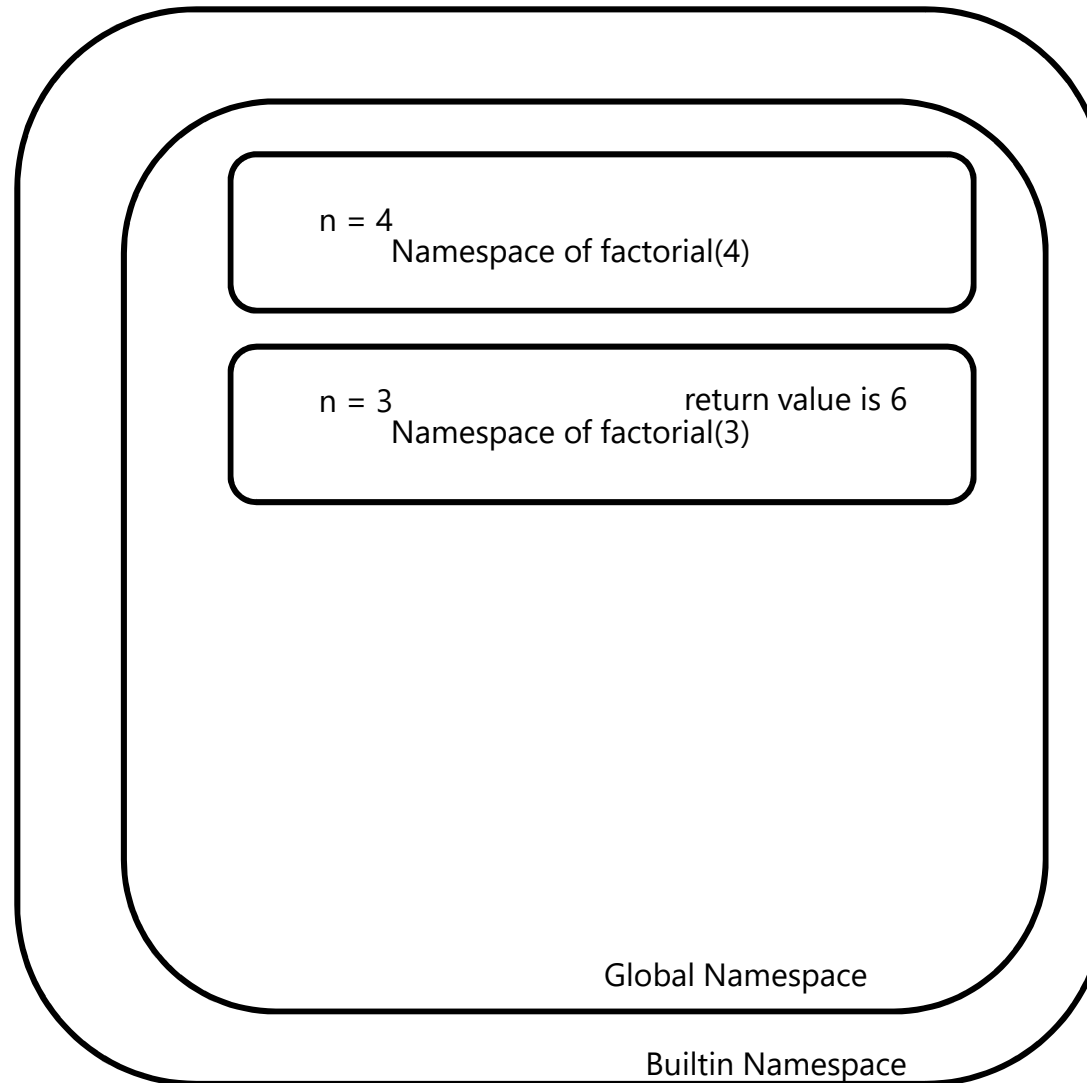
Namespaces: Recursive Function

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



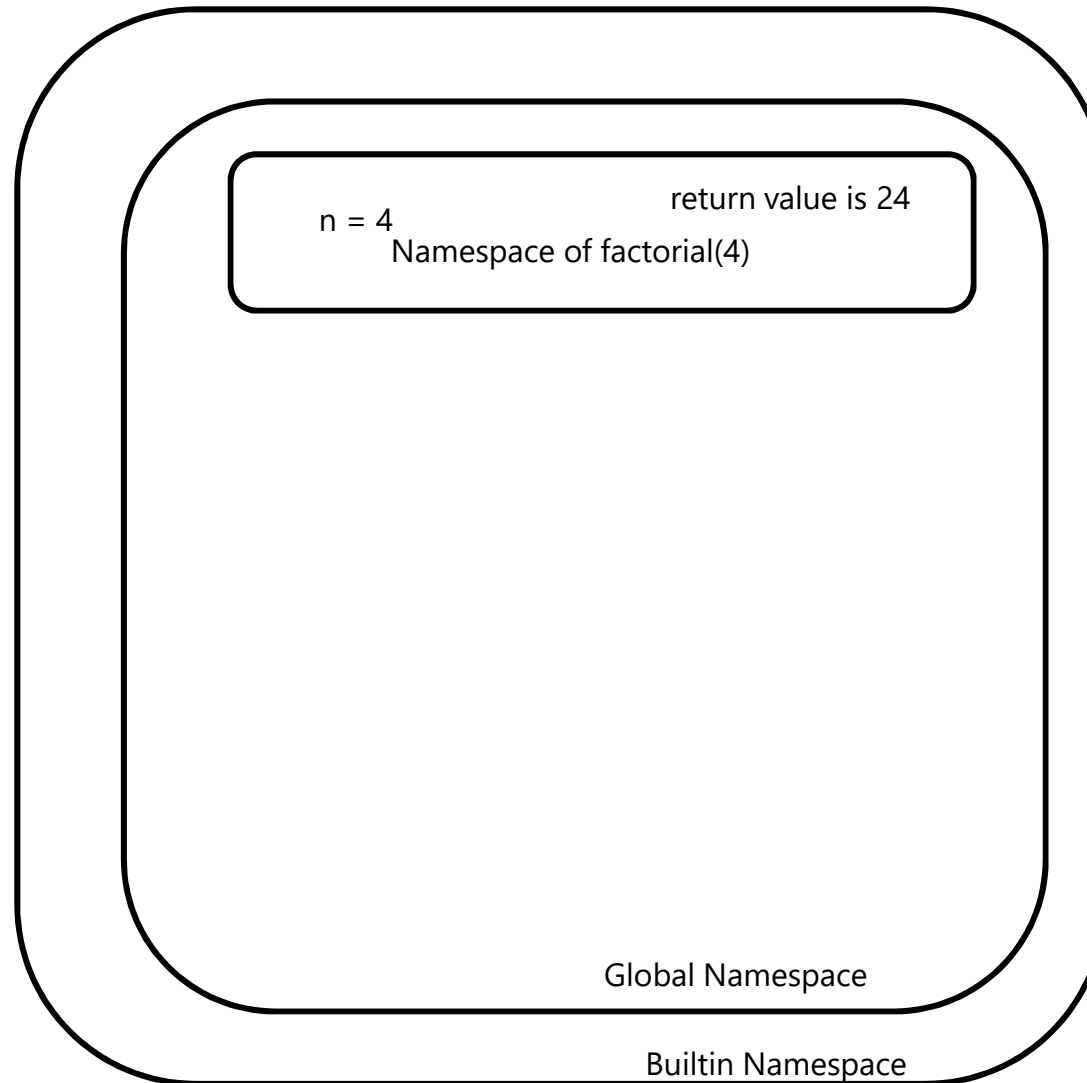
Namespaces: Recursive Function

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

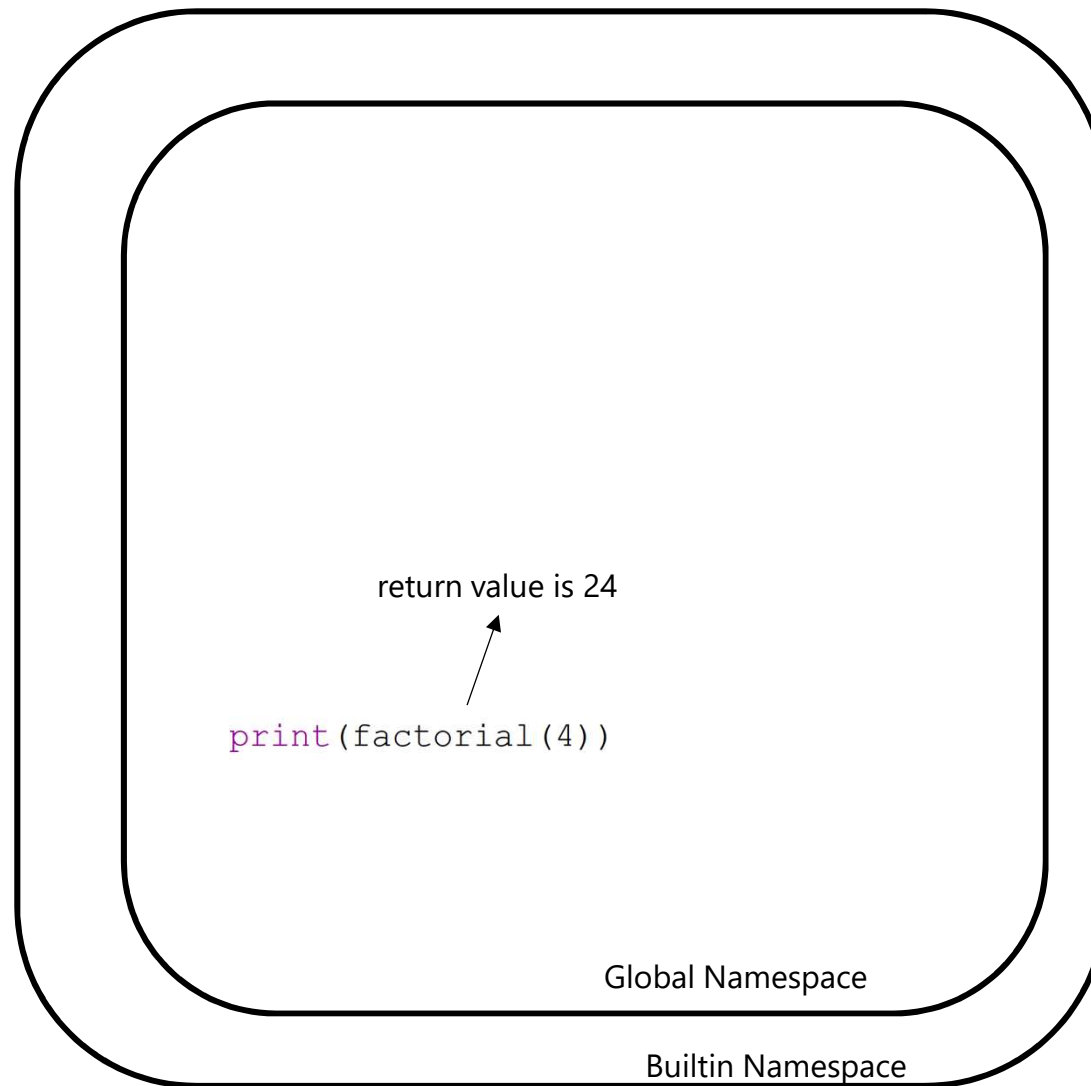


Namespaces: Recursive Function

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



Namespaces: Recursive Function



Recursive Vs Iteration

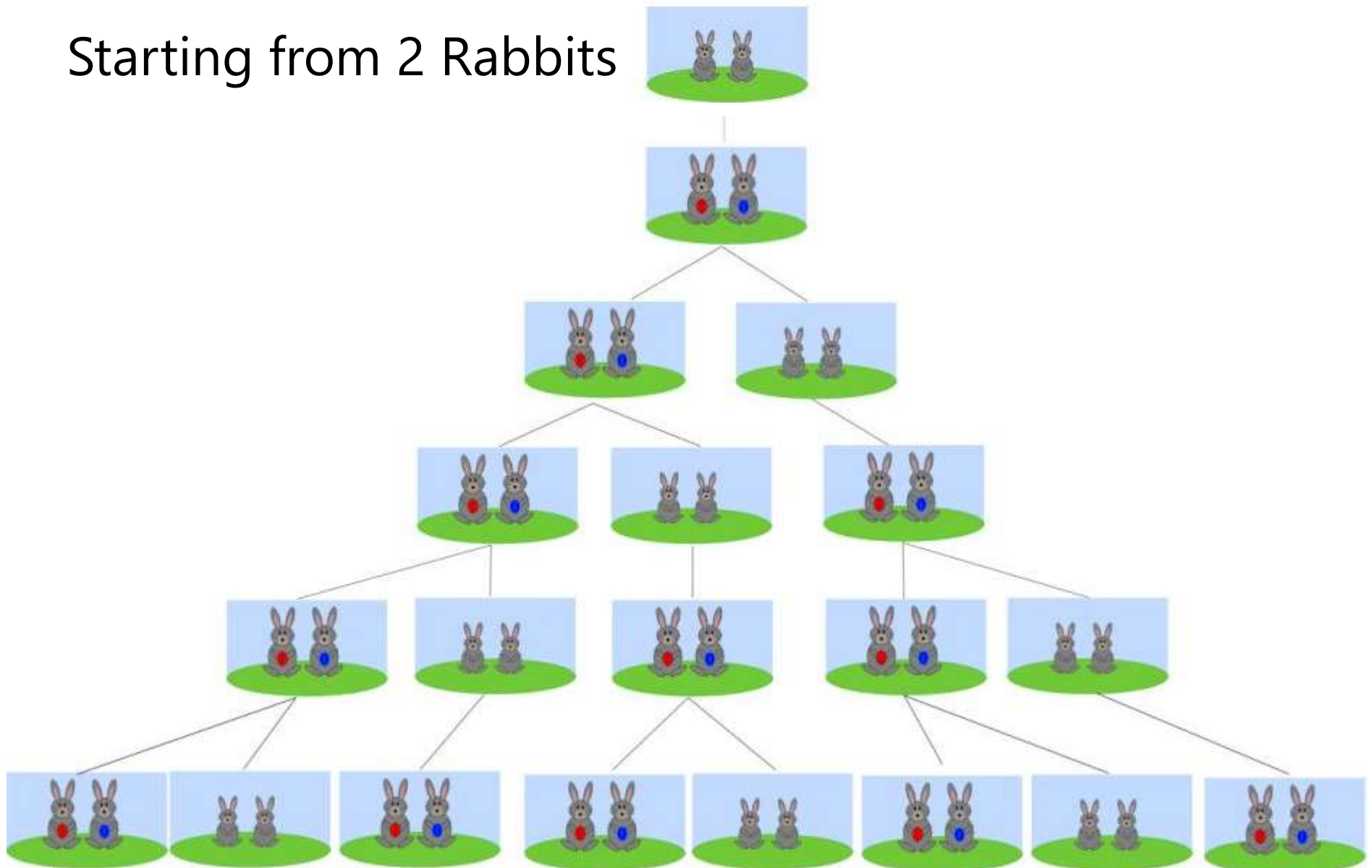
```
def factorial(n):  
    if n==1:  
        return 1  
    else:  
        return n*factorial(n-1)
```

```
def factorial_iter(n):  
    fact = 1  
    for elem in range(1,n+1):  
        fact = fact*elem  
    return fact
```

Fibonacci Number

(Recursion)

Starting from 2 Rabbits



How many ways to arrange cars?

- Let's say we have two types of vehicles, cars and buses



- And each car can park into one parking space, but a bus needs two consecutive ones
- If we have 1 parking space, I can only park a car



1 way

- But if there are 2 parking spaces, we can either park a bus or two cars



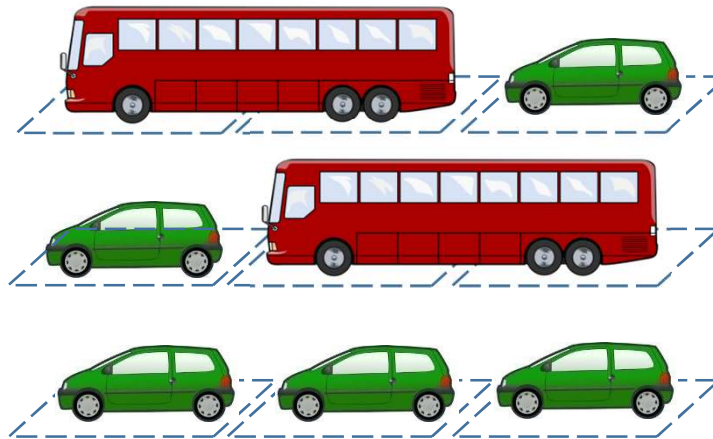
2 ways

-



How many ways to arrange cars?

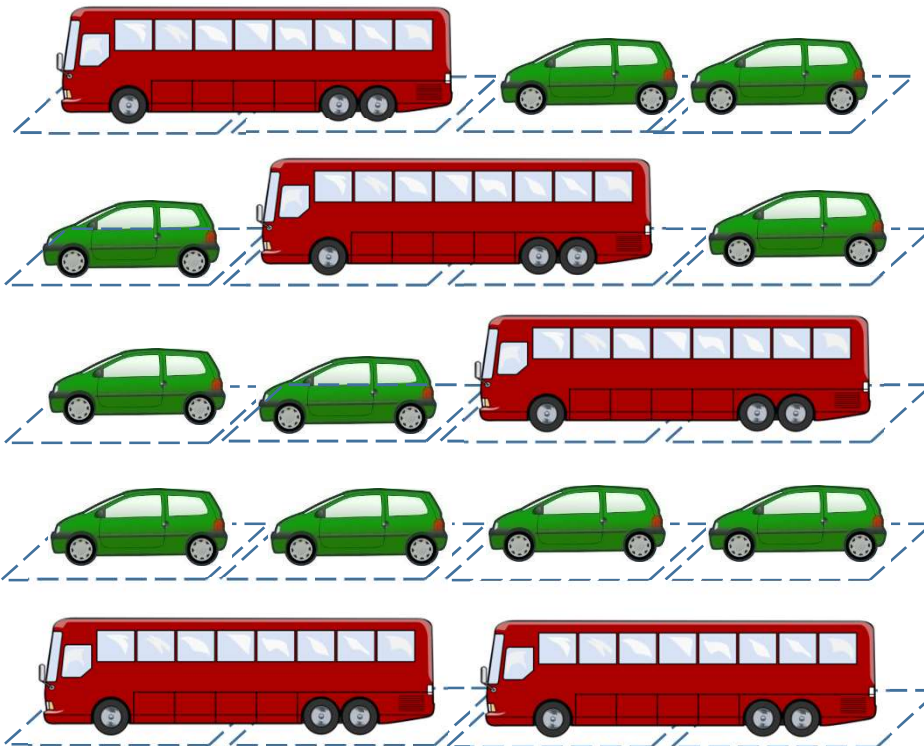
- So if we have 3 parking spaces, how many different ways can we park cars and buses?



3 ways

How many ways to arrange cars?

- So if we have 4 parking spaces, how many different ways can we park cars and buses?



5 ways

How many ways to arrange cars?

- 5 parking spaces?
- 6 parking spaces?

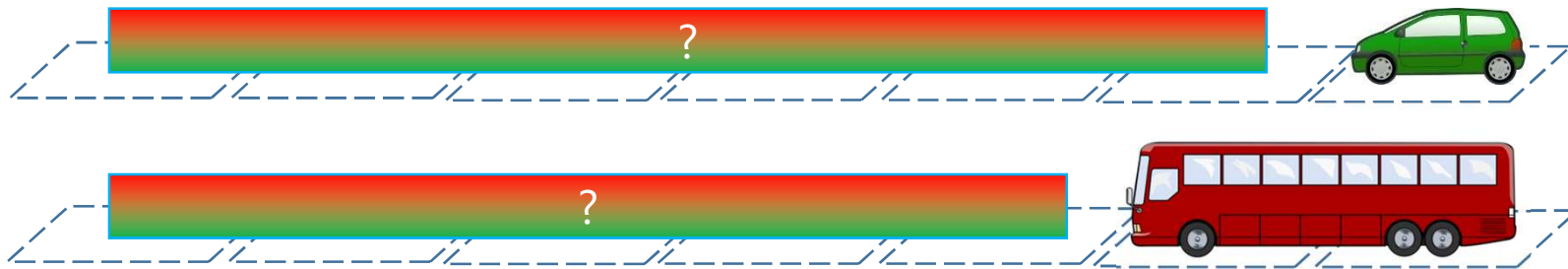


#parking spaces	#ways
0	1
1	1
2	2
3	3
4	5
5	8
6	13

- Can you figured out THE pattern?
 - 1, 1, 2, 3, 5, 8, 13, ...
 - What is the next number?

How many ways to arrange cars?

- In general, if we have n parking spaces, how many ways can we park the vehicles?
- You can think backward, the last parking space can be either a car or a bus

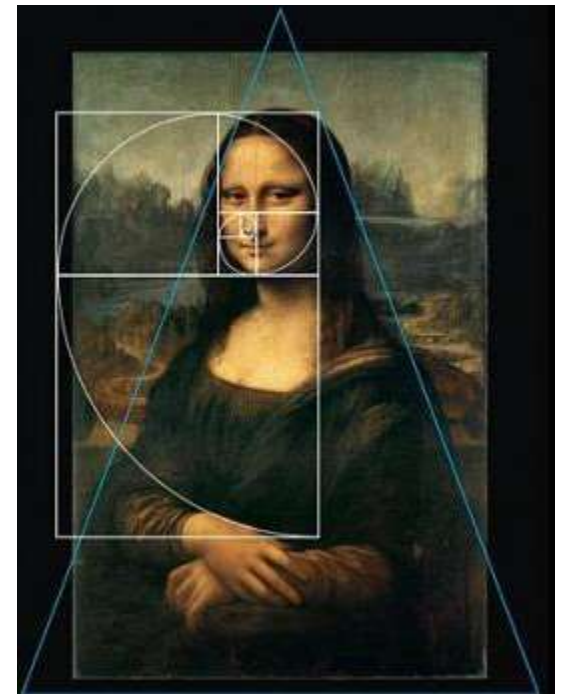
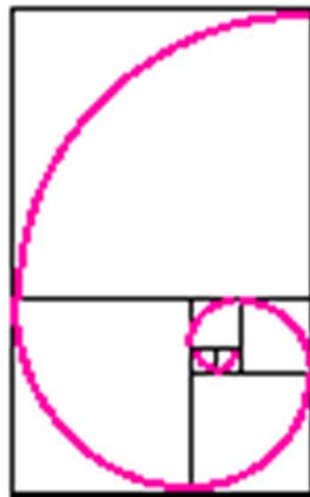
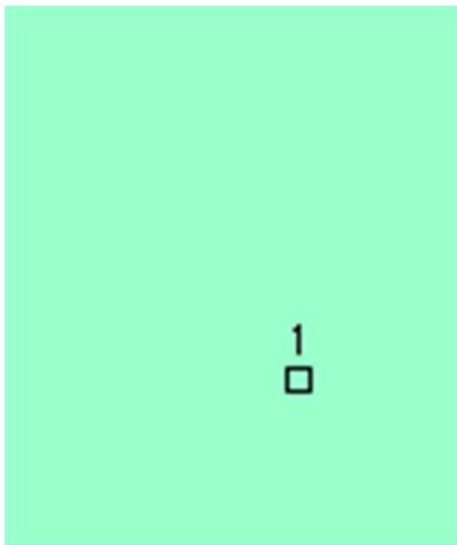
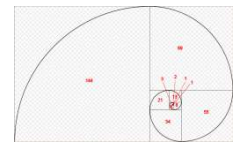


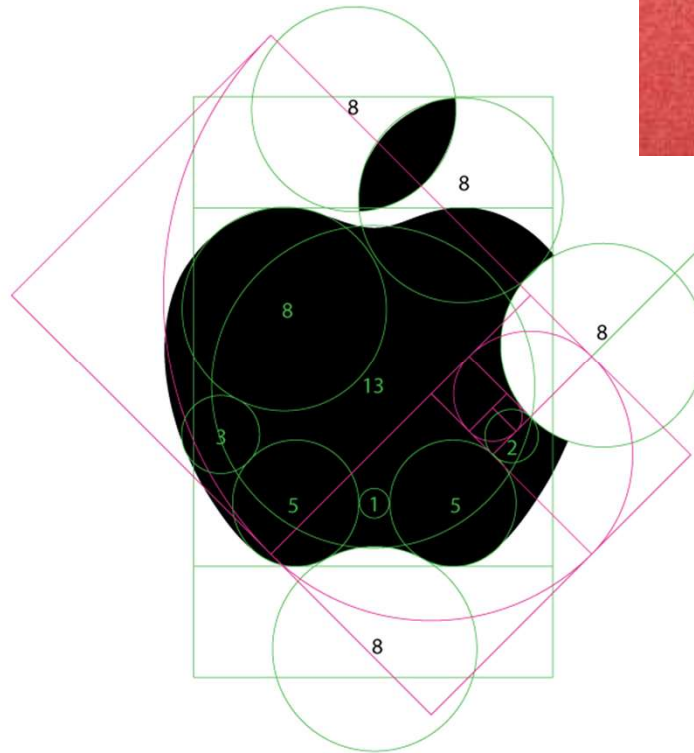
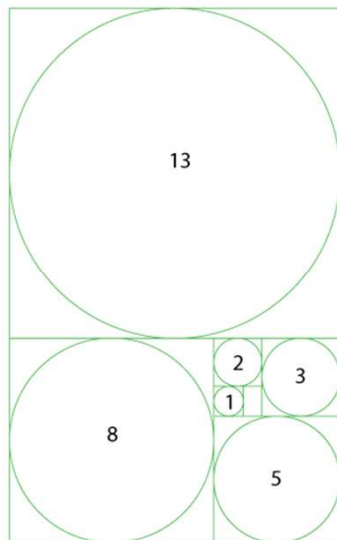
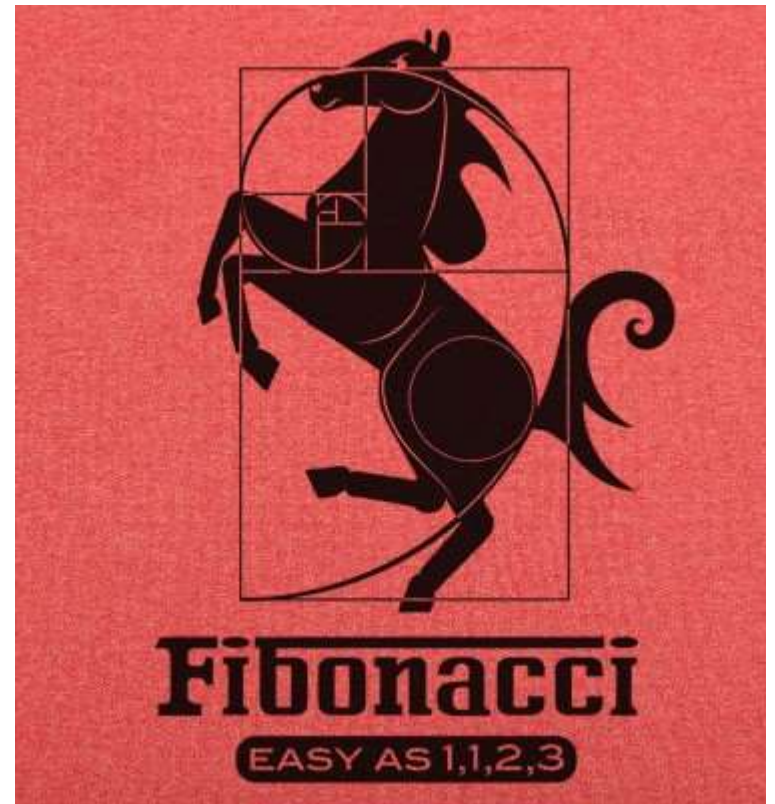
- If it's a car, there are $n - 1$ spaces left, you can have the number of way for $n - 1$ spaces
 - Otherwise, it's the number of way for $n - 2$ spaces
- So

$$f(n) = f(n - 1) + f(n - 2) \quad \text{for } f(0) = f(1) = 1$$

Fibonacci Numbers

- Fibonacci numbers are found in nature (sea-shells, sunflowers, etc)
- <http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html>






```
def fibonacci(n):  
    if n == 1 or n == 0:  
        return 1  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```

```
>>> fibonacci(10)
```

```
89
```

```
>>> fibonacci(20)
```

```
10946
```

```
>>> fibonacci(994)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#16>", line 1, in <module>  
    fibonacci(994)
```

```
File "<pyshell#5>", line 5, in fibonacci  
    return fibonacci(n-1) + fibonacci(n-2)
```

```
File "<pyshell#5>", line 5, in fibonacci  
    return fibonacci(n-1) + fibonacci(n-2)
```

```
File "<pyshell#5>", line 5, in fibonacci  
    return fibonacci(n-1) + fibonacci(n-2)
```

```
[Previous line repeated 989 more times]
```

```
File "<pyshell#5>", line 2, in fibonacci  
    if n == 1 or n == 0:
```

```
RecursionError: maximum recursion depth exceeded in comparison
```

```
>>> fibonacci(50)
```



Challenge

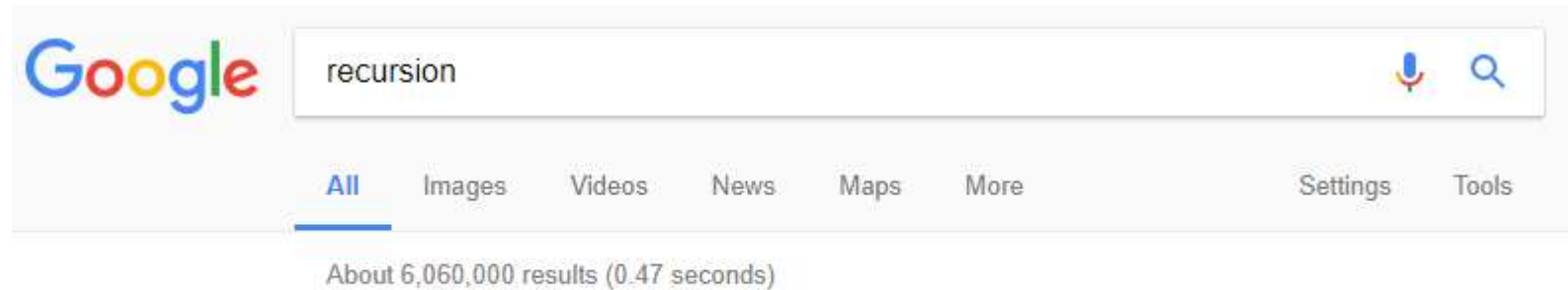
- Write a fibonacci function that can compute $f(n)$ for $n > 1000$

```
>>> fibonacci(1000)
70330367711422815821835254877183549770181269836358732742604905087154537118196933
57974224949456261173348775044924176599108818636326545022364710601205337412127386
7339111198139373125598767690091902245245323403501
>>> fibonacci(2000)
68357022595758066470453965491705801070554080293655245654075533677980824544080540
14954534318953113802726603726769523447478238192192714526677939943338306101405105
41481970566409090181363729645376709552810486826470491443352935557914873104468563
41354877358979546298425169471014942535758696998934009765395457402148198191519520
85089538422954565146720383752121972115725761141759114990448978941370030912401573
418221496592822626
```



**TAKE THE
CHALLENGE**

Google about Recursion



- Did you mean: *recursion*
 - Do a barrel roll
 - Askew
 - Anagram
 - Google in 1998
 - Zerg rush
- More in [Google Easter Eggs](#)