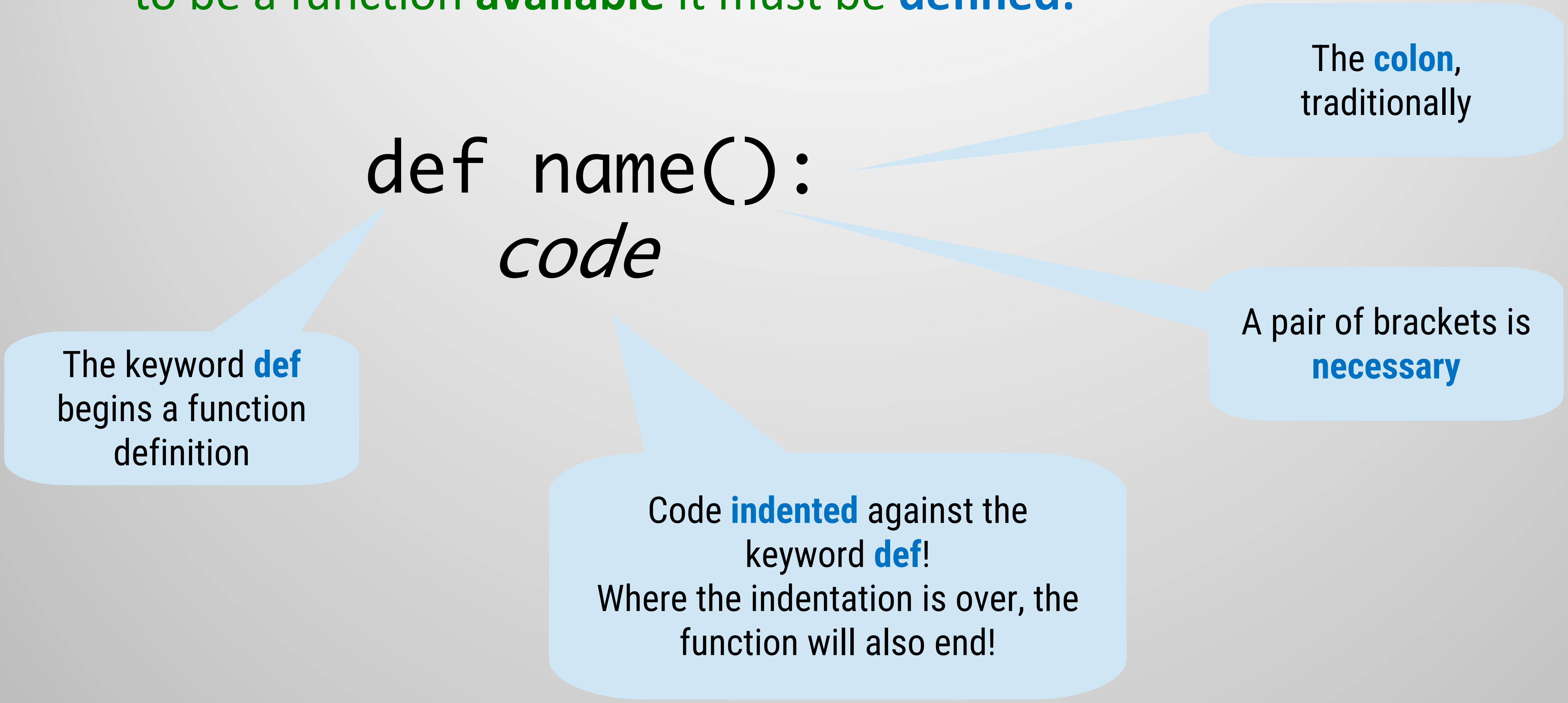
Python Programming Language – functions

Assoc. Prof. Krzysztof Małecki

kmalecki@zut.edu.pl
room 308 (WI2)

A function

- to be a function **available** it must be **defined:**



```
def name():
    code
```

The **colon**, traditionally

A pair of brackets is **necessary**

The keyword **def** begins a function definition

Code **indented** against the keyword **def**!
Where the indentation is over, the function will also end!

Example

```
def message():
    print('Enter the next value: ')

print('Start')
message()
print('Stop')
```

```
Start
Enter the next value:
Stop
```

Example

```
print('Start')
message()
print('Stop')

def message():
     print('Enter the next value: ')
```

Here, the **function** is **unknown!!!**

```
Start
Traceback (most recent call last):
  File "/Users/KM 1/Wykłady/Python/Python - Erasmus/Examples/Test1.py",
line 2, in <module> message()
NameError: name 'message' is not defined
```

Example

```
def message():
    print('Enter next value: ')

message = 1
message()
```

Having a function and a variable with **the same name** **is not desirable**

```
Start
Traceback (most recent call last):
  File "/Users/KM 1/Wykłady/Python/Python - Erasmus/Examples/Test1.py",
line 5, in <module> message()
TypeError: 'int' object is not callable
```

A function

- **functions** reveal their full power when they are **parameterized**

- a **parameter** is a specific variable that exists **only inside its function**

- the value of a parameter is set at the time the function is called by specifying **the corresponding argument**

Example

The function **parameter** will get the value of the corresponding **argument** ...

```
def message(number):
     print('Enter value number ' + str(number))

message(1)
```

... which is given here... (at the time of calling this function)

```
Enter value number 1
```

Example

```
def message(number):
     print('Enter value number ' + str(number))

message(1)
print(number)
```

A function parameter **only exists** **inside the function** and **only while the function is running**

```
Enter value number 1
Traceback (most recent call last):
  File "/Users/KM 1/Wykłady/Python/Python - Erasmus/Examples/Test1.py",
line 5, in <module> print(number)
NameError: name 'number' is not defined
```

Remember

- if you have a variable named as a certain parameter in a specified function, a mechanism called **override** will work
- when this function is running, the parameter name will **override** the variable name
- the name of this variable **will be exposed** when the function stops

Example

The name of the **parameter number** will **cover** the **variable named number**

```
def message(number):
     print('Enter value number ' + str(number))

number = 10
message(1)
print(number)
```

... which is given here...

```
Enter value number 1
10
```

A function with two parameters

```
def message(what, number):
     print('Enter ' + what + ' value ' + str(number))

message('amount',1)
message('tax',2)
```

The parameter **what** will represent the name of value **number**, so it will be a string.

```
Enter amount value 1
Enter tax value 2
```

What if the arguments get mixed up?

```
def message(what, number):
     print('Enter ' + what + ' value ' + str(number))

message(1,'amount')
message('tax',2)
```

```
Traceback (most recent call last):
  File "/Users/KM 1/Wykłady/Python/Python - Erasmus/Examples/Test1.py",
line 4, in <module> message(1, 'amount')
  File "/Users/KM 1/Wykłady/Python/Python - Erasmus/Examples/Test1.py",
line 2, in message print('Enter ' + what + ' value ' + str(number))
TypeError: can only concatenate str (not "int") to str
```

Remember

- the situation in which the **i-th argument** is assigned to the **i-th parameter** is called **positional parameter passing**

- in many programming languages this is the only way to associate arguments with parameters

- Python enables something extra ...

named parameter passing

- Python offers **a second method of passing parameters** that rules out such confusion as mixing up the parameters

- the parameter value is determined not by the argument position, but by **the explicitly specified name** of the corresponding argument

A function with two named parameters

```
def message(what, number):
     print('Enter ' + what + ' value ' + str(number))

message(number=1, what='amount')
message(what='tax', number=2)
```

```
Enter amount value 1
Enter tax value 2
```

Mixed parameter passing

- it is also possible **to mix both styles** of parameter passing

- however, positional parameters **are required to appear before** the named ones

- to show this, let's write a simple function with three parameters

Positional parameter passing

```
def sum(a,b,c):
    print(a, '+', b, '+', c, '=', a+b+c)

sum(1,2,3)
```

```
1 + 2 + 3 = 6
```

Named parameter passing

```
def sum(a,b,c):
    print(a, '+', b, '+', c, '=', a+b+c)

sum(c=1,a=2,b=3)
```

```
2 + 3 + 1 = 6
```

Mixed parameter passing

```
def sum(a,b,c):
    print(a, '+', b, '+', c, '=', a+b+c)

sum(3,c=1,b=2)
```

The arguments for the parameters **b** and **c** are passed as **named**

The argument for the parameter **a** is passed as **a positional**

```
3 + 2 + 1 = 6
```

Be careful…

```
def sum(a,b,c):
    print(a, '+', b, '+', c, '=', a+b+c)

sum(3,a=1,b=2)
```

It seems we are trying to set **a** parameter in two different ways…

```
Traceback (most recent call last):
  File "/Users/KM 1/Wykłady/Python/Python - Erasmus/Examples/Test1.py",
line 4, in <module>   sum(3,a=1,b=2)
TypeError: sum() got multiple values for argument 'a'
```

Correct code, although it doesn't make sense

```
def sum(a,b,c):
    print(a, '+', b, '+', c, '=', a+b+c)

sum(3,2,c=1)
```

The argument for the parameter **c** is passed as **named**

The arguments for the parameters **a** and **b** are passed as **positional**

```
3 + 2 + 1 = 6
```

Default parameters

- it is common for some parameters to use certain argument values more often than others

- in such cases they can be given **default values** → as a result the corresponding arguments can be omitted as long as their default value is acceptable

- let's say, the most common surname is "Smith"

- let's try to make use of this fact ...

A function with a default parameter

```
def welcome(name, surname='Smith'):
     print('Hello, I am ' + surname +\
           '... ' + name + ' ' + surname + '...')

welcome('James', 'Bond')
```

This is how to assign **a default value to a parameter**.
In this case, the lack of a matching argument
will not be a mistake!

The simplest situation

```
Hello, I am Bond... James Bond...
```

A function with a default parameter

```
def welcome(name, surname='Smith'):
     print('Hello, I am ' + surname +\
           '... ' + name + ' ' + surname + '...')

welcome(surname='Smith', name='John')
welcome('John')
```

Named parameters

It looks bad because instead of two arguments there is only one...
But the second parameter has a certain **default value**!

```
Hello, I am Smith... John Smith...
Hello, I am Smith... John Smith...
```

A function with a default parameter

```
def welcome(name='John', surname='Smith'):
     print('Hello, I am ' + surname +\
          '... ' + name + ' ' + surname + '...')

welcome()
welcome('Bruce')
welcome(surname='Willis')
```

It is also possible…

It is a positional argument. It is associated with the first parameter

But if we want to use the "non-default" value of the second parameter, we must use a named argument

```
Hello, I am Smith... John Smith...
Hello, I am Smith... Bruce Smith...
Hello, I am Willis... John Willis...
```

A function

- all functions presented so far had **the effect** → writing out some text

- in addition to the effect, functions can also have **a result**

- we use for that `return` statement

`Return` instruction – variant #1

- causes **that function ends** immediately and **return to the place** from which it was called
- if the function **only has an effect**, explicitly using the `return` statement **is optional**
- then `return` will be **executed implicitly** where the function ends

A function without `return` instruction

```
def func():
     print('Three…')
     print('Two…')
     print('One…')
     print('Boom!')

func()
```

```
Three…
Two…
One…
Boom!
```

A function with `return` instruction

```
def func():
      print('Three…')
      print('Two…')
      print('One…')
      print('Boom!')
      return

func()
```

Here, this statement is useless

```
Three…
Two…
One…
Boom!
```

because nothing comes out of it ☹

A function with `return` instruction

```
def func(explosion = True):
    print('Three…')
    print('Two…')
    print('One…')
    if not explosion:
        return
    print('Boom!')

func()
```

The explosion is default

```
Three…
Two…
One…
Boom!
```

A function with `return` instruction

```
def func(explosion = True):
    print('Three…')
    print('Two…')
    print('One…')
    if not explosion:
        return
    print('Boom!')

func(False)
```

```
Three…
Two…
One…
```

We left the function before the explosion… ☺

`Return` instruction – variant #2

```
return expression
```

- causes that **function ends immediately** and **return** to the place from which it was called

- in addition, the **function will return** as its result the value of the expression behind the `return` instruction

A function with `return` instruction

```
def func():
      return 5

x = func()
print('The result of func(): ', x)
```

```
The result of func(): 5
```

An extremely interesting data... None

- a data with the value None is completely useless - it doesn't represent any meaningful value
- consequently, such data cannot participate in any calculations

```
>>> print (None + 2)
```

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(None+2)
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

None

- the None data can be assigned to any variable to indicate that the variable has not useful content (it is "empty")
- variables can also be compared to None to diagnose their state

```
value = None

if value == None:
     print('...')
```

Example

```
value = 2

if value != None:
     print('This information is useful')
```

```
This information is useful
```

Remember

if a certain function **does not return a value** explicitly using

```
return expression
```

it means, the function **will implicitly return**

None

Example

```
def func(n):
      if(n % 2 == 0):
            return True

print(func(2))
print(func(1))
```

```
True
None
```

Example – the function can return a string

```
def func(n):
    if(n % 2 == 0):
        return 'even'
    else:
        return 'odd'

print(func(2))
print(func(1))
```

```
even
odd
```

A list as a parameter/argument of the function

- Remember, if you pass a list as an argument to a function, the parameter must be processed as if it were a list

```
def func(l):
    sum = 0
    for el in l:
        sum += el
    return sum

print(func([5,4,3]))
```

```
12
```

Example

```
def func(l):
    sum = 0
    for el in l:
        sum += el
    return sum

print(func(5))
```

Is it a list???

```
Traceback (most recent call last):
  File "/Users/KM 1/Wykłady/Python/Python - Erasmus/Examples/Test1.py",
line 13, in <module>  print(func(5))
  File "/Users/KM 1/Wykłady/Python/Python - Erasmus/Examples/Test1.py",
line 5, in func     for el in l:
TypeError: 'int' object is not iterable
```

Example – the function can return a list

```
def func(n):
    list = []
    for i in range(0,n):
        list.insert(0,i)
    return list

print(func(5))
```

```
[4, 3, 2, 1, 0]
```

*args and **kwargs

- `*args` and `**kwargs` allow you to pass an unspecified number of arguments to a function,
- when writing the function definition, you do not need to know how many arguments will be passed to your function,
- writing `*args` and `**kwargs` is just a convention,
- it is not necessary to write `*args` or `**kwargs`. Only the `*` (asterisk) is necessary. You could have also written `*var` and `**vars`

Example (*args)

```
def test_var_args(f_arg, *args):
    print("first normal arg:", f_arg)
    for arg in args:
        print("another arg through *argc:", arg)

test_var_args('first', 'second', 'python', 'test')
```

```
first normal arg: first
another arg through *argc: second
another arg through *argc: python
another arg through *argc: test
```

**kwargs

- `**kwargs` allows you to pass **keyworded** variable length of arguments to a function,
- You should use **kwargs if you want to handle **named arguments** in a function.

Example (**kwargs)

```
def test_var_kwargs(**kwargs):
    for key, value in kwargs.items():
        print("{0} = {1}".format(key, value))

test_var_kwargs(name1 = 'first', name2 ='second')
```

```
name1 = first
name2 = second
```

Example (**kwargs)

```
def concatenate(**kwargs):
    result = ""
    # Iterating over the Python kwargs dictionary
    for arg in kwargs.values():
        result += arg
    return result

print(concatenate(a="Python", b="Is", c="Great", d="!"))
```

If you don’t specify `.values()` your function will iterate over the keys of your Python kwargs dictionary, returning the wrong result!!!

```
PythonIsGreat!
```

Let us now consider the range of variable names

- the scope of a variable name means **all the places in the program** where **we can use** the variable

- we already know that the scope of the function parameter is this function (and nothing else)

- …and what is the scope of the variable used outside the function?

Example

```
def func():
    print("Do I know this variable? ", variable)

variable = 1
func()
print(variable)
```

```
Do I know this variable?  1
1
```

Example

```
def func():
    variable = 2  ←
    print("Do I know this variable? ", variable)

variable = 1
func()
print(variable)
```

```
Do I know this variable?  2  ←
1
```

Remember

- if a function uses the name of a certain variable, the name **overrides** the scope of the variable **defined outside the function**
- override works for the whole function, even before the variable is set to the value

```
def func():
    print("Do I know this variable? ", variable)
    variable = 2

variable = 1
func()
print(variable)
```

```
Traceback (most recent call last):
  File "/Users/KM 1/Wykłady/Python/Python - Erasmus/Examples/Test1.py",
line 10, in <module>     func()
  File "/Users/KM 1/Wykłady/Python/Python - Erasmus/Examples/Test1.py",
line 3, in func     print("Do I know this variable? ", variable)
UnboundLocalError: local variable 'variable' referenced before assignment
```

`global` instruction

```
global variable
global variable1,variable2,…
```

- tells Python that we do not wish to override the external name
- we want to have full access to the variable
- if a function marks a certain variable as global, then it will not override this variable

Example

```python
def func():
    global variable
    print("Do I know this variable? ", variable)
    variable = 2

variable = 1
func()
print(variable)
```

```
Do I know this variable?  1
2
```

Function – the value of the argument

- swapping a parameter value does not affect the value of the argument
- it follows that the value of the argument is passed to the function, **not the argument itself**

```
def func(n):
    print("(func) - I have gotten: ", n)
    n+=1
    print("(func) - Now, I have: ", n)

variable = 3
func(variable)
print("variable = ", variable)
```

```
(func) - I have gotten:  3
(func) - Now, I have:  4
variable =  3
```

Example with a list

```
def func(l):
    print(l)
    l = []

list = [1,2,3]
func(list)
print(list)
```

```
[1, 2, 3]
[1, 2, 3]
```

Example with a list

```python
def func(l):
    print(l)
    del l[0]

list = [1,2,3]
func(list)
print(list)
```

```
[1, 2, 3]
[2, 3]
```

What's happen?

Remember

- if the argument is a list variable, **changing the contents of that list** through the parameter will also be visible in the argument

- **note:** changing the content of the list, **not changing the list itself**!

Some examples

- BMI (Body Mass Index)…
- Is it a triangle?...
- Factorial…
- Fibonacci sequence…

BMI

$$BMI = \frac{weight\ [kg]}{growth^2\ [m]}$$

It can be seen that:

- the function should have 2 parameters: weight and growth
- the function should return the **BMI** calculated according to the formula
- the most natural name for this function will of course be `BMI` ... or `bmi` (a matter of taste)

BMI – v.1

```python
def BMI(weight,growth):
        return weight / growth**2

print(BMI(76.5, 1.75))
```

```
24.979591836734695
```

A digression

- in **professional Python programming**, it is good to put a string in the function that informs **what the function is** and **what it is for**

- in particular, it is worth describing the purpose and role of the parameters

- it can be done like this…

BMI – v.1 (PRO)

```
def BMI(weight,growth):
    '''Function calculates BMI from your weight (kg) and height (m)'''
    return weight / growth**2

print(BMI(76.5, 1.75))
```

```
24.979591836734695
```

what have we missed?

- our function assumes that the data passed into it is always meaningful

- it is worth making the function check if the received arguments are related to the reality

BMI – v.2

```
def BMI(weight,growth):
    if growth < 0.5 or growth > 2.5:
        return None
    if weight < 20 or weight > 300:
        return None
    return weight / growth**2

print(BMI(376.5, 1.75))
```

```
None
```

what else have we missed?

- we have omitted a large part of the world that does not use meters and kilograms – for them our function will be very difficult to use

- so we will help them – we will start with pounds:

1 British pound [lb] = 0.45359237 kg

let's write an auxiliary function

```
def PoundsForKilos(lb):
    return lb * 0.45359237

print(PoundsForKilos(1))
```

```
0.45359237
```

What else?

- now it's time for feet and inches:

1 foot [ft] = 0.3048 m

1 inch [in] = 2.54 cm = 0.0254 m

let's write the next auxiliary function

```
def FtInToMetres(ft,inch):
    return ft * 0.3048 + inch * 0.0254

print(FtInToMetres(1,1))
```

```
0.3302
```

let's write better auxiliary function

```
def FtInToMetres(ft,inch = 0.0):
    return ft * 0.3048 + inch * 0.0254

print(FtInToMetres(6))
```

Maybe a measure only in feet is enough?

```
1.8288000000000002
```

BMI – v.3

```
def PoundsForKilos(lb):
        return lb * 0.45359237
def FtInToMetres(ft,inch=0.0):
        return ft * 0.3048 + inch * 0.0254
def BMI(weight,growth):
    if growth < 0.5 or growth > 2.5:
        return None
    if weight < 20 or weight > 300:
        return None
    return weight / growth**2

print(BMI(growth=FtInToMetres(5,7), weight=PoundsForKilos(176)))
```

What is the **BMI** of an individual **5'7"** tall and weighing **176 lbs**?

```
27.565214082533313
```

Triangle

- let's see if certain three lines can form a triangle
- we know that in a triangle the sum of the lengths of any two sides must be greater than the length of the third side
- the function will have 3 parameters – one on each side of the triangle
- the function will result `True` (this can be a triangle) or `False` (this cannot be a triangle)

Is it a triangle? v.1

```
def IsTriangle(a,b,c):
     if a + b <= c:
          return False
     if b + c <= a:
          return False
     if c + a <= b:
          return False
     return True

print(IsTriangle(1,1,1))
print(IsTriangle(1,1,3))
```

```
True
False
```

Is it a triangle? v.2

```
def IsTriangle(a,b,c):
     return a + b > c and b + c > a and c + a > b

print(IsTriangle(1,1,1))
print(IsTriangle(1,1,3))
```

```
True
False
```

PRO ver.

- the user is asked to enter three numbers
- one should give them in one line!
- we will tell her/him whether it is possible to make a triangle out of such values

Is it a triangle? v.3

```
def IsTriangle(a,b,c):
     return a + b > c and b + c > a and c + a > b

sides=[ ]
while len(sides) != 3:
    inputLine=input("Enter three numbers (in one line, separate them with spaces: ")
    sides=inputLine.split()

a=float(sides[0])
b=float(sides[1])
c=float(sides[2])

if IsTriangle(a,b,c):
    print("It could be a triange!")
else:
    print("It could not be a triangle")
```

Function calculates the factorial

- the factorial – the product of an integer and all the integers below
- the function will have one parameter: `n`
- we know:

```
0! = 1
1! = 1
```

- we also know:

```
n! = 1 · 2 · 3 · … · n-1 · n
```

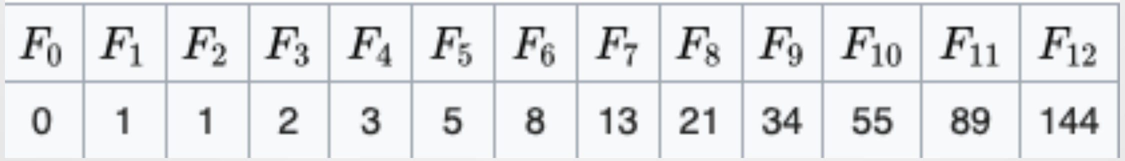
- and that computing the factorial from negative data is a mistake!

Example

```
def Factorial(n):
        if n < 0:
                return None
        if n < 2:
                return 1
        product = 1
        for i in range(2,n+1):
                product *= i
        return product

for n in range(1,6):
        print(n, Factorial(n))
```

```
1 1
2 2
3 6
4 24
5 120
```

Function for calculating the Fibonacci sequence



F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}
0	1	1	2	3	5	8	13	21	34	55	89	144

- the function will have one parameter: n
- we know:

$$\mathtt{fib}_1 = 1$$
$$\mathtt{fib}_2 = 1$$

- we also know:

$$\mathtt{fib}_n = \mathtt{fib}_{n-1} + \mathtt{fib}_{n-2}$$

Example

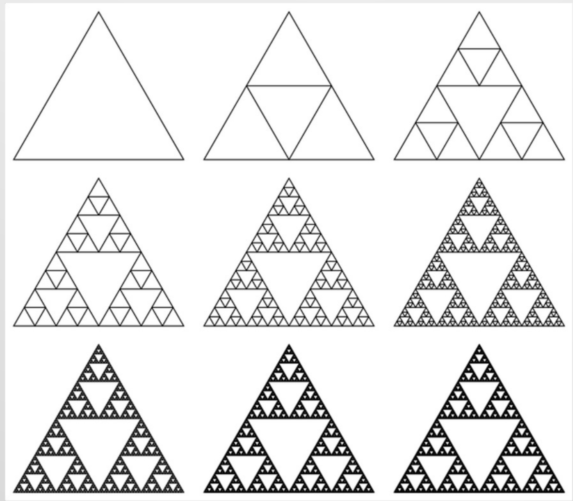
```
def Fib(n):
        if n < 1:
                return None
        if n < 3:
                return 1
        el1 = el2 = 1
        sum = 0
        for i in range(3,n+1):
                sum = el1 + el2
                el1, el2 = el2, sum
        return sum

for n in range(1,10):
        print(Fib(n),end='  ')
else:
        print('...')
```

```
1  1  2  3  5  8  13  21  34  ...
```

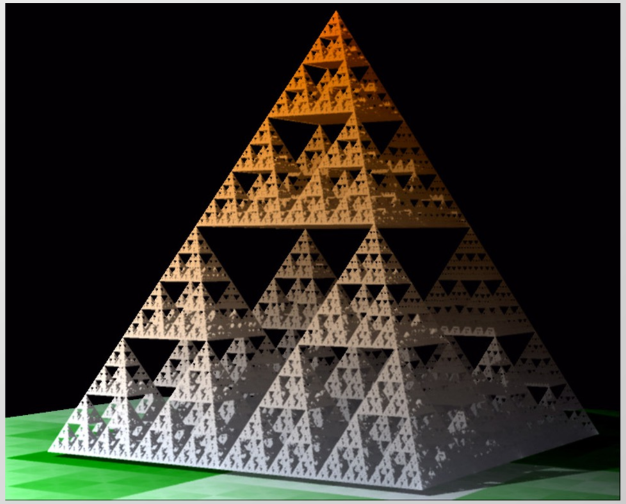
Recursion

- in computer science, recursion most often means that a certain function **calls itself**

visual recursion



Sierpinski's triangles



Sierpinski's pyramid

Factorial – recursion

$$0! = 1$$

$$1! = 1$$

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n-1 \cdot n$$

Does factorial have a recursion nature?

Note, that:

$$1 \cdot 2 \cdot 3 \cdot \ldots \cdot n-1 = (n-1)!$$

it follows that:

$$n! = (n-1)! \cdot n$$

Factorial – recursion

```
def Factorial(n):
        if n < 0:
                return None
        if n < 2:
                return 1
        return n * Factorial(n-1)

for n in range(1,6):
        print(n, Factorial(n))
```

```
1 1
2 2
3 6
4 24
5 120
```

Fibonacci sequence - recursion

```
def Fib(n):
        if n < 1:
                return None
        if n < 3:
                return 1
        return Fib(n-1) + Fib(n-2)

for n in range(1,10):
        print(Fib(n),end='  ')
else:
        print('...')
```

```
1   1   2   3   5   8   13   21   34   ...
```

Remember

- recursion allows you to write many algorithms **shorter and more readable**

- unfortunately, recursive algorithms are usually **slower** than those working iterative

Thank you for your attention

see you at the next lecture