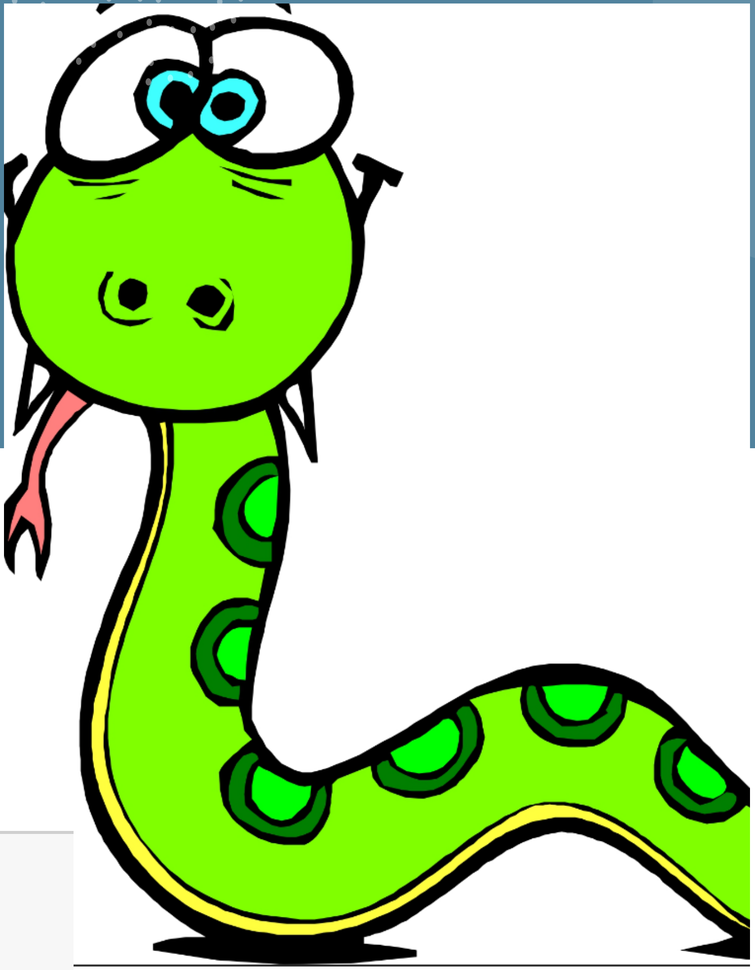
Welcome to my 4th Python Lecture

Lutz Plümer



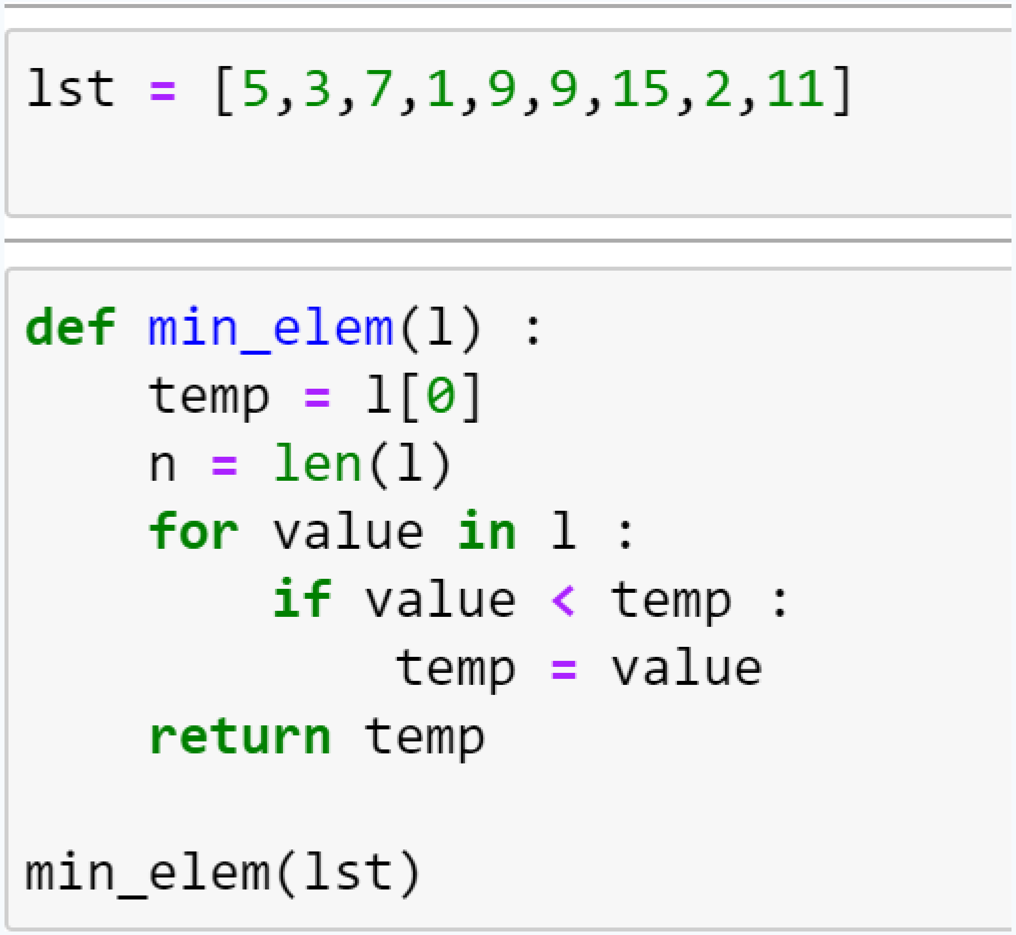
```
WelcomeToMyLecture = "欢迎来到我的讲座"
print(WelcomeToMyLecture)
```

```
欢迎来到我的讲座
```

Midterm Exam

Model Solutions

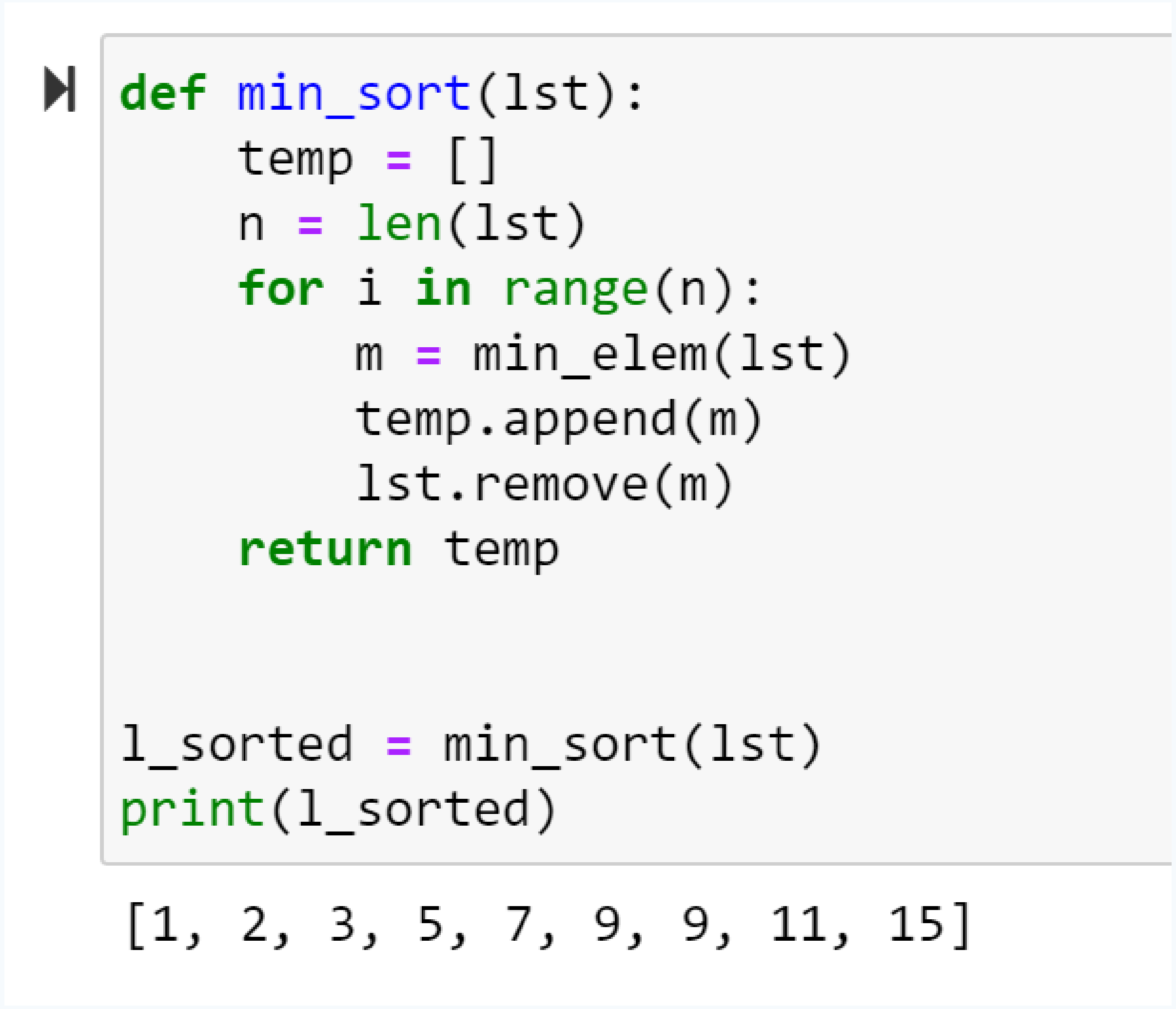
Exercise 2



```
lst = [5,3,7,1,9,9,15,2,11]
```

```
def min_elem(l) :
    temp = l[0]
    n = len(l)
    for value in l :
        if value < temp :
            temp = value
    return temp

min_elem(lst)
```

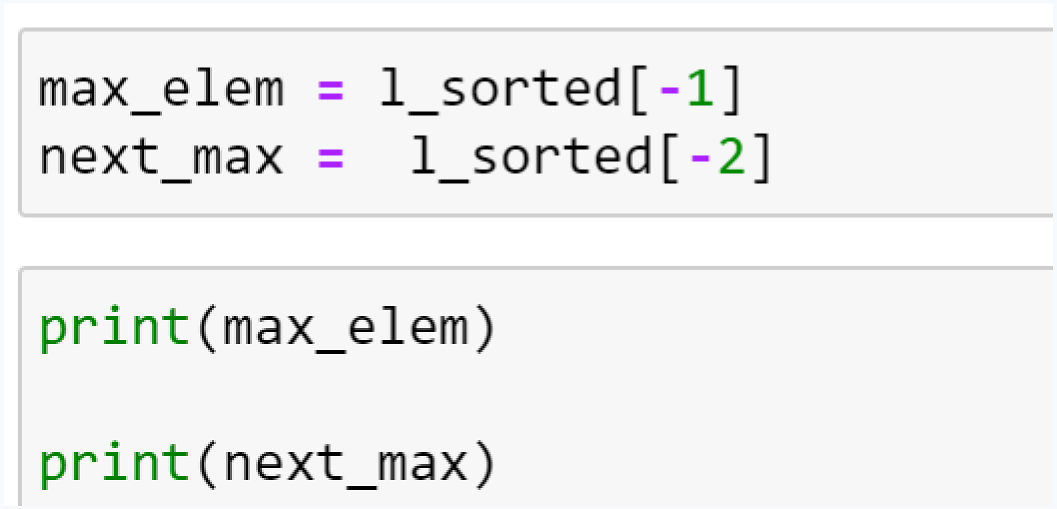
Exercise 3



```python
def min_sort(lst):
    temp = []
    n = len(lst)
    for i in range(n):
        m = min_elem(lst)
        temp.append(m)
        lst.remove(m)
    return temp


l_sorted = min_sort(lst)
print(l_sorted)
```

```
[1, 2, 3, 5, 7, 9, 9, 11, 15]
```

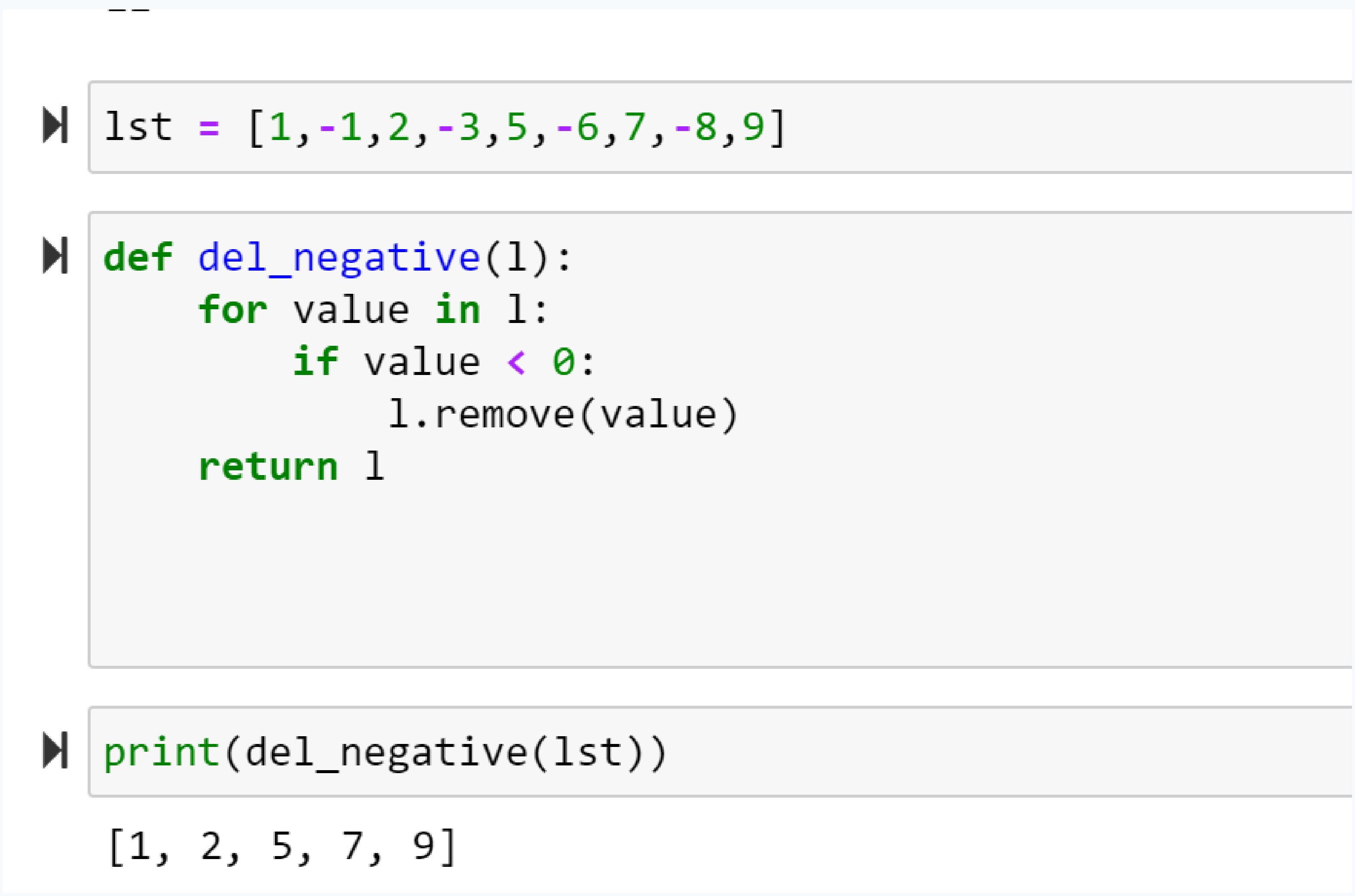
Exercise 4



```
max_elem = l_sorted[-1]
next_max =  l_sorted[-2]
```

```
print(max_elem)

print(next_max)
```

Exercise 5



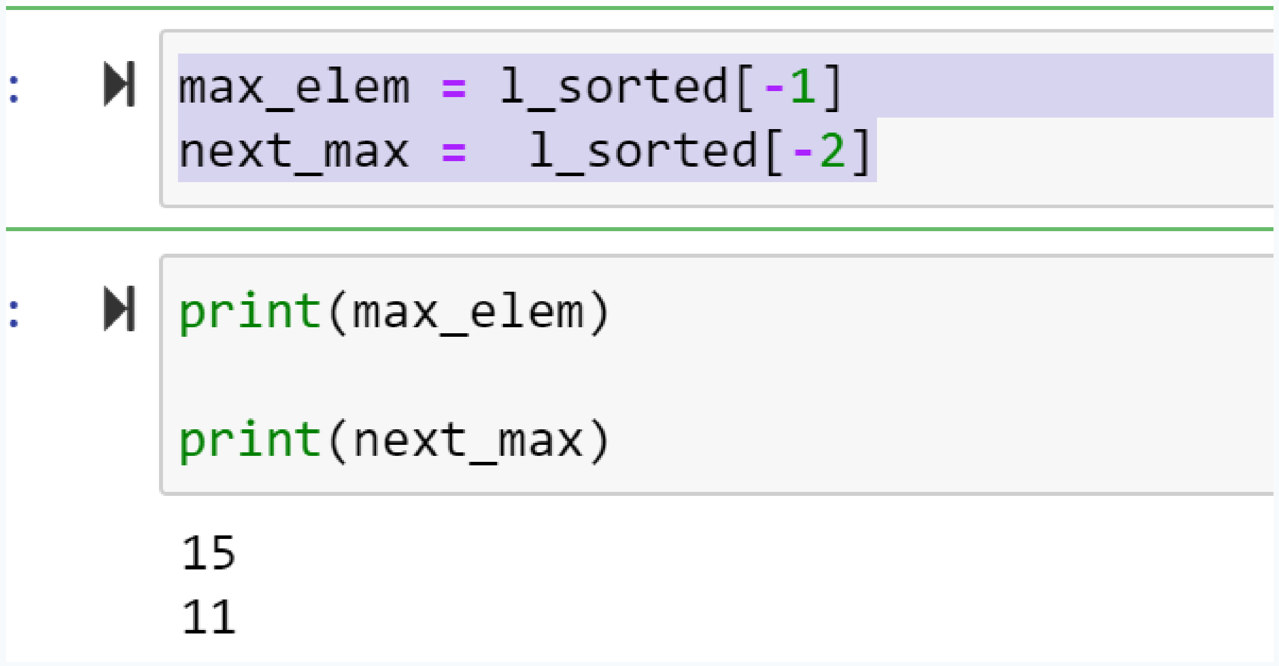
```
lst = [1,-1,2,-3,5,-6,7,-8,9]
```

```
def del_negative(l):
    for value in l:
        if value < 0:
            l.remove(value)
    return l
```

```
print(del_negative(lst))
```

```
[1, 2, 5, 7, 9]
```

Exercise 6



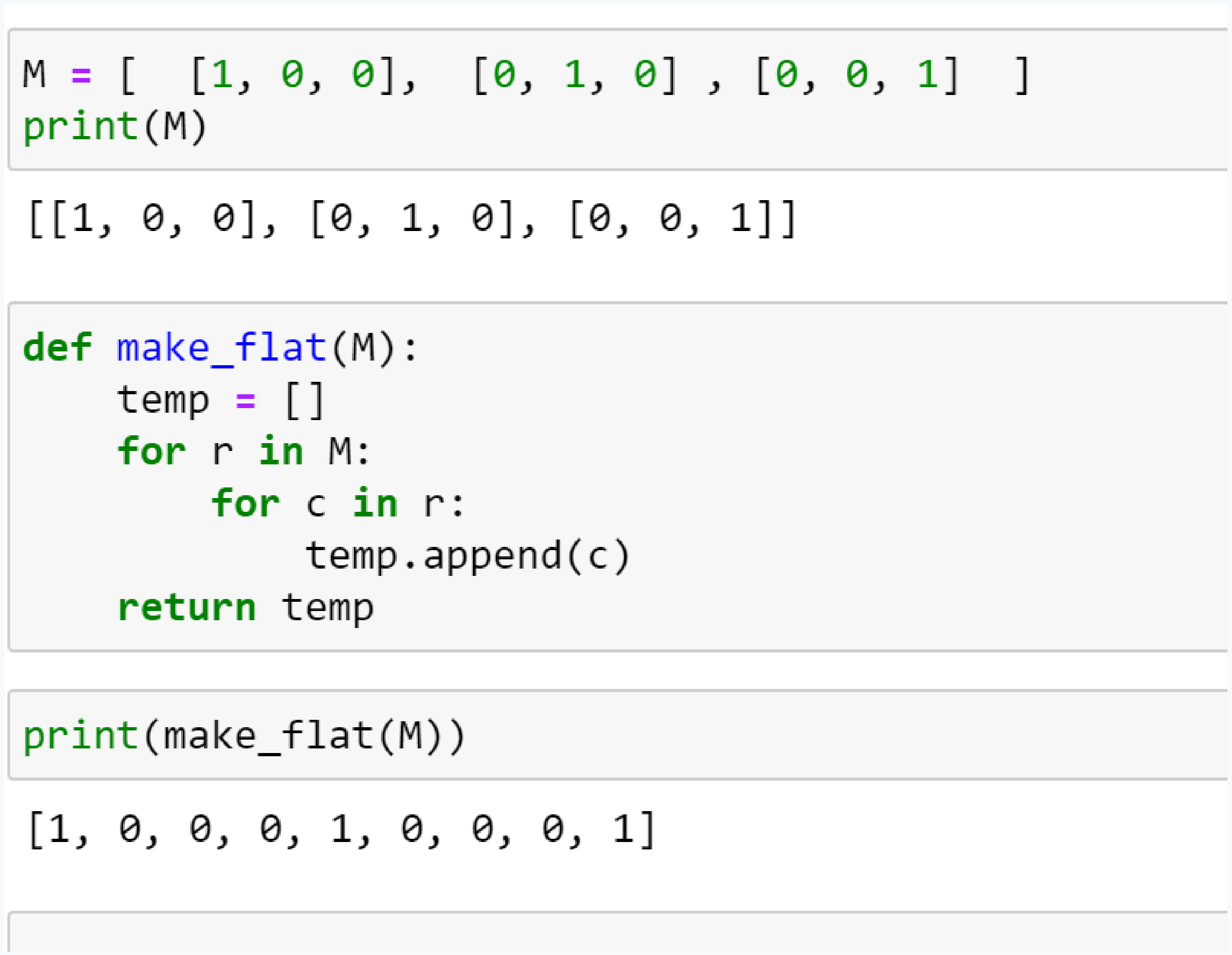
```
max_elem = l_sorted[-1]
next_max =  l_sorted[-2]
```

```
print(max_elem)

print(next_max)
```

```
15
11
```

- Other option: look for them maximum element, remove, and repeat this search

Exercise 7



```
M = [  [1, 0, 0],  [0, 1, 0] , [0, 0, 1]  ]
print(M)
```

```
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

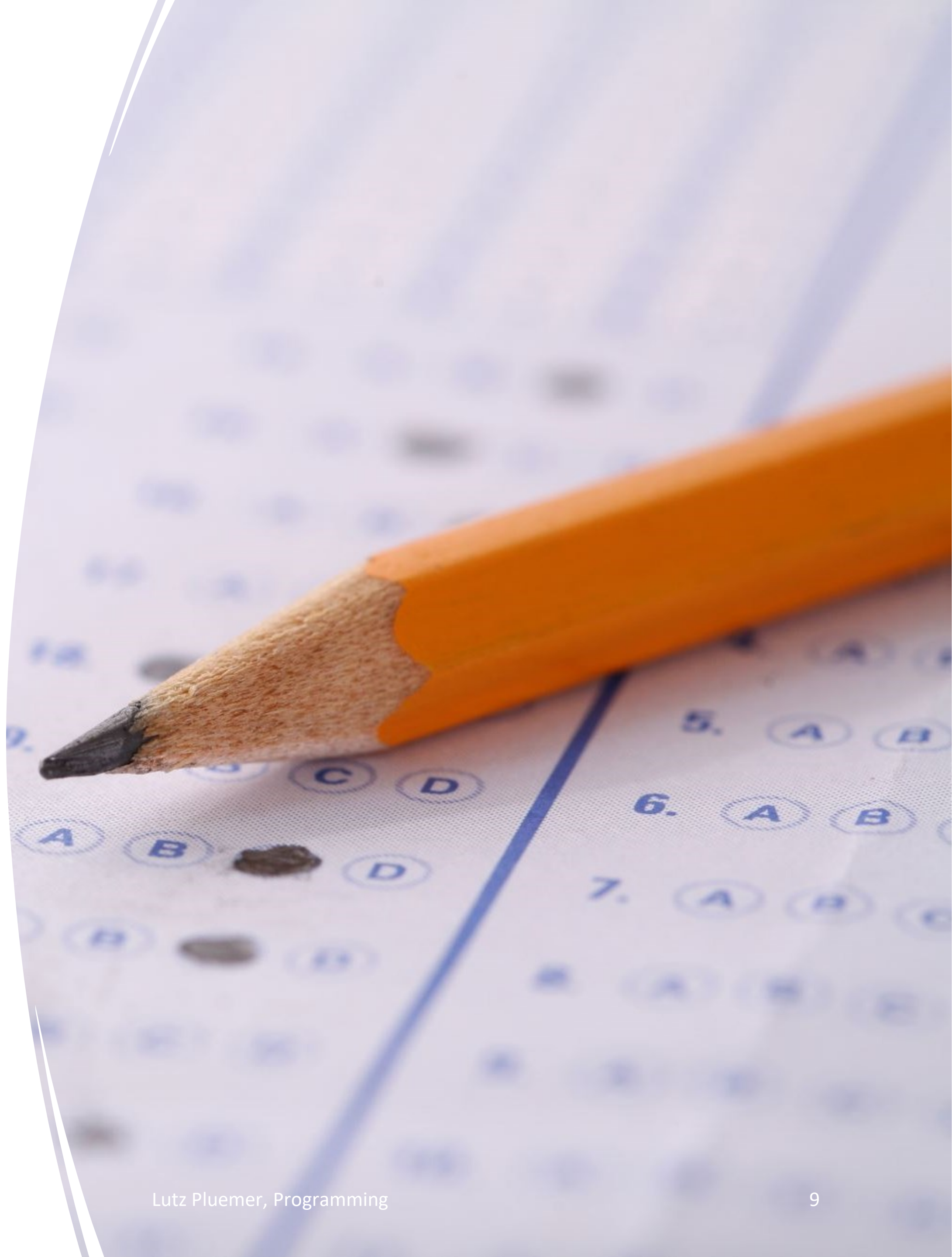
```
def make_flat(M):
    temp = []
    for r in M:
        for c in r:
            temp.append(c)
    return temp
```

```
print(make_flat(M))
```

```
[1, 0, 0, 0, 1, 0, 0, 0, 1]
```

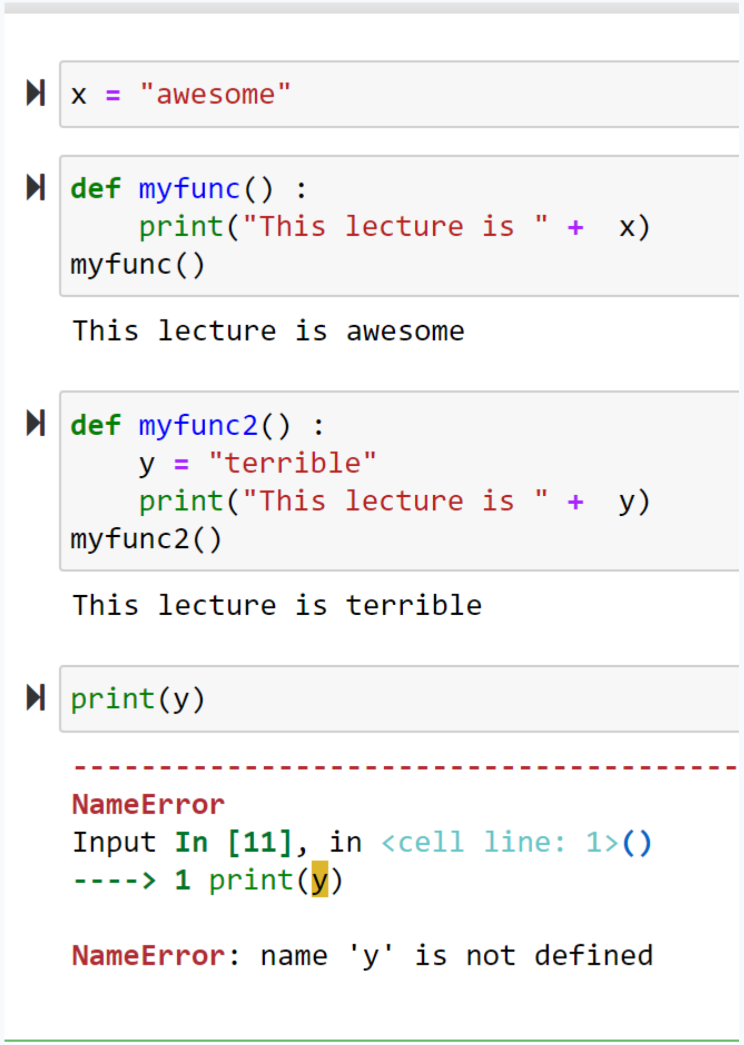
Recursion

Local and Global Variables



Local and Global Variables

- Variable x, defined inside the main program, is visible inside the function
- Variable y, defined inside the function, is not visible for the main program
- We call x a global variable of the program and y a local variable of the function



```
x = "awesome"
```

```
def myfunc() :
    print("This lecture is " +  x)
myfunc()
```

```
This lecture is awesome
```

```
def myfunc2() :
    y = "terrible"
    print("This lecture is " +  y)
myfunc2()
```

```
This lecture is terrible
```

```
print(y)
```

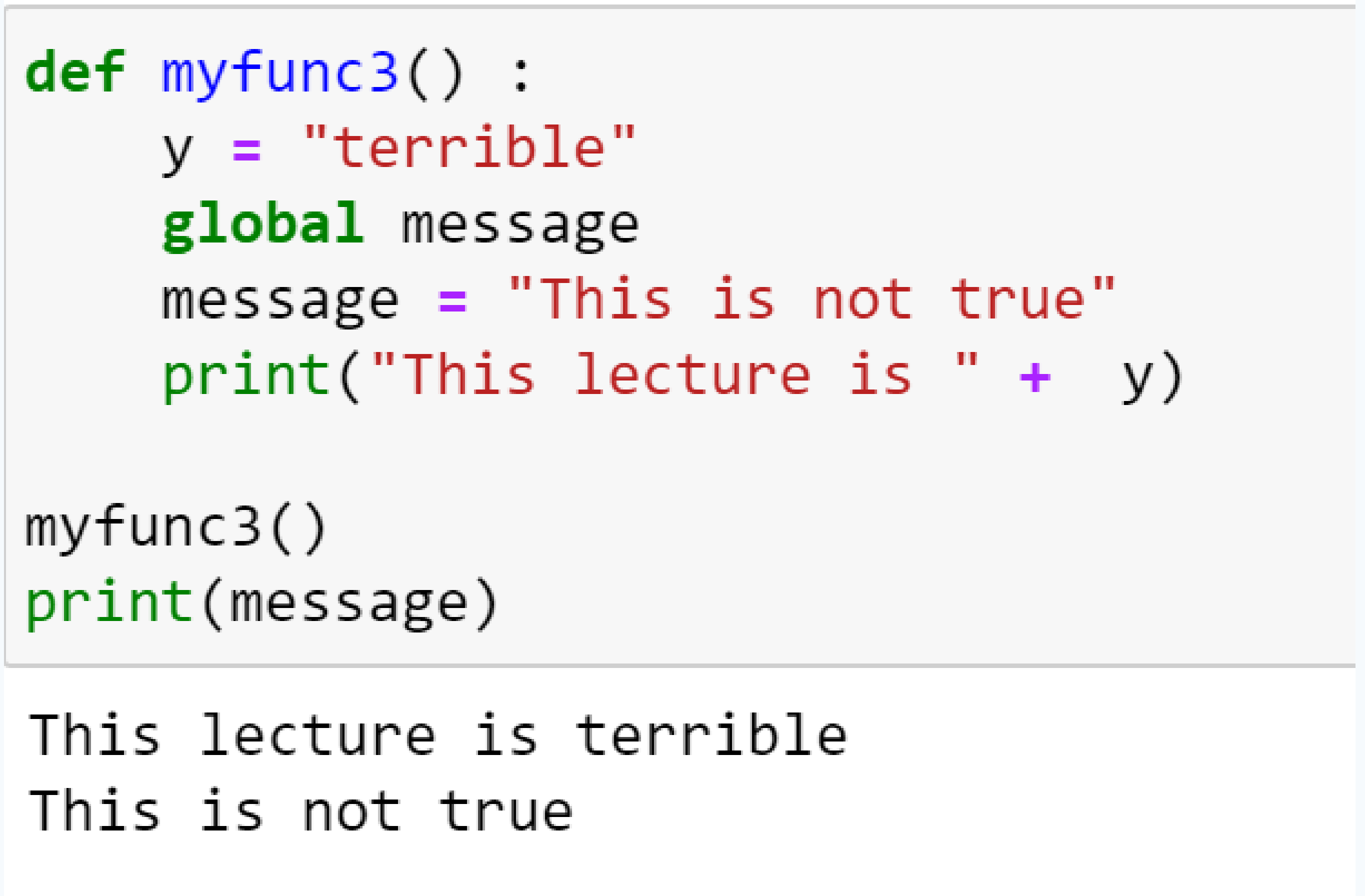
```
---------------------------------------
NameError
Input In [11], in <cell line: 1>()
----> 1 print(y)

NameError: name 'y' is not defined
```

Global and Local Variables - Visibility

- **Global Variables**
- In Python, a variable declared **outside** of the function or in a “global scope” is known as a **global** variable.
  This means that a global variable can be accessed inside or outside of the function.
- **Local Variables**
- When we declare variables **inside** a function, these variables will have a local scope (within the function). We cannot access them outside the function.
- We call this the visibility of the variable.
- This rule promotes the transparency of programs and makes it easier to assign names

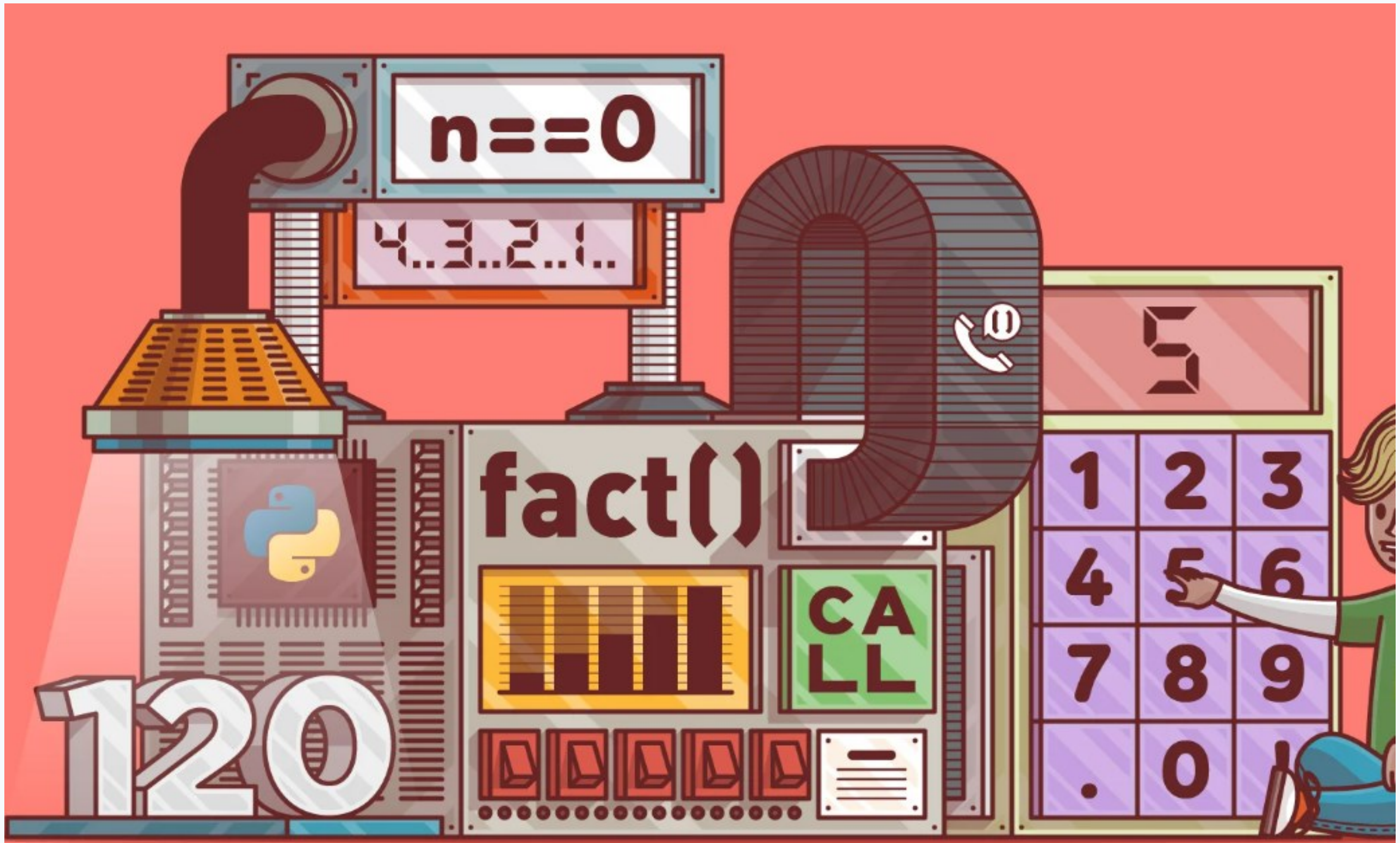
The keyword global

- but you can bypass this rule with the **keyword** global
- this will make the scope, the visibility of the variable **global**
- As a rule, you should not use it



```
def myfunc3() :
    y = "terrible"
    global message
    message = "This is not true"
    print("This lecture is " +  y)

myfunc3()
print(message)
```

```
This lecture is terrible
This is not true
```



Recursion – a smart way to define and solve problems

Recursion – a smart and tricky way to use s

- Recursion works by reducing a problem to the **same** problem of a **smaller** size
- always starting with the trivial case
- Look again at the definition of faculty:

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

You can also write informally:
fac = 1 * 2 * … * n

Note: it starts with 1, not with 0

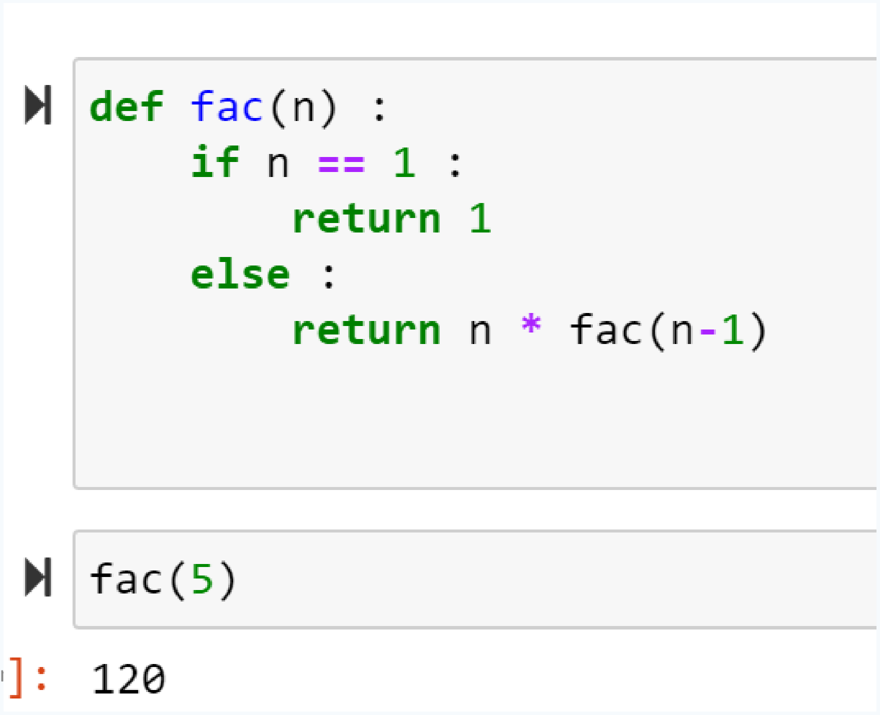
Recursive definition

$$fac(1) = 1$$
$$fac(n) \ = n \ * fac(n-1)$$

A recursive Program for the Faculty Function

- Recursive programs look a bit weird at first glance
- You have to adjust internally to the fact that a function calls itself
- Make sure that the first step is a simple, non-recursive case and that recursion reduces the problem to a smaller problem of the same kind
- Look at this – the left side is “pseudo code”, the right side Python

```
if n == 1 :
        fac(n) = 1
else :
        n * fac(n-1)
```



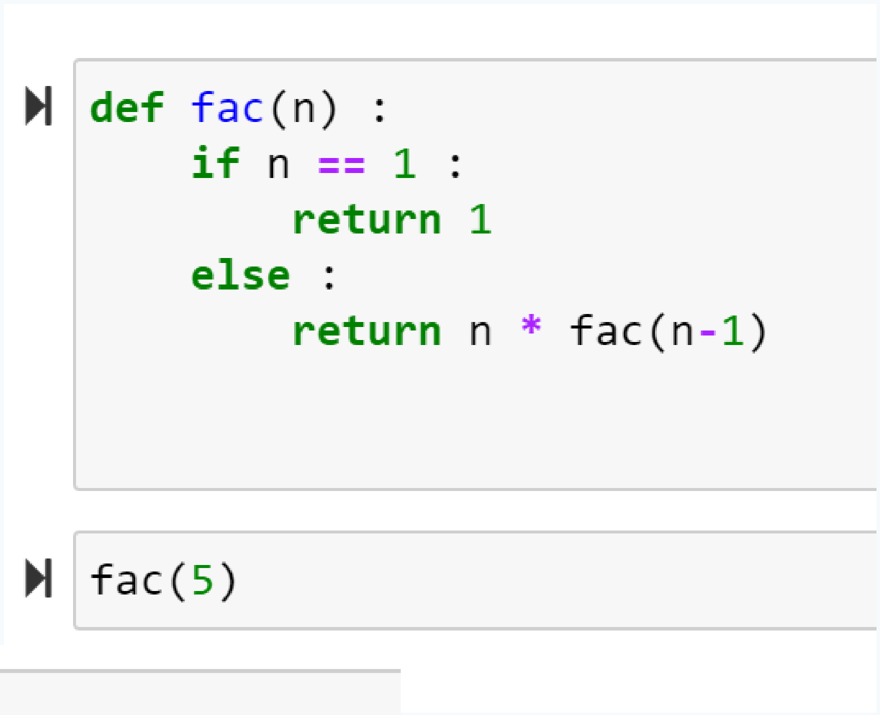
```
def fac(n) :
    if n == 1 :
        return 1
    else :
        return n * fac(n-1)
```

```
fac(5)
```

```
]:  120
```

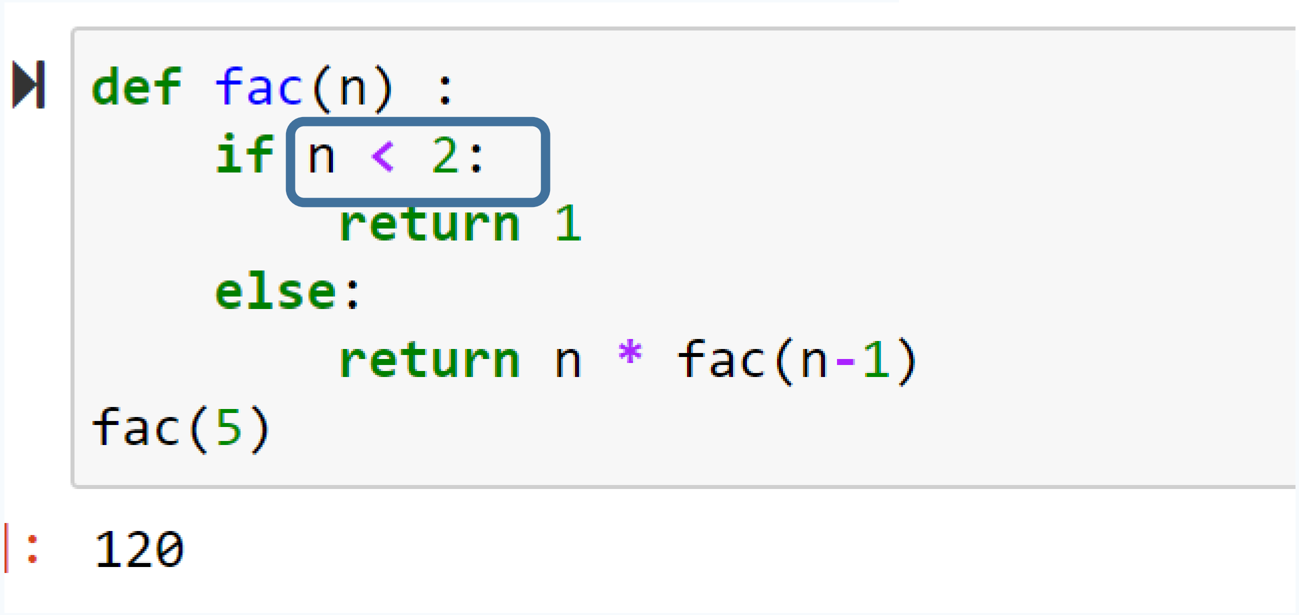
Take care

- But take care: this program, as it is, assumes that n is a positive integer
- If you call it with 3.5, it will never end
- So you may better write



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

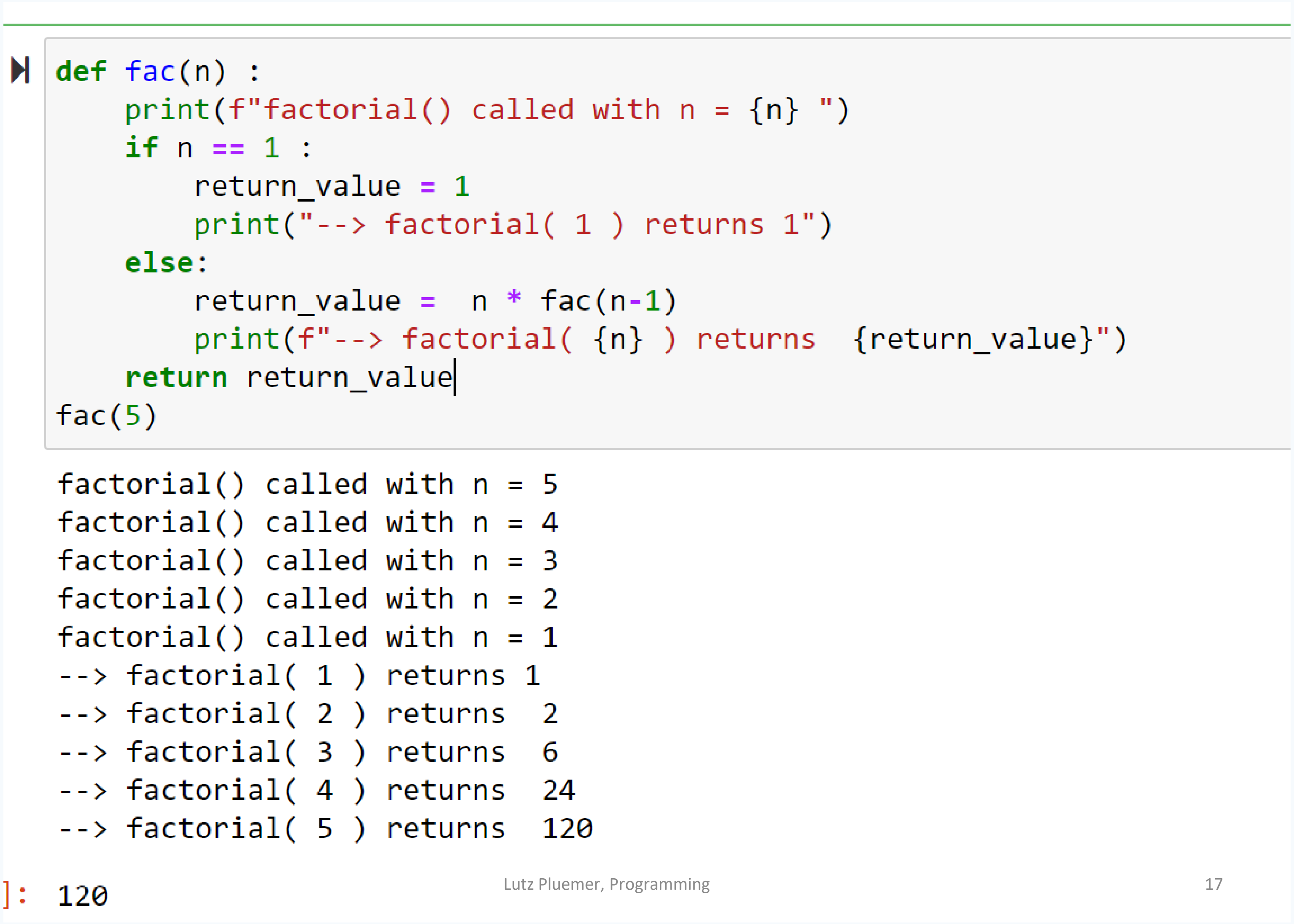
```
fac(5)
```



```
def fac(n) :
    if n < 2:
        return 1
    else:
        return n * fac(n-1)
fac(5)
```

```
120
```

Let’s see how it works



```
def fac(n) :
    print(f"factorial() called with n = {n} ")
    if n == 1 :
        return_value = 1
        print("--> factorial( 1 ) returns 1")
    else:
        return_value =  n * fac(n-1)
        print(f"--> factorial( {n} ) returns  {return_value}")
    return return_value
fac(5)
```

```
factorial() called with n = 5
factorial() called with n = 4
factorial() called with n = 3
factorial() called with n = 2
factorial() called with n = 1
--> factorial( 1 ) returns 1
--> factorial( 2 ) returns  2
--> factorial( 3 ) returns  6
--> factorial( 4 ) returns  24
--> factorial( 5 ) returns  120
```

```
]:  120
```

How it works

- Notice how all the recursive calls **stack** up.
- The function gets called with n = 4, 3, 2, and 1 in succession …
- before any of the calls return.
- Finally, when n is 1, the problem can be solved without any more recursion.
- Then each of the stacked-up recursive calls unwinds back out,
- returning 1, 2, 6, 24 and finally 120 from the outermost call.

F-strings

- The formatting of the output is done by f-strings.
- **f-strings** are string literals that have an **f** at the beginning and **curly braces** containing **expressions** that will be **replaced** with their **values**.
- As here:

```
print(f"--> factorial( {n} ) returns  {return_value}")
```

Fibonacci Numbers

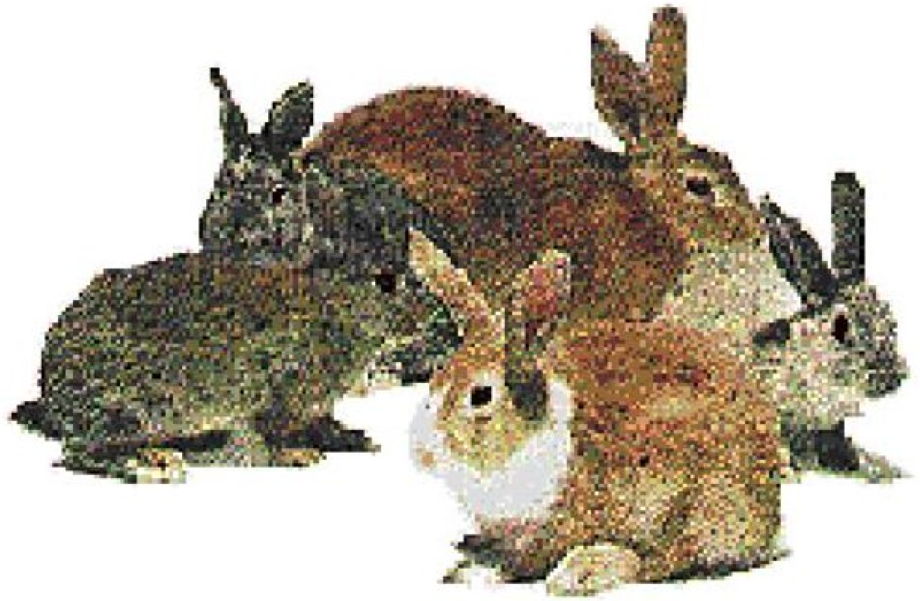
In the Year of Rabbits

The Fascinating FIBONACCI NUMBERS

Fibonacci (1170 - 1240) is considered one of the greatest European mathematicians of the Middle Ages. He was born in Pisa, the Italian city famous for its leaning tower. His father was a customs officer in North Africa, so the Fibonacci increased among North African civilization, making, however, many trips around the Mediterranean coasts.

Fibonacci is known as one of the first who introduced arabic numerals to Europe, numbers that we use today: 0, 1, 2, 3, .. 9.

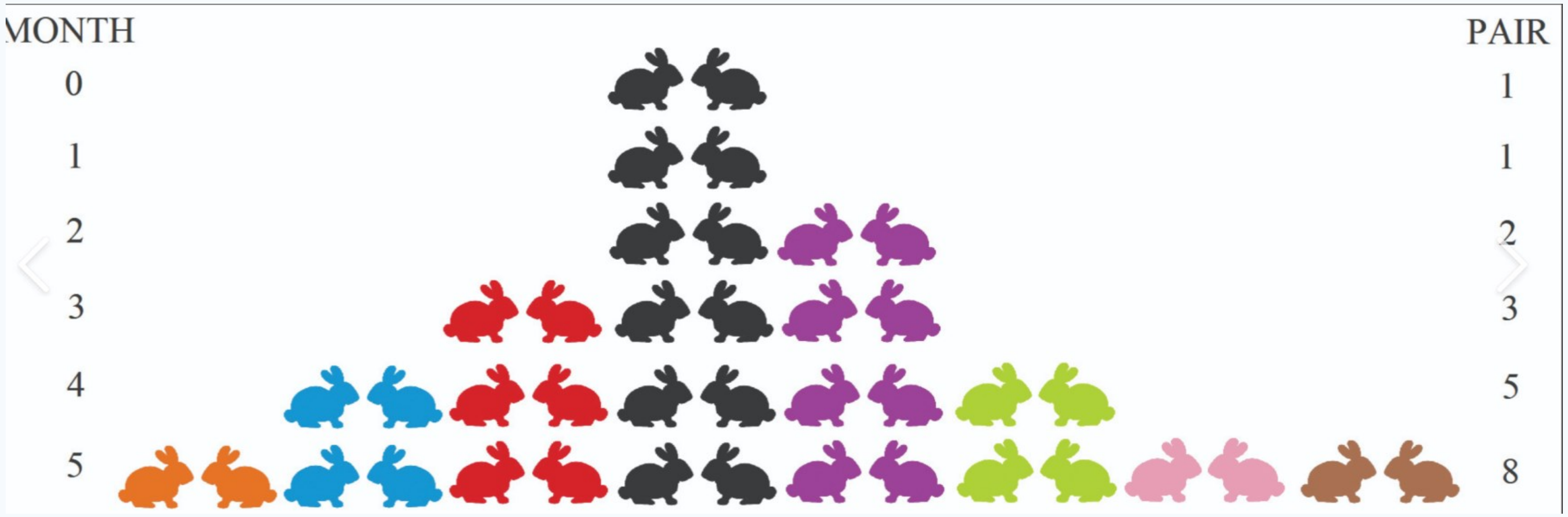
Fibonacci's Rabbits



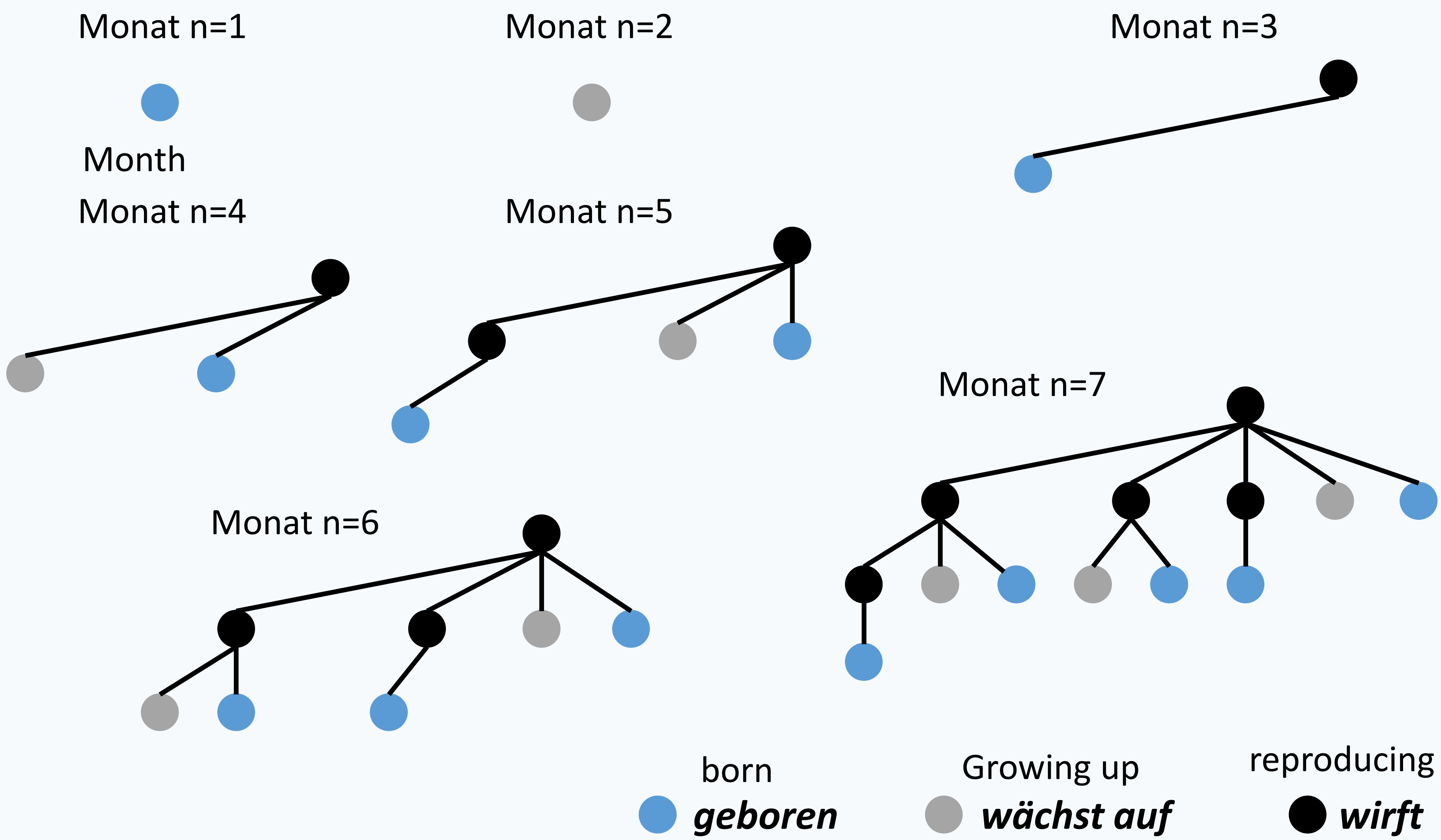
Problem:

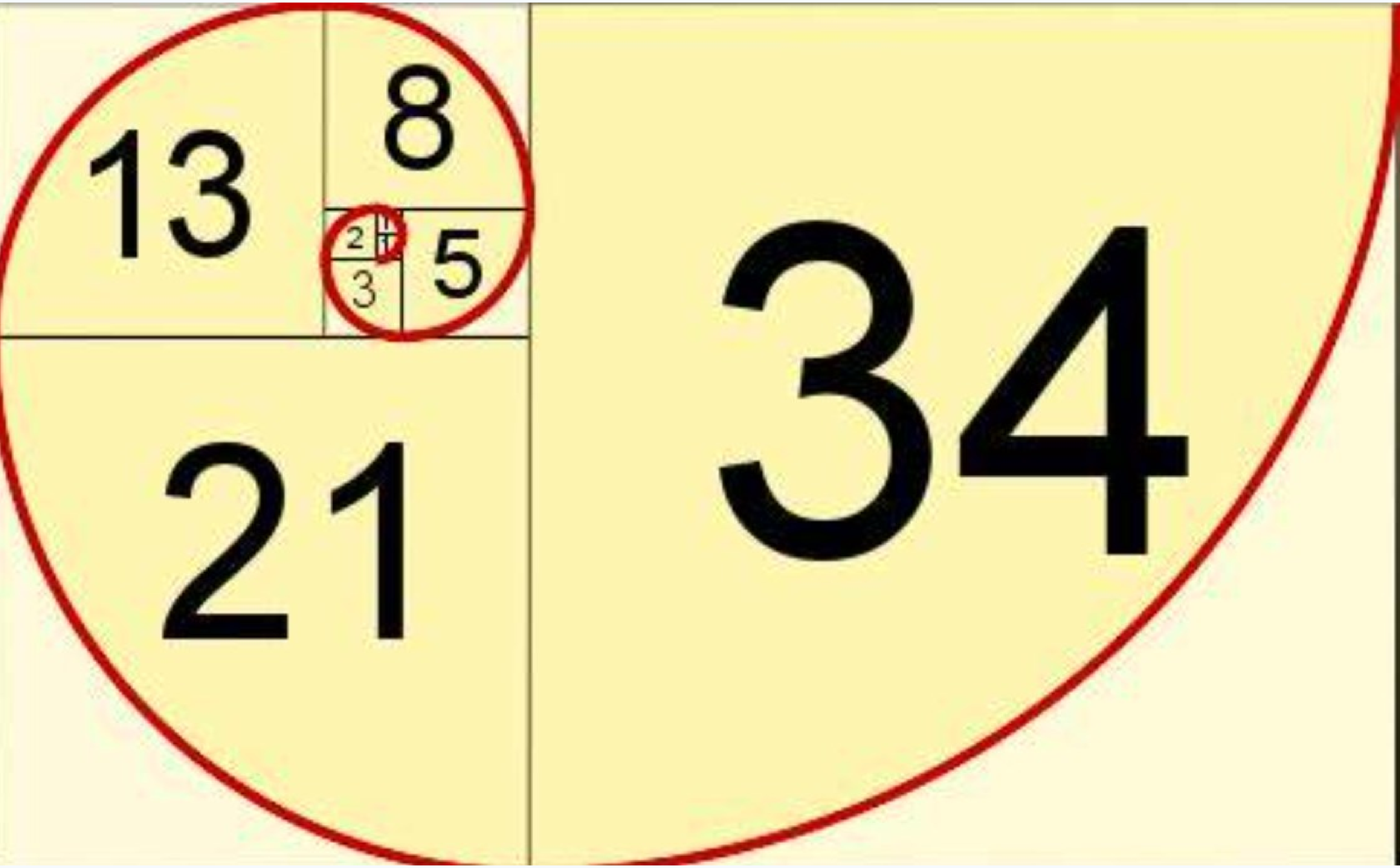
Suppose a newly-born pair of rabbits (one male, one female) are put in a field. Rabbits are able to mate at the age of one month so that at the end of its second month, a female can produce another pair of rabbits. Suppose that the rabbits never die and that the female always produces one new pair (one male, one female) every month from the second month on. How many pairs will there be in one year?

This leads to the sequence of Fibonacci Numbers:
1, 1, 3, 5, 8, 13, 21, …



Number of Rabbits after n Monthts




13
8
2
1
1
3
5
21
34

Fibonacci Numbers have many Applications in Nature and Art

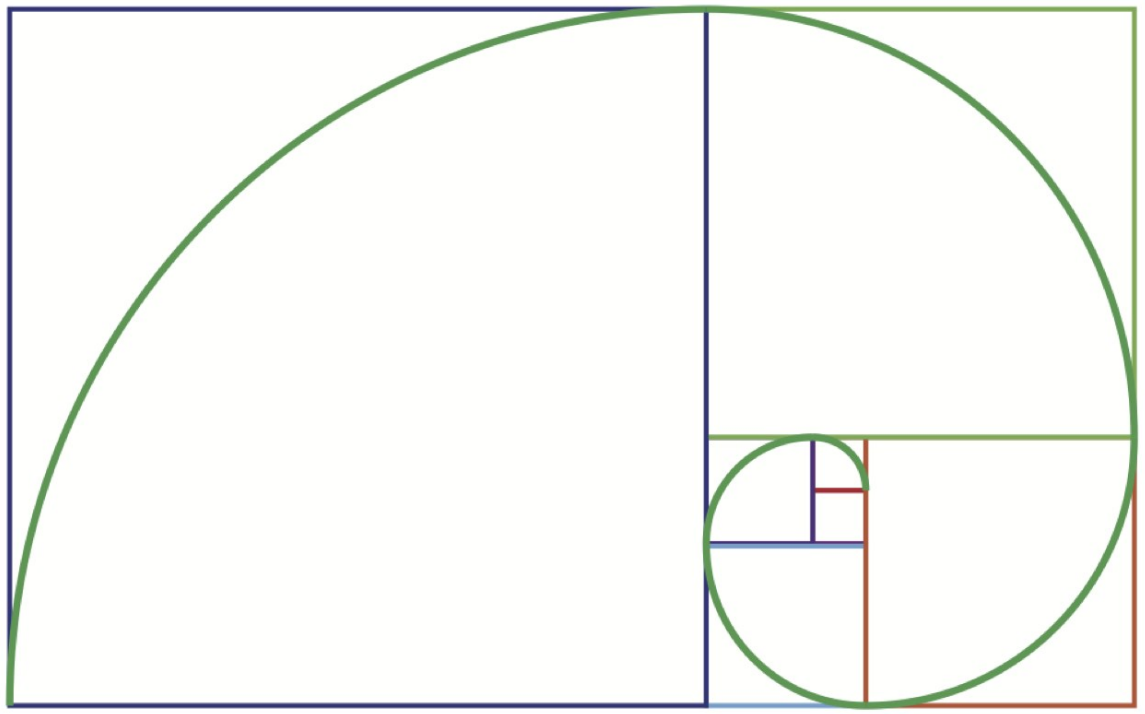
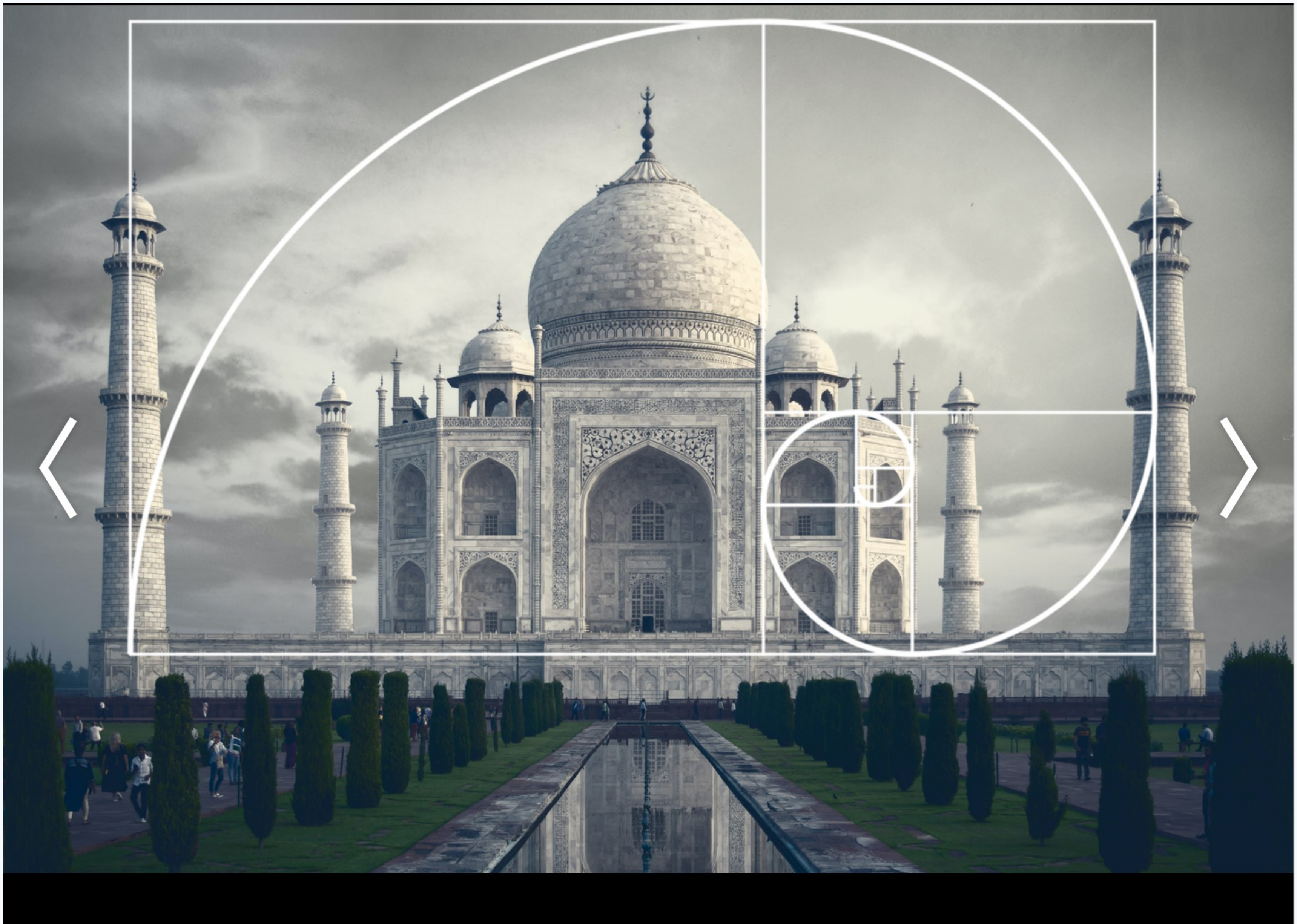
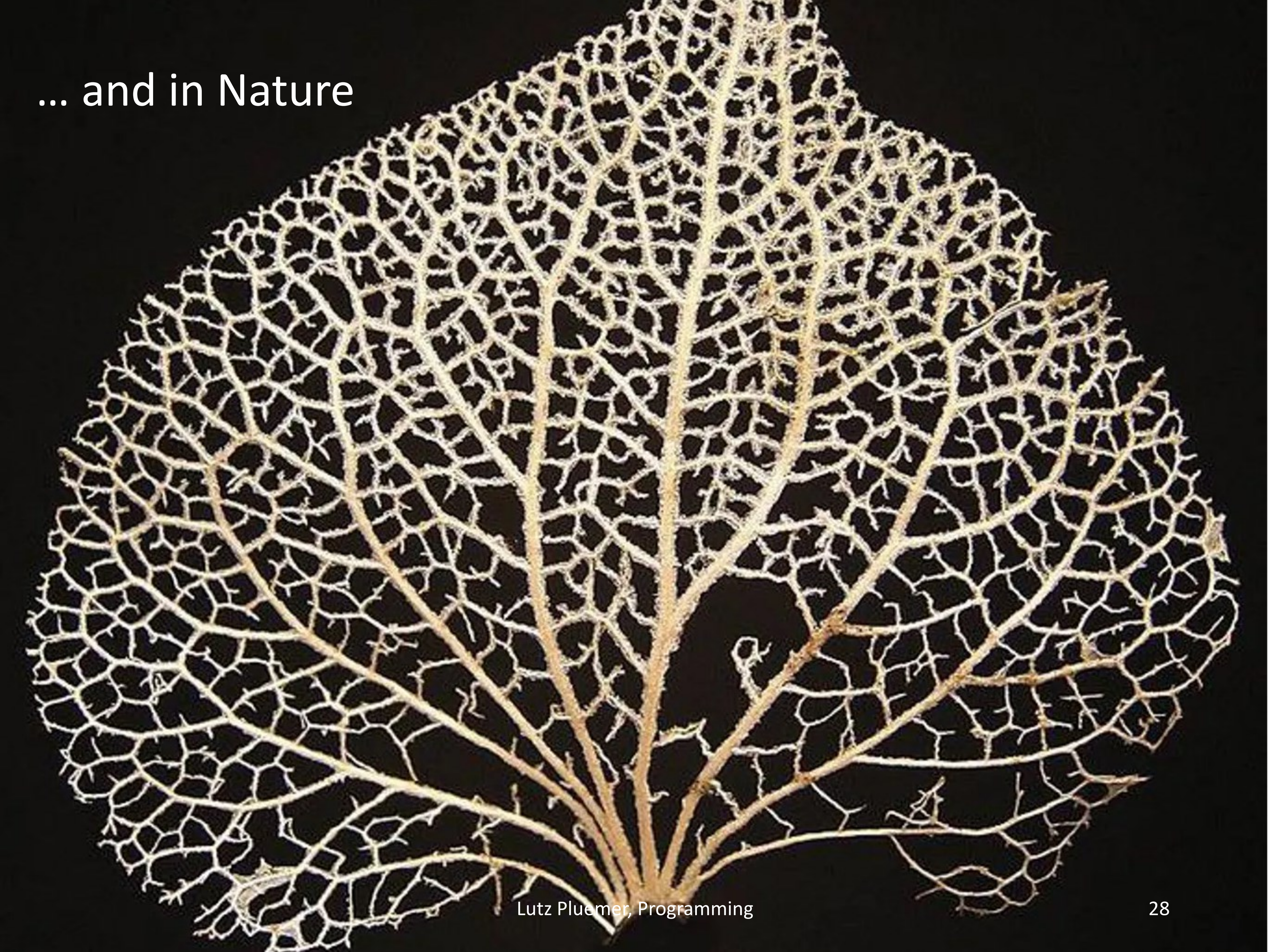


Photo: J Brew (bottom left) via WIkimedia Commons

Applications in Classical Architecture



… and in Nature

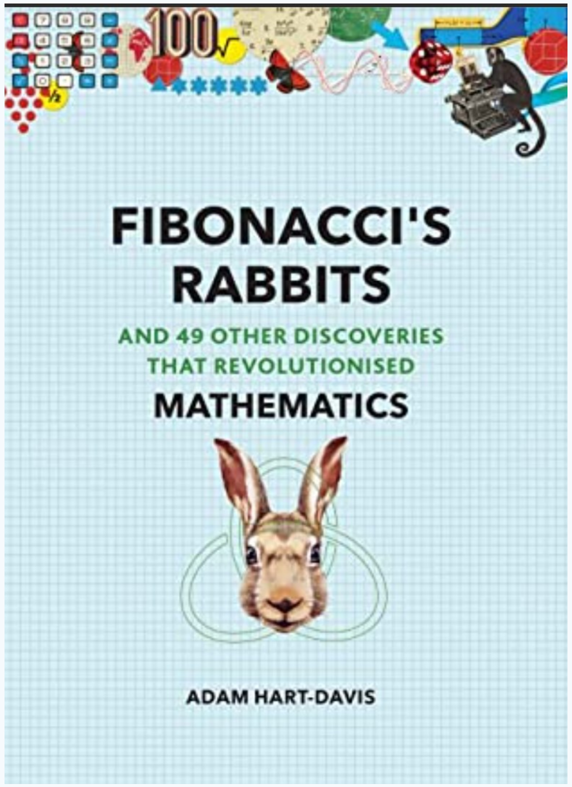


Sequence of Fibonacci Numbers

- Wehave the following sequence of numbers:
  1, 1, 2, 3, 5, 8, 13, 21, …
- More general:
- $fib(n) = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ fib(n-2) + fib(n-1) & n > 2 \end{cases}$

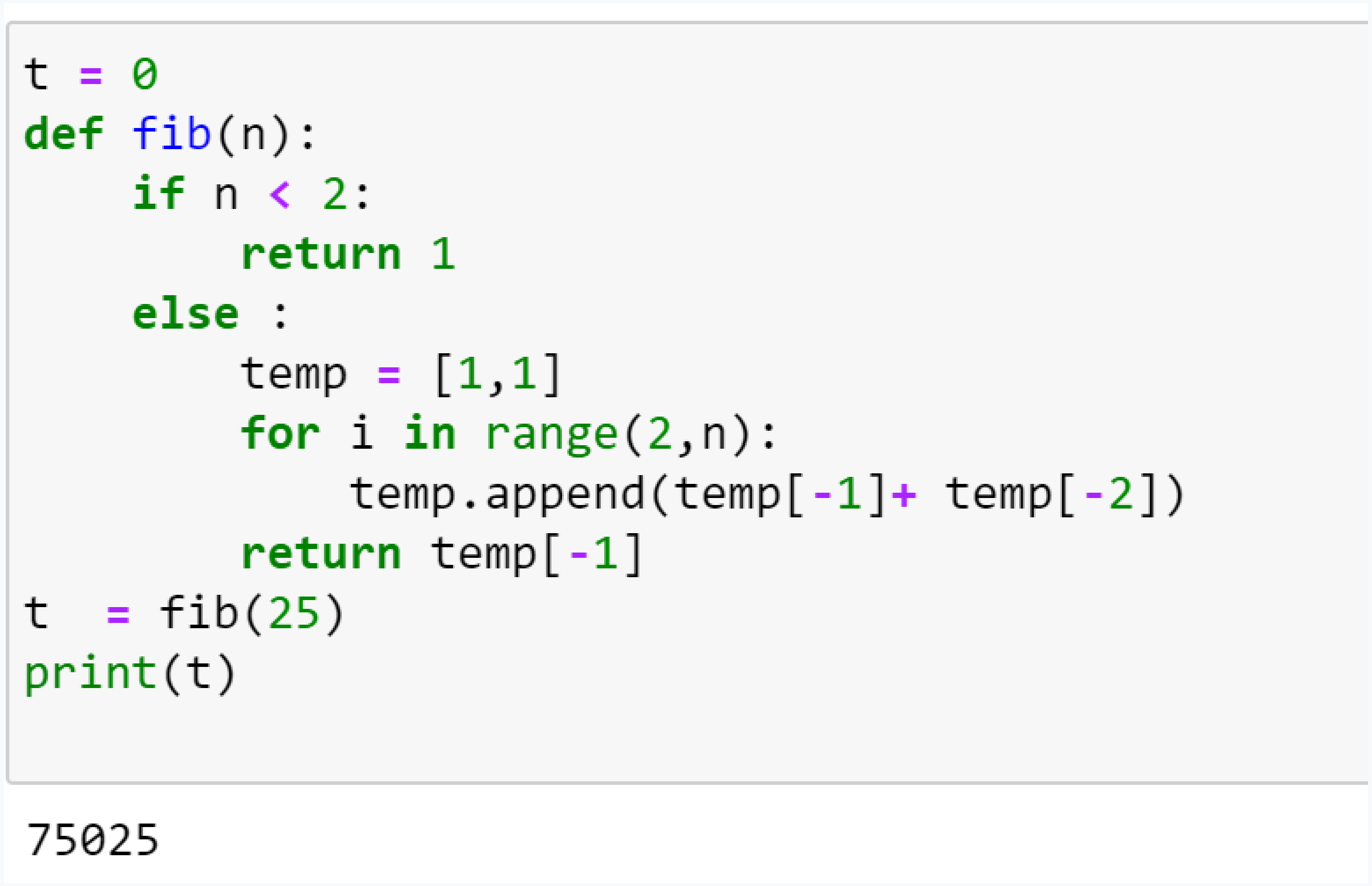
How to program?

- I will show you two different approaches:
- An iterative approach, which generates the sequence of Fibonacci numbers
- A recursive approach, which directly translates the recursive definition



$$fib(n) = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ fib(n-2) + fib(n-1) & n > 2 \end{cases}$$

The iterative approach

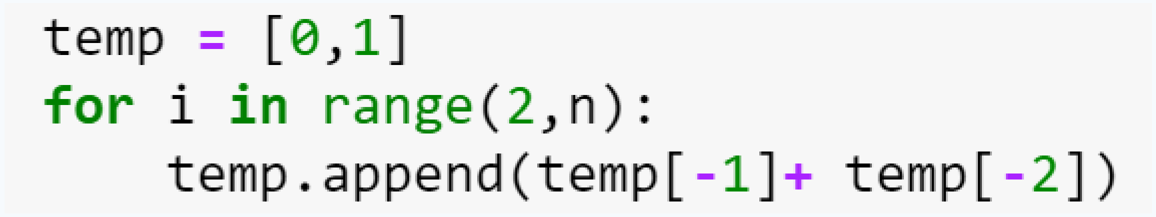


```
t = 0
def fib(n):
    if n < 2:
        return 1
    else :
        temp = [1,1]
        for i in range(2,n):
            temp.append(temp[-1]+ temp[-2])
        return temp[-1]
t  = fib(25)
print(t)
```

```
75025
```

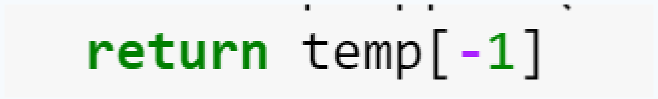
Comments

- Main work is done in a for loop construction a list step by step using append:



```
temp = [0,1]
for i in range(2,n):
    temp.append(temp[-1]+ temp[-2])
```

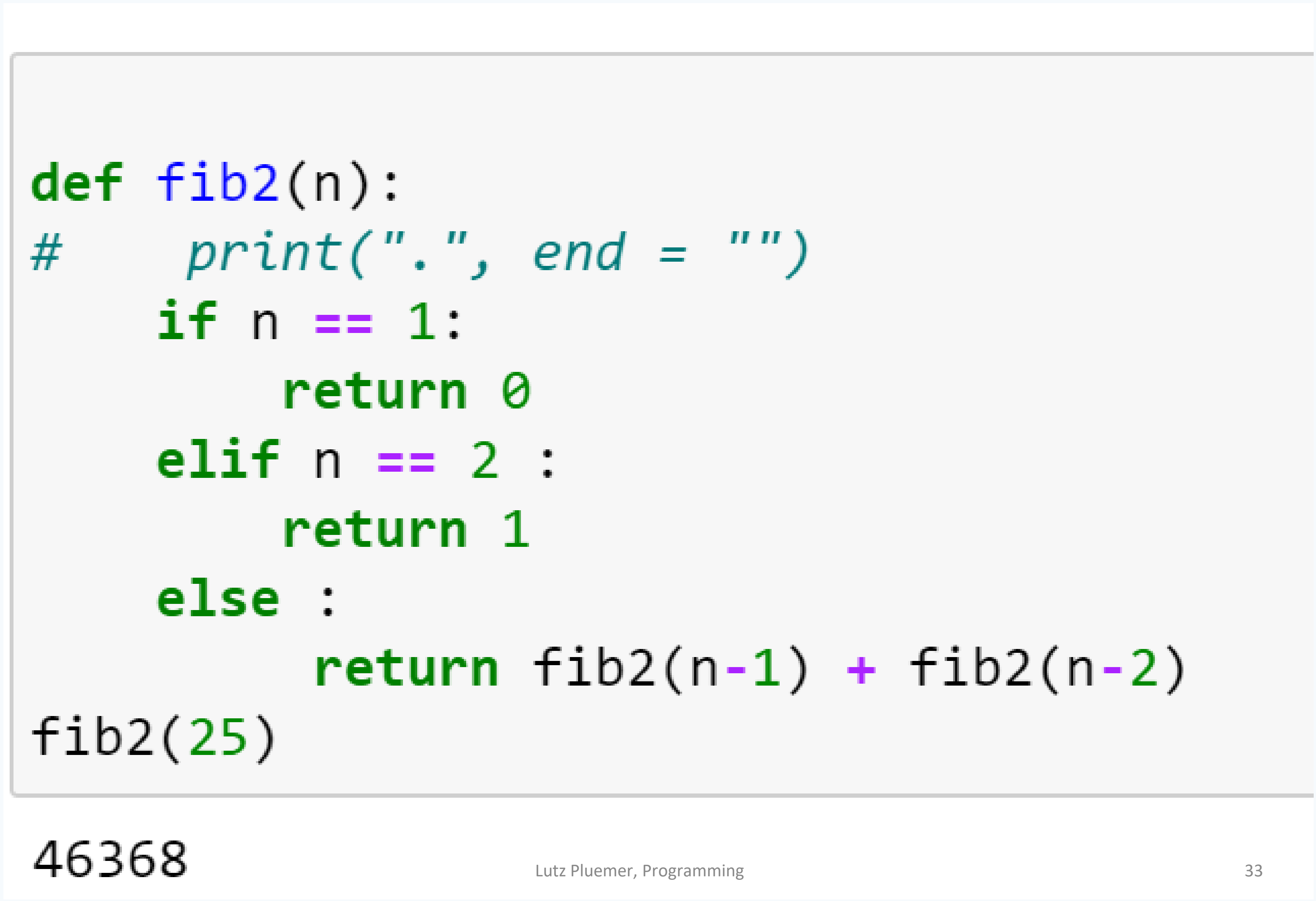
- The result is the last element of this list



```
  return temp[-1]
```

- This function is very, very fast, you can call it with big numbers
- You can also modify:
  return temp[-1], temp
  and call it with:
  f, list = fib(n)
  then you can also see the sequence

The recursive Version

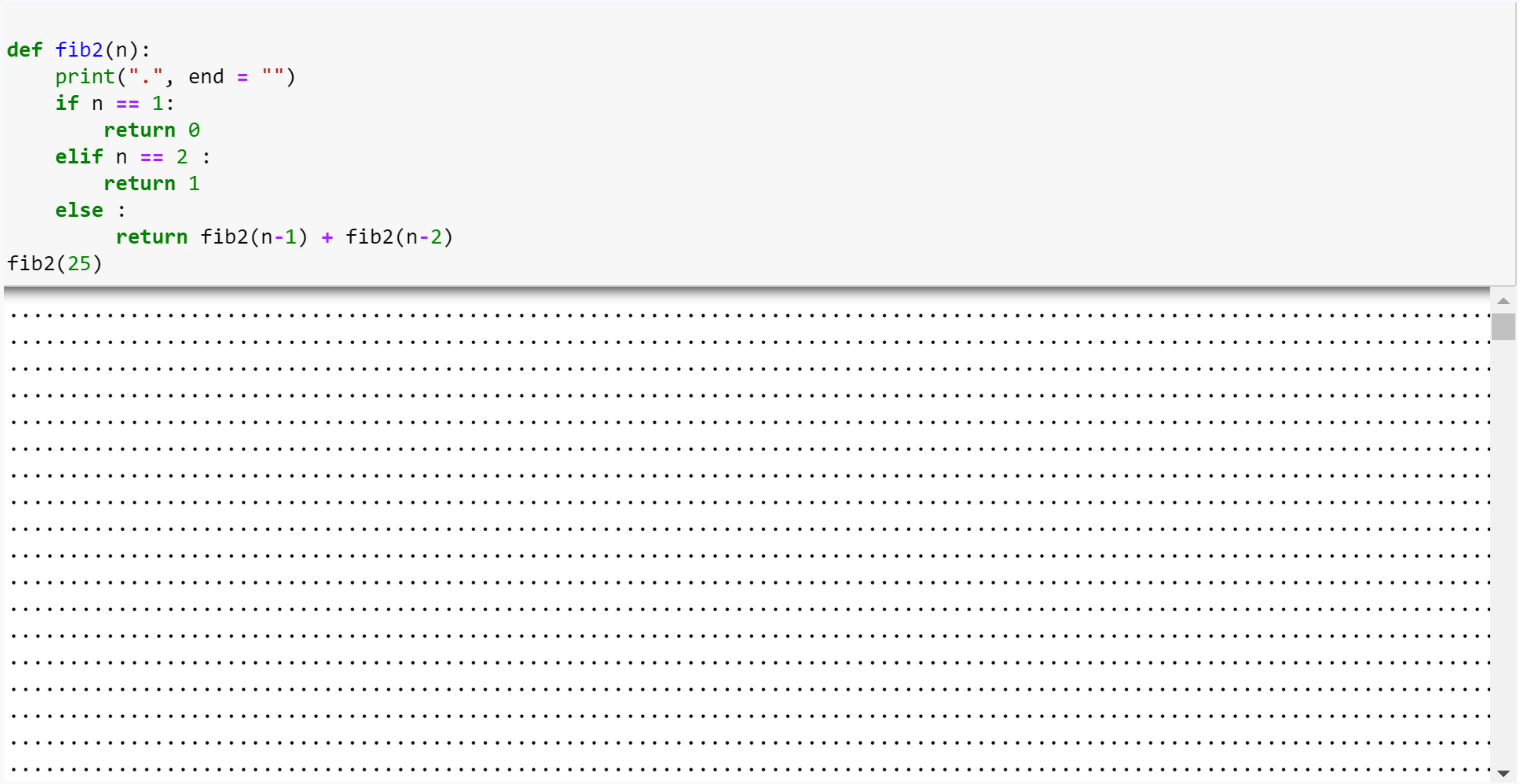


```
def fib2(n):
#    print(".", end = "")
    if n == 1:
        return 0
    elif n == 2 :
        return 1
    else :
         return fib2(n-1) + fib2(n-2)
fib2(25)
```

```
46368
```

Comments

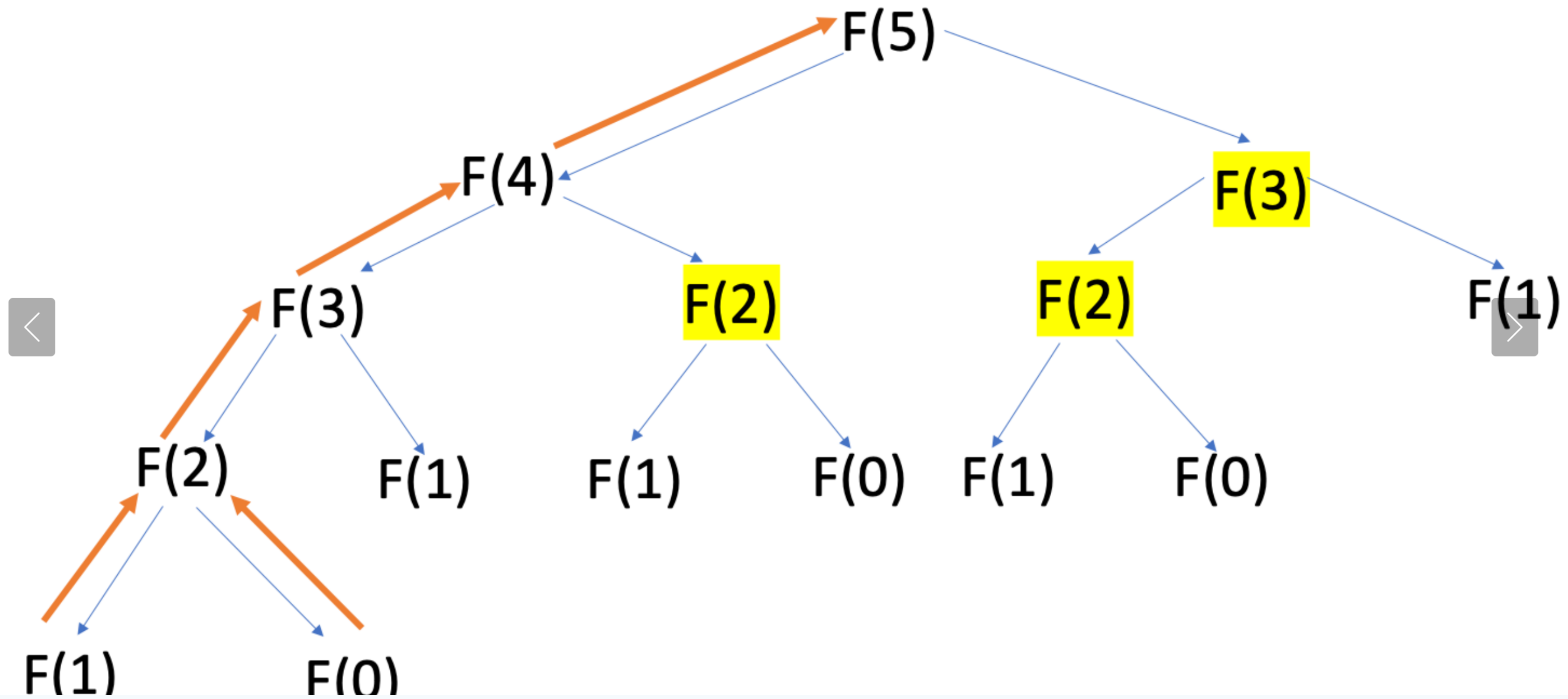
- We have used a new name, **fib2**
- Second line is a comment, started by **#**
- Programming is very easy, nearly the same as the definition
- But you should only use it for smaller numbers, maybe up to **35**, and even that takes some time. Why?
- Lets try to apply the print statement, by uncommenting, means: cancel the #
- On the next slide you see the output: a huge number of dots, which means a huge number of recursive function calls
- Such as pairs of rabbits



```python
def fib2(n):
    print(".", end = "")
    if n == 1:
        return 0
    elif n == 2 :
        return 1
    else :
         return fib2(n-1) + fib2(n-2)
fib2(25)
```

```
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
...........................................................................................................................
```

Here F stands for fib2



Resume: Recursive Definition fib2

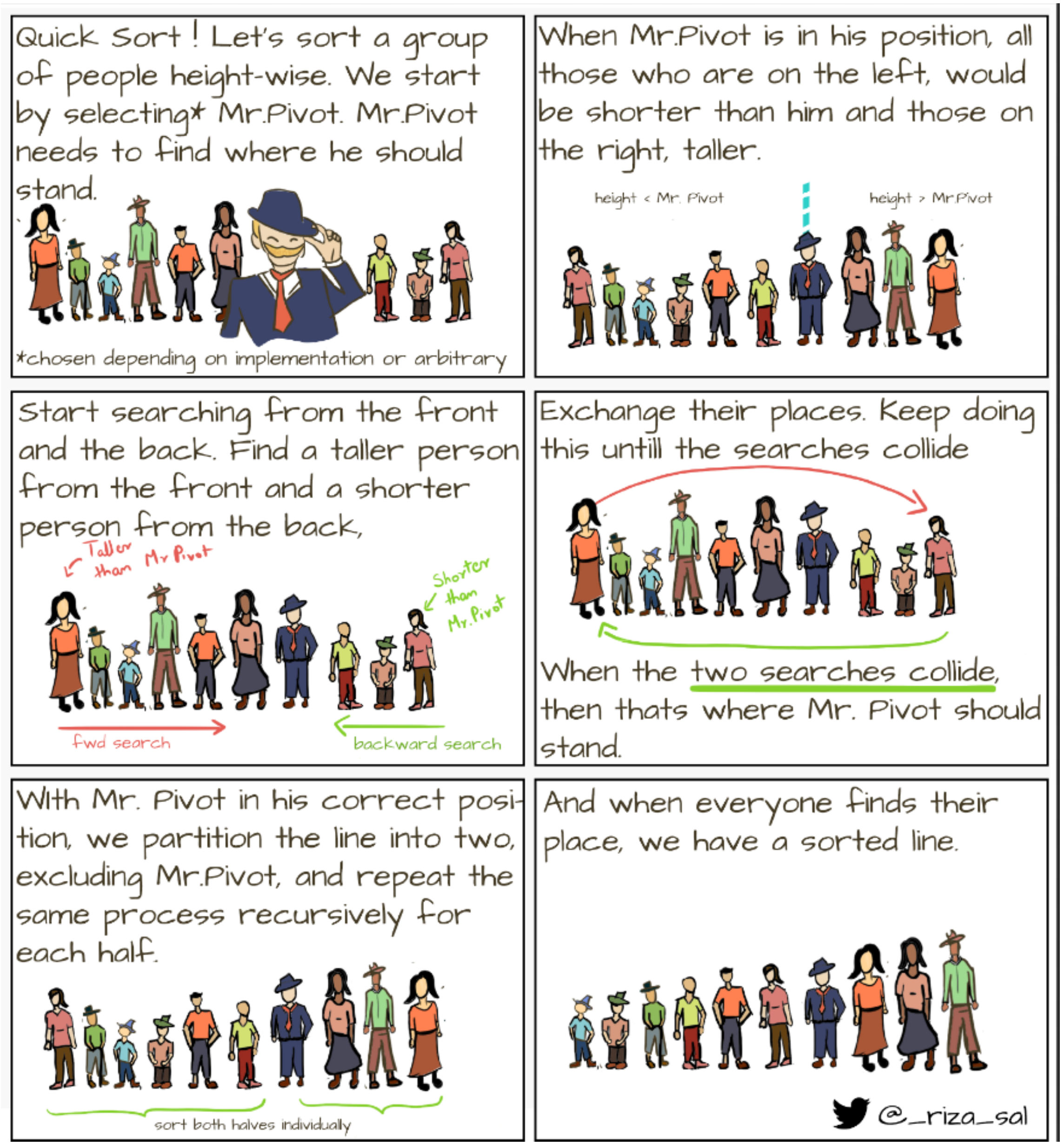
- Is very easy, but not very efficient
- Big number of redundant calls
- The size of calls to fib2 grows as fast as the rabbits
- Iterative version fib is much more efficient

Quicksort

A recursive sorting program which is very fast (as a rule)

Cartoon

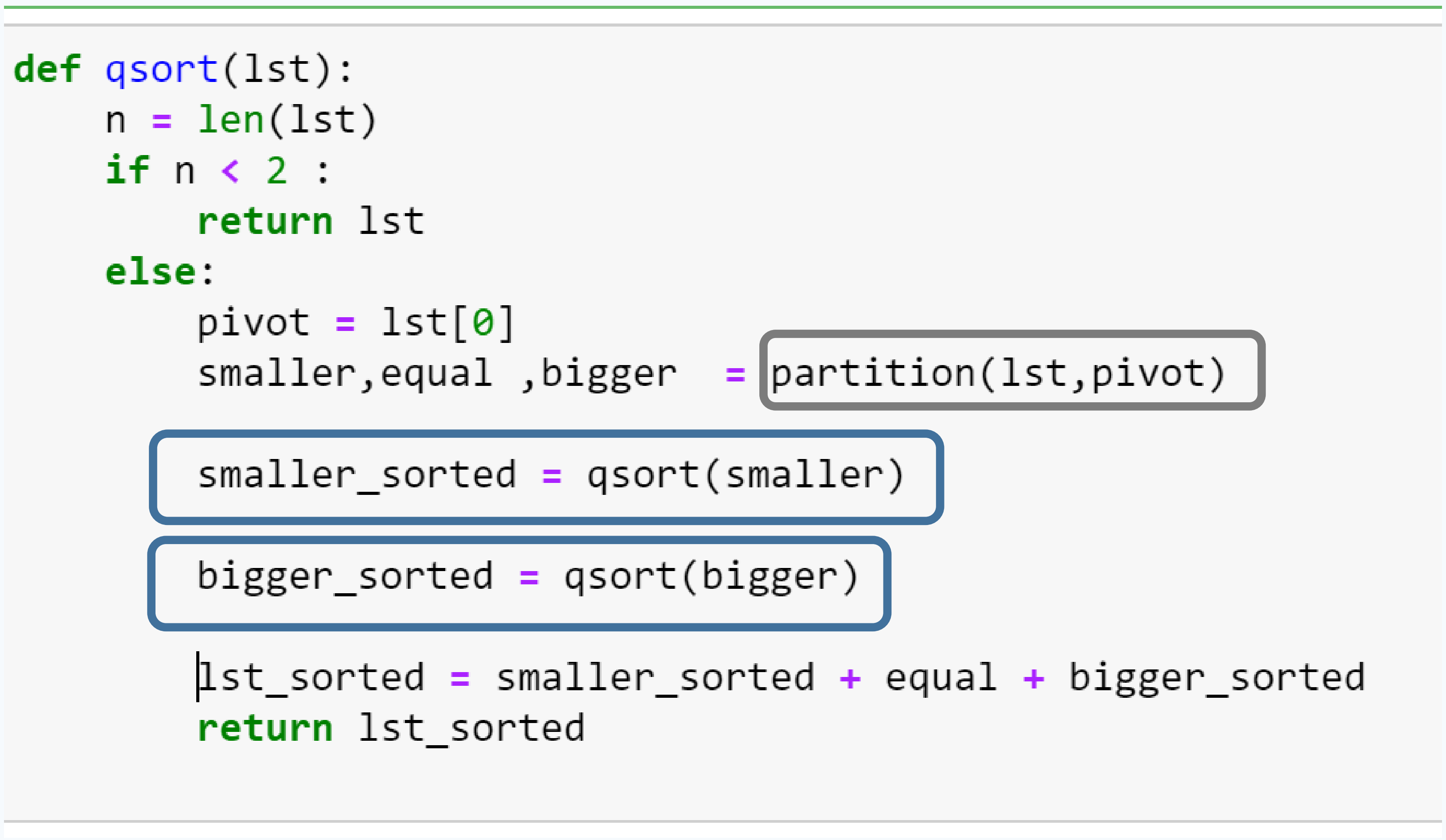
Quick sort


Quick Sort! Let's sort a group of people height-wise. We start by selecting* Mr.Pivot. Mr.Pivot needs to find where he should stand.
*chosen depending on implementation or arbitrary
When Mr.Pivot is in his position, all those who are on the left, would be shorter than him and those on the right, taller.
height < Mr. Pivot
height > Mr.Pivot
Start searching from the front and the back. Find a taller person from the front and a shorter person from the back,
Taller than Mr Pivot
Shorter than Mr.Pivot
fwd search
backward search
Exchange their places. Keep doing this untill the searches collide
When the two searches collide, then thats where Mr. Pivot should stand.
With Mr. Pivot in his correct position, we partition the line into two, excluding Mr.Pivot, and repeat the same process recursively for each half.
sort both halves individually
And when everyone finds their place, we have a sorted line.
@_riza_sal

The Idea of Quicksort

- Basic Idea is **Divide and Conquer**, an important programming paradigm
- If the list consists of one or zero elements, do nothing.
- If you split a list in two lists of similar size, named Smaller and Bigger, and a Pivot Element in between,
- Such that all elements of Smaller are smaller than Pivot, and all elements of Bigger are bigger than Pivot
- Just sort smaller and bigger separately, and concatenate both, with Pivot in between, at the end

qsort recursive function



```python
def qsort(lst):
    n = len(lst)
    if n < 2 :
        return lst
    else:
        pivot = lst[0]
        smaller,equal ,bigger  = partition(lst,pivot)

        smaller_sorted = qsort(smaller)

        bigger_sorted = qsort(bigger)

        lst_sorted = smaller_sorted + equal + bigger_sorted
        return lst_sorted
```

- Let's try to understand it step by step – starting with partition

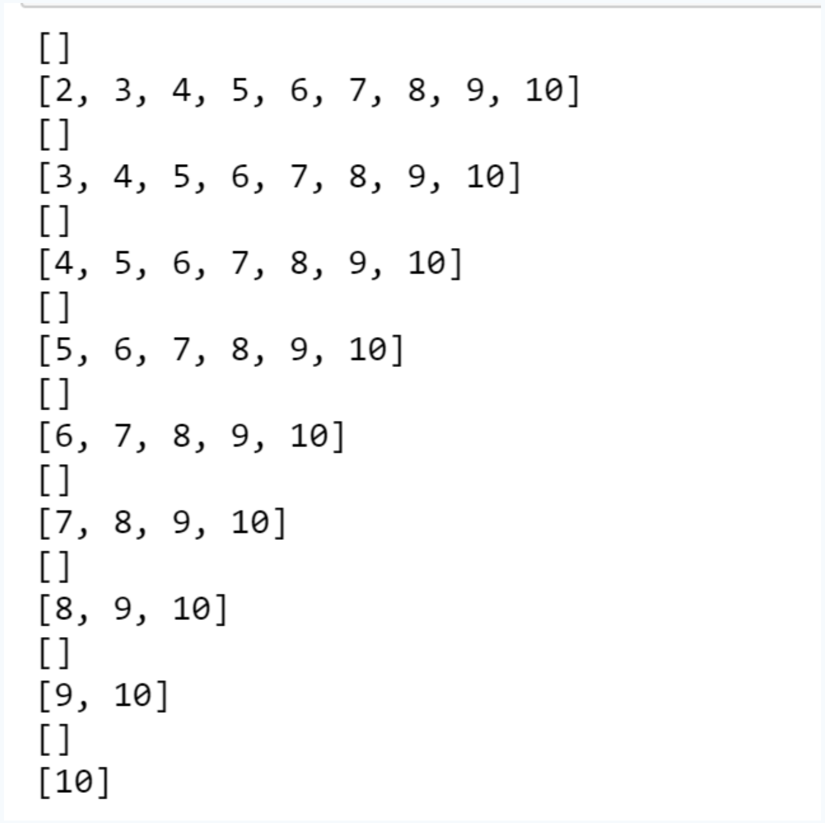
partition function

```python
def partition(lst,k):
    bigger = []
    smaller = []
    equal = []
    i =  0
    while i < len(lst) :
        if lst[i] < k:
            smaller.append(lst[i])
            i += 1
        elif lst[i] == k:
            equal.append(lst[i])
            i += 1
        else:
            bigger.append(lst[i])
            i += 1
    # merge 3 list:
    merged = [smaller,equal ,bigger]
    return merged
```

Performance of qsort

- qsort works wonderful, if smaller and bigger
  have roughly the same size
- In this case, the performance of qsort is the best
  what you can get for sorting
- Then qsort is a good example for the “divide and conquer” paradigm, going back to the Romans (“Divide et Impera”), an important paradigm for programming
- But if either “smaller” or “bigger” is comparatively small and
  the other is comparatively big, performance is not good
- Let’s see why

```
def qsort(lst):
    n = len(lst)
    if n < 2 :
        return lst
    else:
        pivot = lst[0]
        smaller,equal ,bigger  = partition(lst,pivot)

        print(smaller)
        print(bigger)

        smaller_sorted = qsort(smaller)

        bigger_sorted = qsort(bigger)

        lst_sorted = smaller_sorted + equal + bigger_sorted
        return lst_sorted
l_sorted = qsort([1,2,3,4,5,6,7,8,9,10])
```



```
[]
[2, 3, 4, 5, 6, 7, 8, 9, 10]
[]
[3, 4, 5, 6, 7, 8, 9, 10]
[]
[4, 5, 6, 7, 8, 9, 10]
[]
[5, 6, 7, 8, 9, 10]
[]
[6, 7, 8, 9, 10]
[]
[7, 8, 9, 10]
[]
[8, 9, 10]
[]
[9, 10]
[]
[10]
```

- Smaller is empty, bigger takes it all
- There is no “divide and conquer”
- This happens if the input list is already sorted
- How can we handle this problem?
- Answer: more intelligent selection of pivot

- Exercise: take the middle element of the list as pivot
- Hint:

  j = len(lst)//2

  pivot = lst[j]

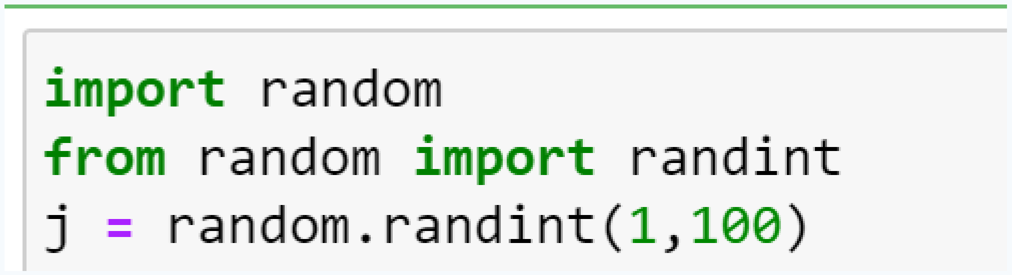
How to look at performance

- You can also introduce a global variable j
- Declare in the main program j = 0
- Declare in the function definition right at the beginning global j
- J = j + 1
- And look at j after calling the function

```
j = 0
def qsort(lst):
    global j
    j = j + 1
    n = len(lst)
    if n < 2 :
        return lst
    else:
        pivot_index = len(lst)//2
        pivot = lst[pivot_index]
        smaller,equal ,bigger  = partition(lst,pivot)
        smaller_sorted = qsort(smaller)
        bigger_sorted = qsort(bigger)
        lst_sorted = smaller_sorted + equal + bigger_sorted
        return lst_sorted
l_sorted = qsort([1,2,3,4,5,6,7,8,9,10])
print(j)
```

Generating long lists

- You can generate a long list easily with a for loop and :



```python
import random
from random import randint
j = random.randint(1,100)
```

- It’s up to you to program the for loop and generate the list, with, say, 100 elements

Classroom Exercise

- Write the recursive factorial function
- Write the recursive and iterative Fibonacci function
- Count the number of recursive calls of Fibonacci with a global variable j as described above
- Write the qsort function
- Count the number of function calls with a global variable j as described above for Fibonacci

Homework

- Todays homework is a bit more advanced
- Write a recursive function that accepts a positive integer as its argument and returns the sum of digits. You may use modulo and integer division
- Modify the qsort function by using an intelligent pivot, as suggested above
- Count the number of function calls with a global variable
- Generate a long list with random integers, apply qsort and count the number of function calls, both for the standard version and with intelligent pivot
- Star exercise: a palindrom list of numbers is a list which looks identical when read in both directions. [1, 2, 3, 4 , 3, 2, 1] is a palindrom. Write a recursive function which tests if a list is a palindrom.
  Hint: lst[0] gives the first element, lst[-1] the last. You can use pop to remove elements