# CCPS109
# Computer Science I
# L3

**Lecturer: Nhan Tran**

**n3tran@Ryerson.ca**

# Agenda

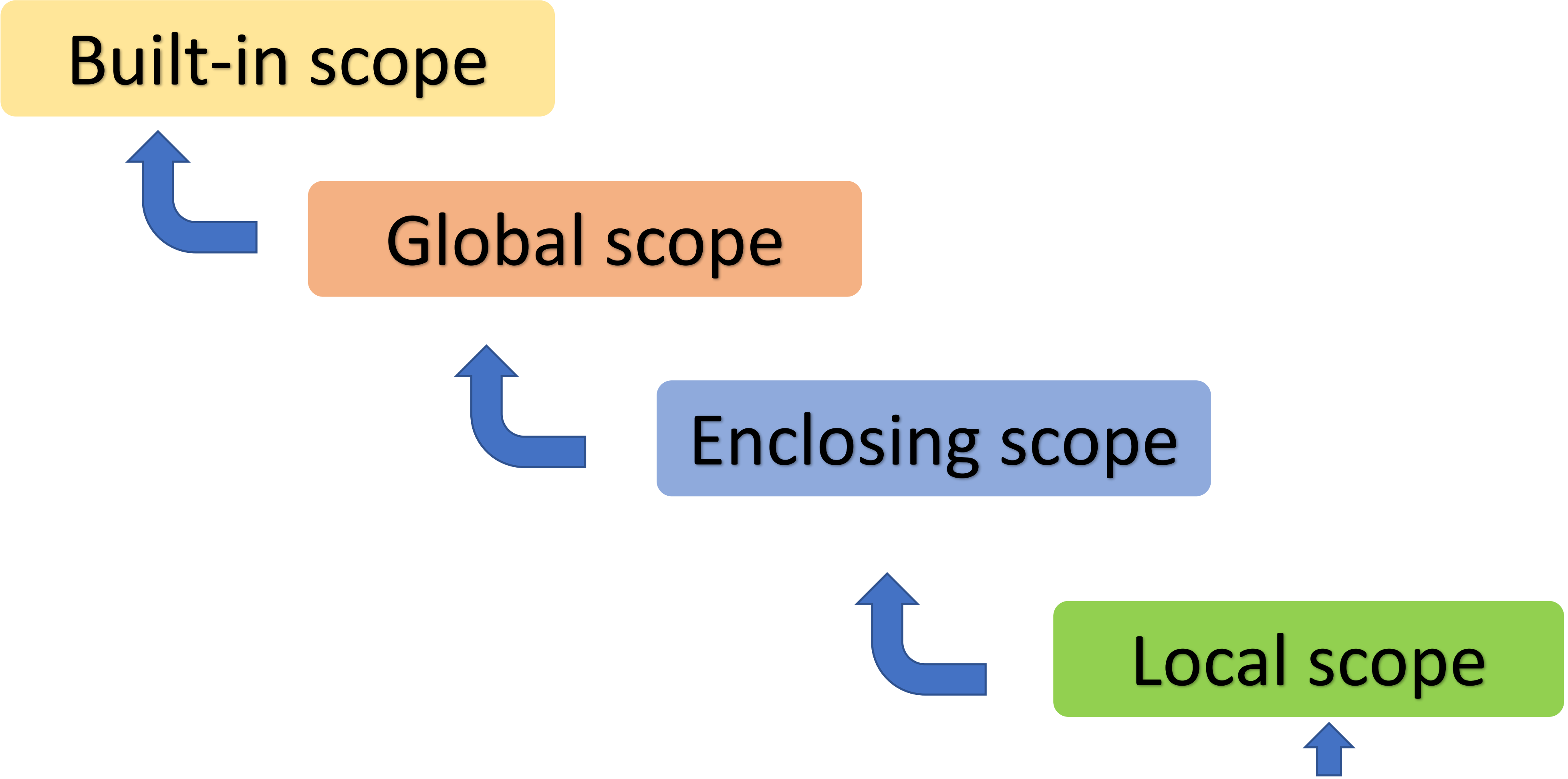
**Announcement**

**Lab 2 tomorrow!**

## Lecture:

- more scope
- more functions
- recursion
- module/ codes in different files

# Scope continue



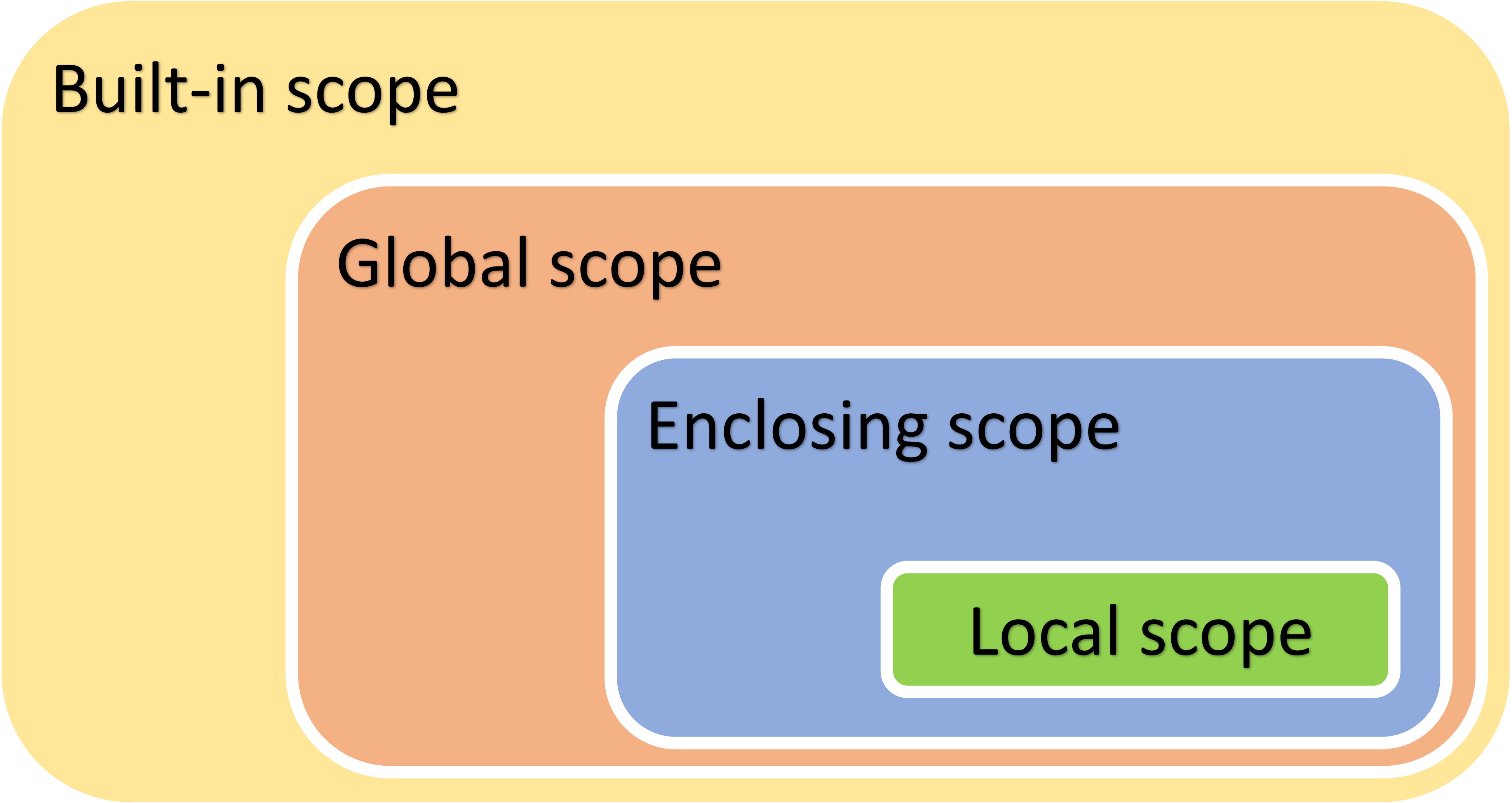
Built-in scope

Global scope

Enclosing scope

Local scope

# Scope continue



Built-in scope

Global scope

Enclosing scope

Local scope

# Scope

inside a function: **can access** variables defined on outside

inside a function: **cannot modify** a variable defined outside

however: Can use **“global” variables**, but not good coding style

# SCOPE EXAMPLE

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

different x objects

# SCOPE EXAMPLE

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

```
def g(y):
    print(x)
    print(x + 1)

x = 5
g(x)
print(x)
```

*x* from outside *g*

*x* inside *g* is picked up from scope that called function *g*

# SCOPE EXAMPLE

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

```
def g(y):
    print(x)
    print(x + 1)

x = 5
g(x)
print(x)
```

```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

UnboundLocalError: local variable 'x' referenced before assignment

# SCOPE EXAMPLE

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

different x objects

```
def g(y):
    print(x)
    print(x + 1)

x = 5
g(x)
print(x)
```

x from outside g

x inside g is picked up from scope that called function g

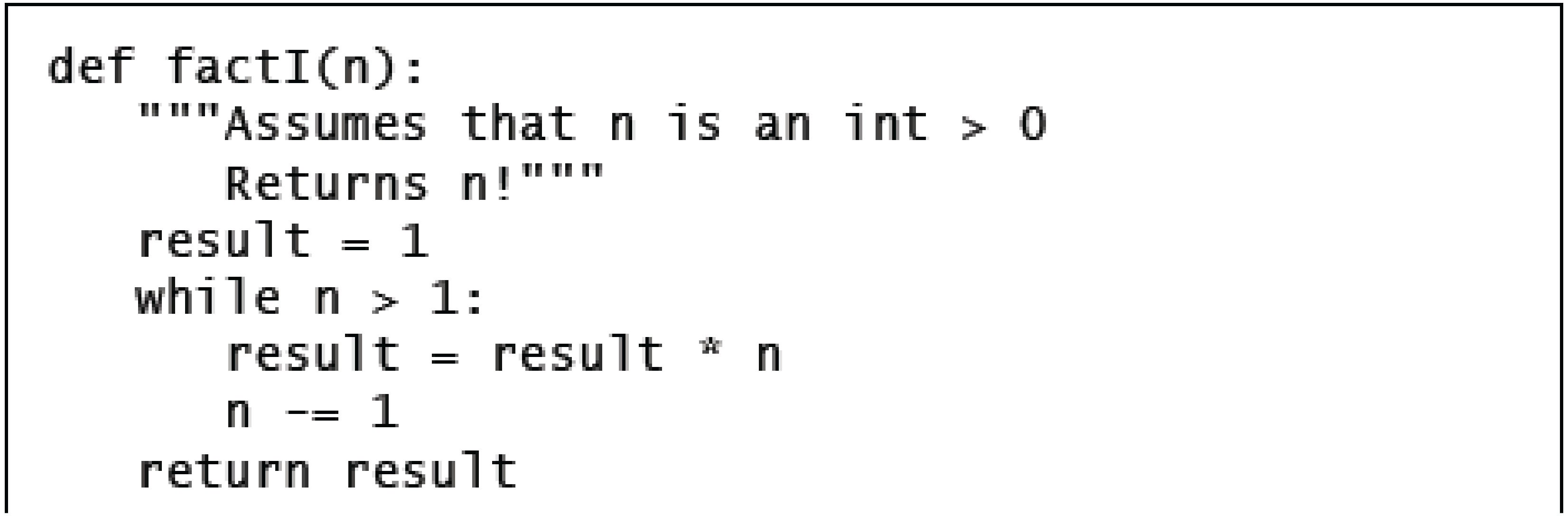
```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

UnboundLocalError: local variable 'x' referenced before assignment

# Write the following function:

$$=\prod_{1}^{n} n$$

# ITERATION



```
def factI(n):
    """Assumes that n is an int > 0
       Returns n!"""
    result = 1
    while n > 1:
        result = result * n
        n -= 1
    return result
```

This iteration will terminate when n is reduced to 1

# Factorial

$$n! = \prod_{1}^{n} n$$

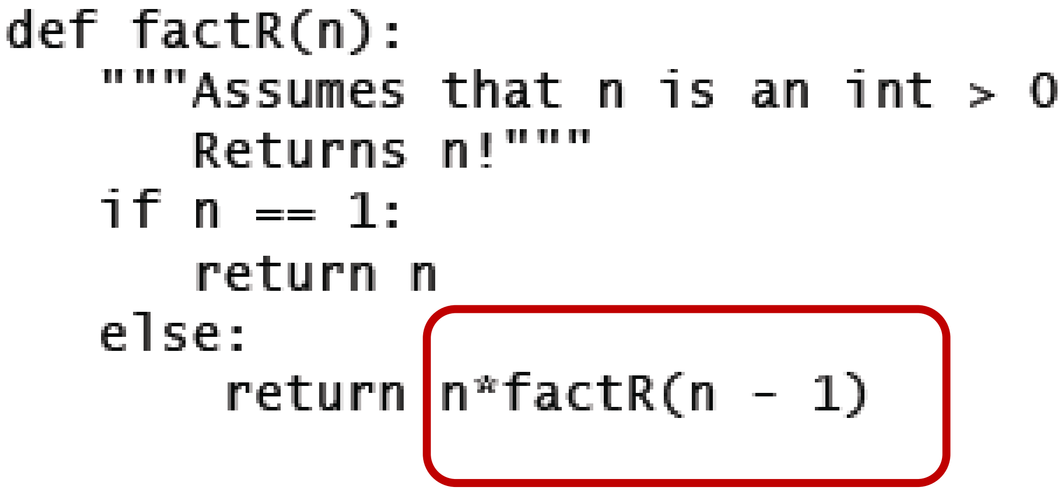
# Recursion

Algorithm uses recursion when a function called, itself or another function, over and over.

Recursive function: when a function calls itself

Mutually recursive : Fa()=>Fb()=>Fa()…

# RECURSION

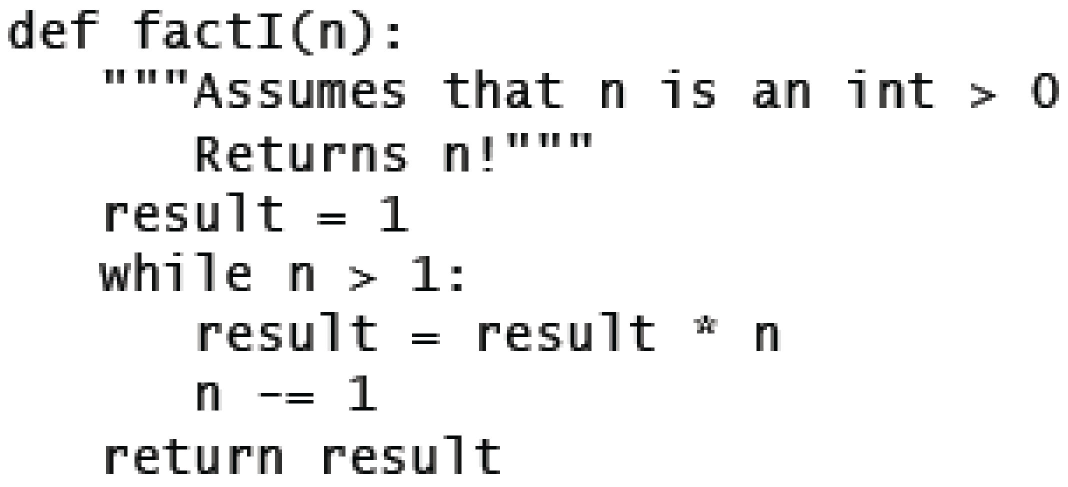


```
def factR(n):
   """Assumes that n is an int > 0
      Returns n!"""
   if n == 1:
      return n
   else:
       return n*factR(n - 1)
```

This recursion will terminate when n is reduced to 1

# ITERATION VS RECURSION

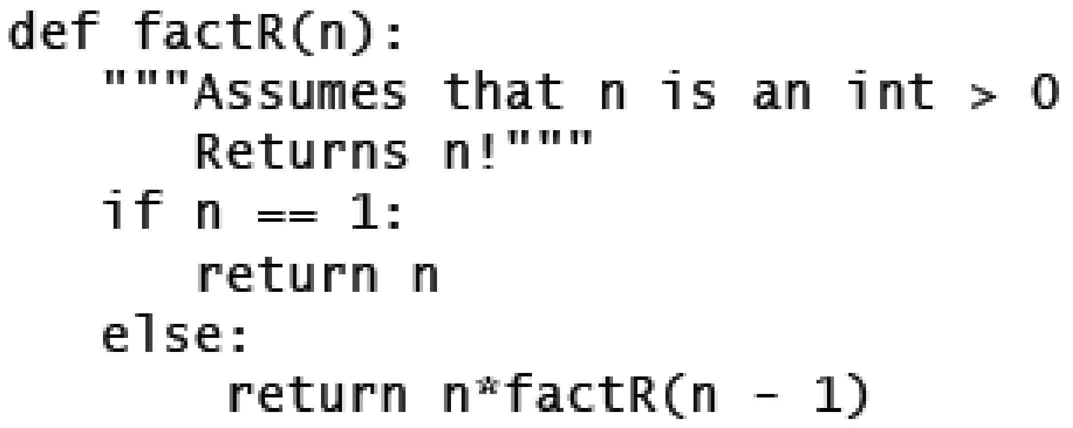
iteration



```
def factI(n):
   """Assumes that n is an int > 0
      Returns n!"""
   result = 1
   while n > 1:
      result = result * n
      n -= 1
   return result
```

This iteration will terminate when n is reduced to 1

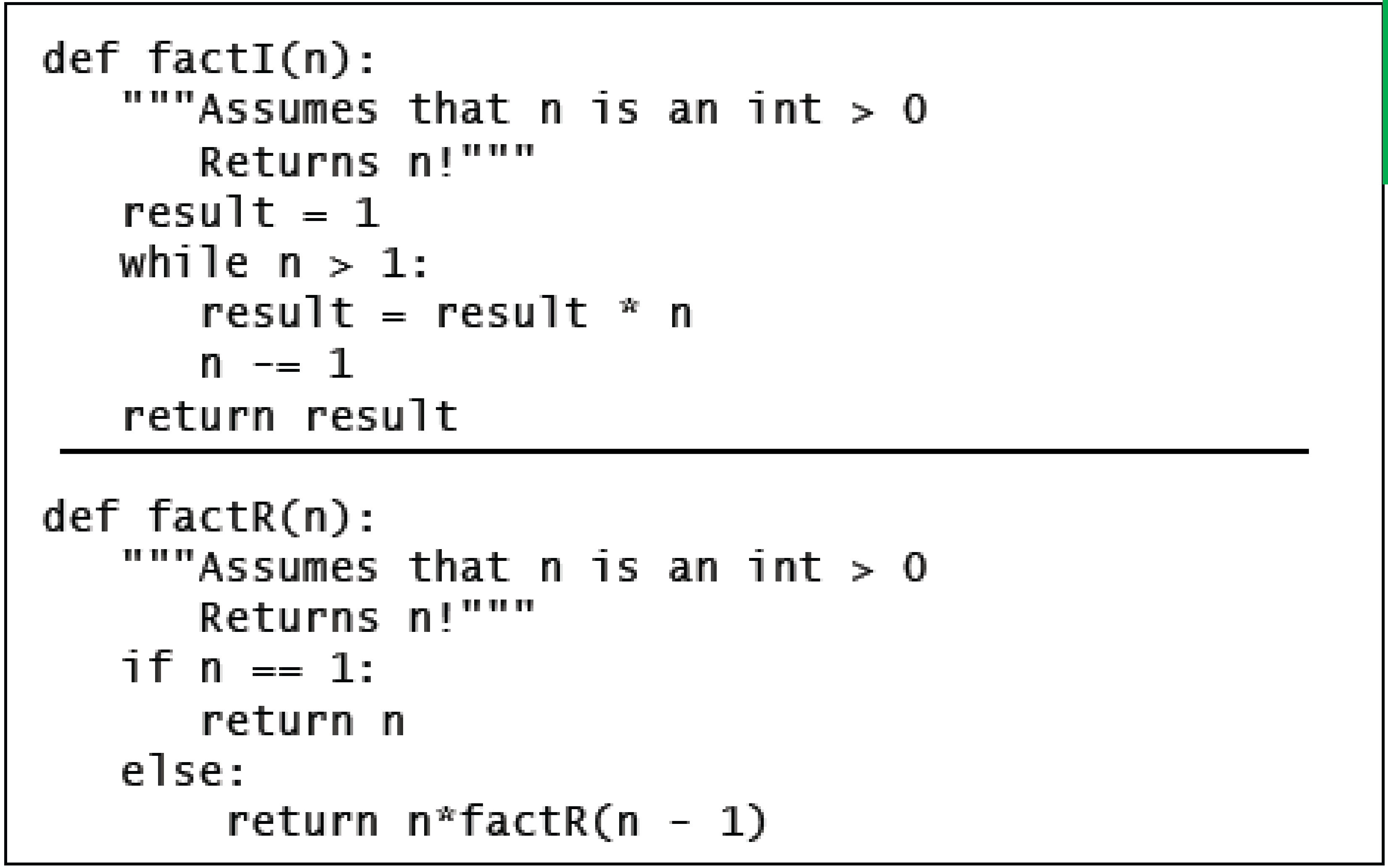
recursion



```
def factR(n):
   """Assumes that n is an int > 0
      Returns n!"""
   if n == 1:
      return n
   else:
       return n*factR(n - 1)
```

This recursion will terminate when n is reduced to 1

# ITERATION VS  RECURSION



```
def factI(n):
    """Assumes that n is an int > 0
       Returns n!"""
    result = 1
    while n > 1:
       result = result * n
       n -= 1
    return result
```

```
def factR(n):
    """Assumes that n is an int > 0
       Returns n!"""
    if n == 1:
       return n
    else:
        return n*factR(n - 1)
```

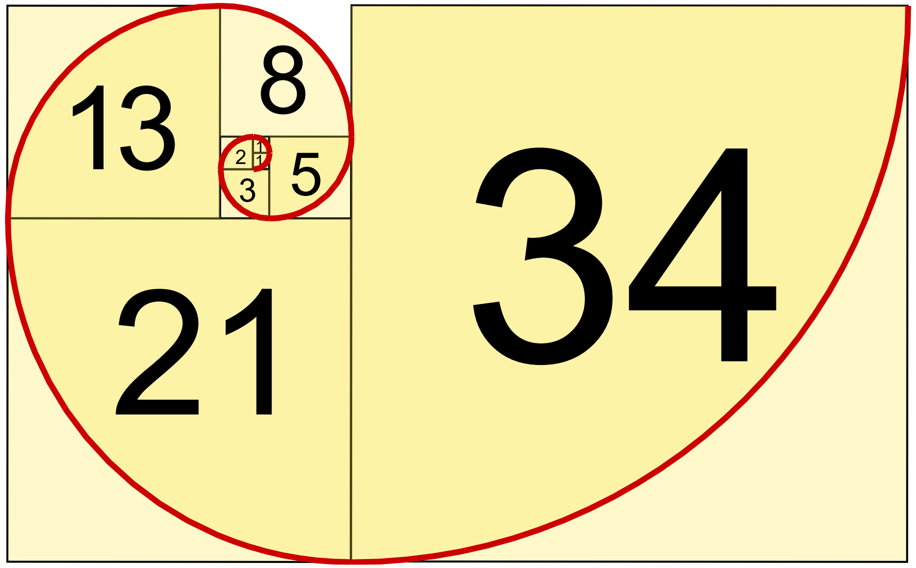
Try them out:

```
print(0, facR(0))

for i in range(1, 10) :
   print(i, facR(i), facI(i))
```

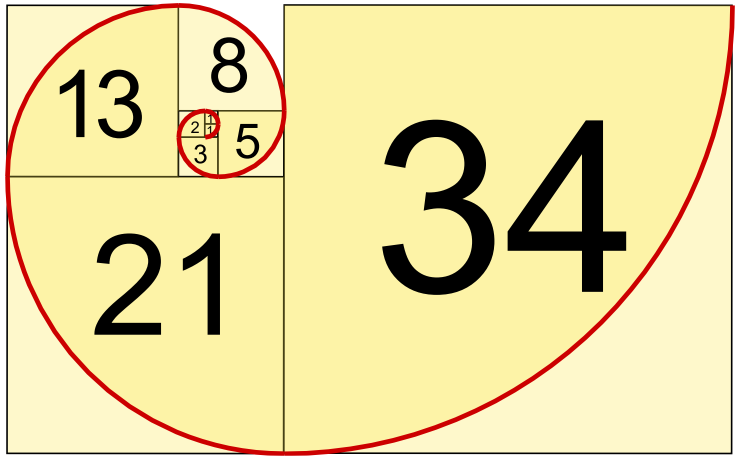
```
0 1
1 1 1
2 2 2
3 6 6
4 24 24
5 120 120
6 720 720
7 5040 5040
8 40320 40320
9 362880 362880
```

# FIBONACCI NUMBERS

- Fibonacci numbers:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, …
- How to write an algorithm to have such a sequence?



# FIBONACCI NUMBERS



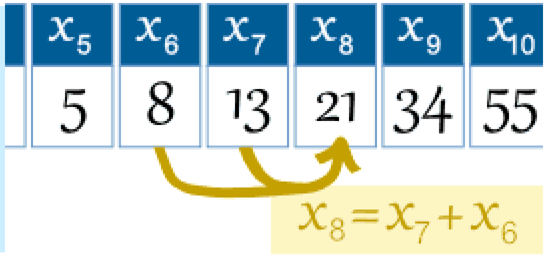
- Fibonacci numbers:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, …

- The rule:

| *n* = | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | … |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_n$ = | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 | … |

Example: the **8th** term is
the **7th** term plus the **6th** term:

$x_8 = x_7 + x_6$



| $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ |
|---|---|---|---|---|---|
| 5 | 8 | 13 | 21 | 34 | 55 |

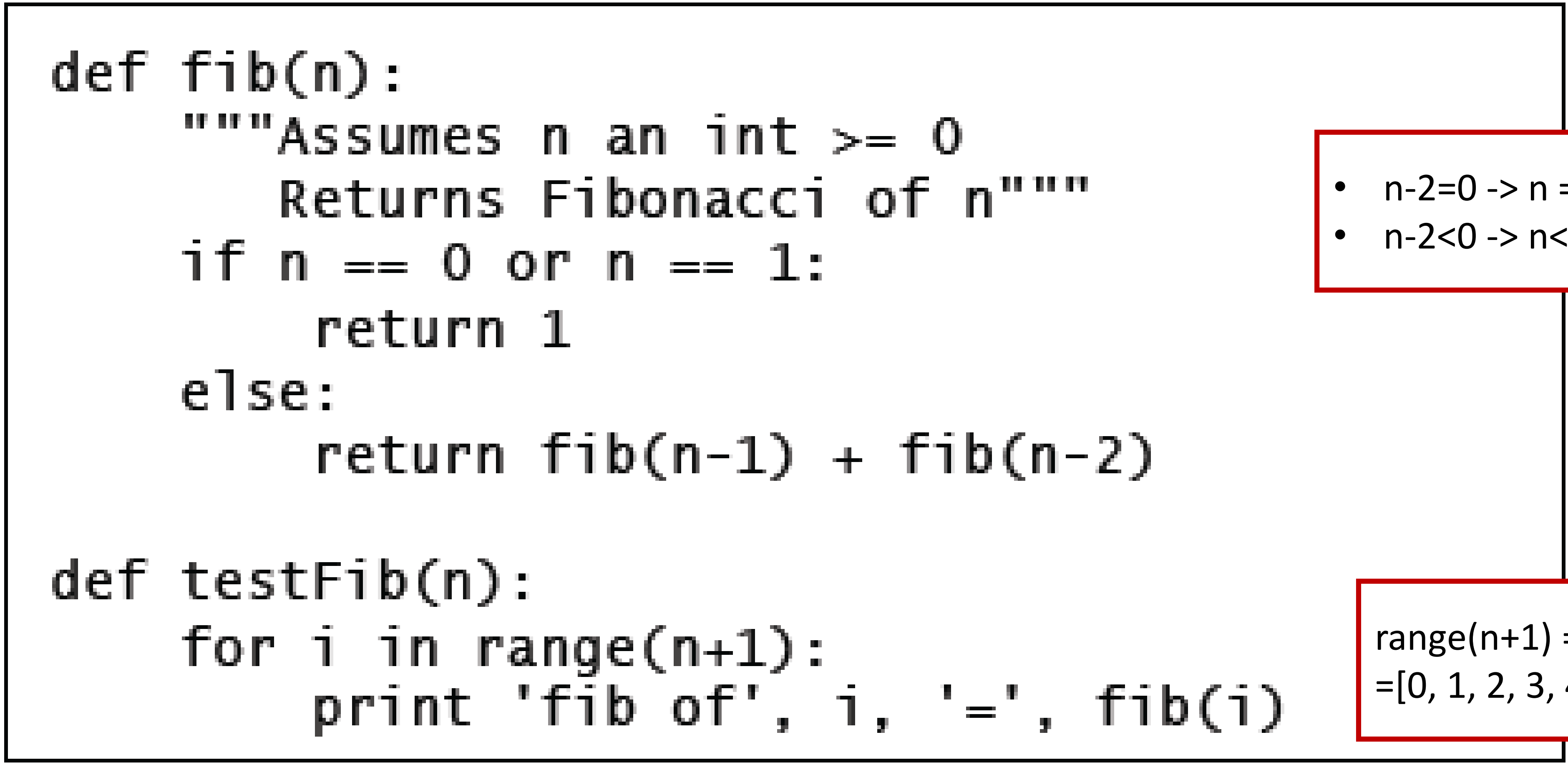
$x_8 = x_7 + x_6$

# FIBONACCI NUMBERS

| *n =* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | *...* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_n$ = | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 | ... |

- Recipes:
  1. f(n) = f(n-1) + f(n-2)
  2. Make f(n) as a function
  3. n-2 can not be less than 0

# FIBONACCI NUMBERS



```
def fib(n):
    """Assumes n an int >= 0
       Returns Fibonacci of n"""
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def testFib(n):
    for i in range(n+1):
        print 'fib of', i, '=', fib(i)
```

- n-2=0 -> n =2
- n-2<0 -> n<2 -> n=0 or n =1

range(n+1) = range(0,n+1,1)
=[0, 1, 2, 3, 4, …, n+1-1]

what is fib(0)? fib(1)? fib(2)? fib(3)?